NetRexx Programming Guide

RexxLA

Version 3.03 of June 29, 2013

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 the formal definition for the language, documenting its syntax and semantics, and prescribing minimal functionality for language implementors. It is the definitive answer to any question on the language, and as such, is subject to approval of the NetRexx Architecture Review Board on any release of the language (including its NRL).
NJPipes Reference	The Data Flow oriented companion to NetRexx, with its CMS Pipes compatible syntax, is documented in this manual. It discusses installing and running Pipes for NetRexx, and has ample examples of defining your own stages in NetRexx.

Typographical conventions

In general, the following conventions have been observed in the NetRexx publications:

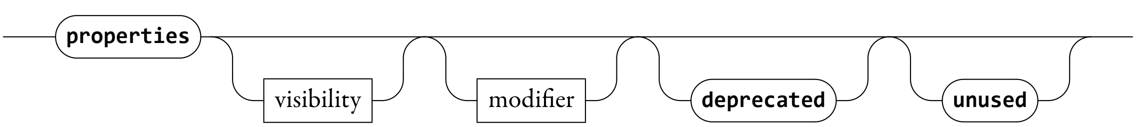
- Body text is in this font
- Examples of language statements are in a **bold** type
- Variables or strings as mentioned in source code, or things that appear on the console, are in a `typewriter` type
- Items that are introduced, or emphasized, are in an *italic* type
- Included program fragments are listed in this fashion:

Listing 1: Example Listing

```
-- salute the reader
say 'hello reader'
```

- Syntax diagrams take the form of so-called *Railroad Diagrams* to convey structure, mandatory and optional items

Properties



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 *Quick Beginnings* 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. This has had its effect on the volume of this book, which means that unlike the other publications in the series, it is probably not a good idea to print it out in its entirety; its size will relegate it to being used electronically.

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 touch.

A big IOU goes out to Alan Sampson, who singlehandedly contributed more then one

[1] because they are unacknowledged in the original publications

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 take out offending passages or paraphrase them beyond recognition. 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 time-sharing 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 left 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.

On June 8th, 2011, IBM 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, NetRexxinstructions 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.

Listing 2.1: Hello Stranger

```
/* 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 MacOSX, 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 77.

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?

Listing 2.2: RAH Exec

```
/* 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

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.

Listing 2.3: Long 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.

Listing 2.4: Long lines with string concatenation without space

```
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:

Listing 2.5: Without a loop

```
say  1 'euro equals' 1 * 2.34 'dollars'
say  2 'euro equals' 2 * 2.34 'dollars'
say  3 'euro equals' 3 * 2.34 'dollars'
say  4 'euro equals' 4 * 2.34 'dollars'
say  5 'euro equals' 5 * 2.34 'dollars'
say  6 'euro equals' 6 * 2.34 'dollars'
say  7 'euro equals' 7 * 2.34 'dollars'
say  8 'euro equals' 8 * 2.34 'dollars'
say  9 'euro equals' 9 * 2.34 'dollars'
say 10 'euro equals' 10 * 2.34 '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**[2] statement.

Listing 2.6: With a loop

```
loop i=1 to 100
  say i 'euro equals' i * 2.34 '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.

A few examples of what we can do with this:

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

  Listing 2.7: Loop Forever

  ```
  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

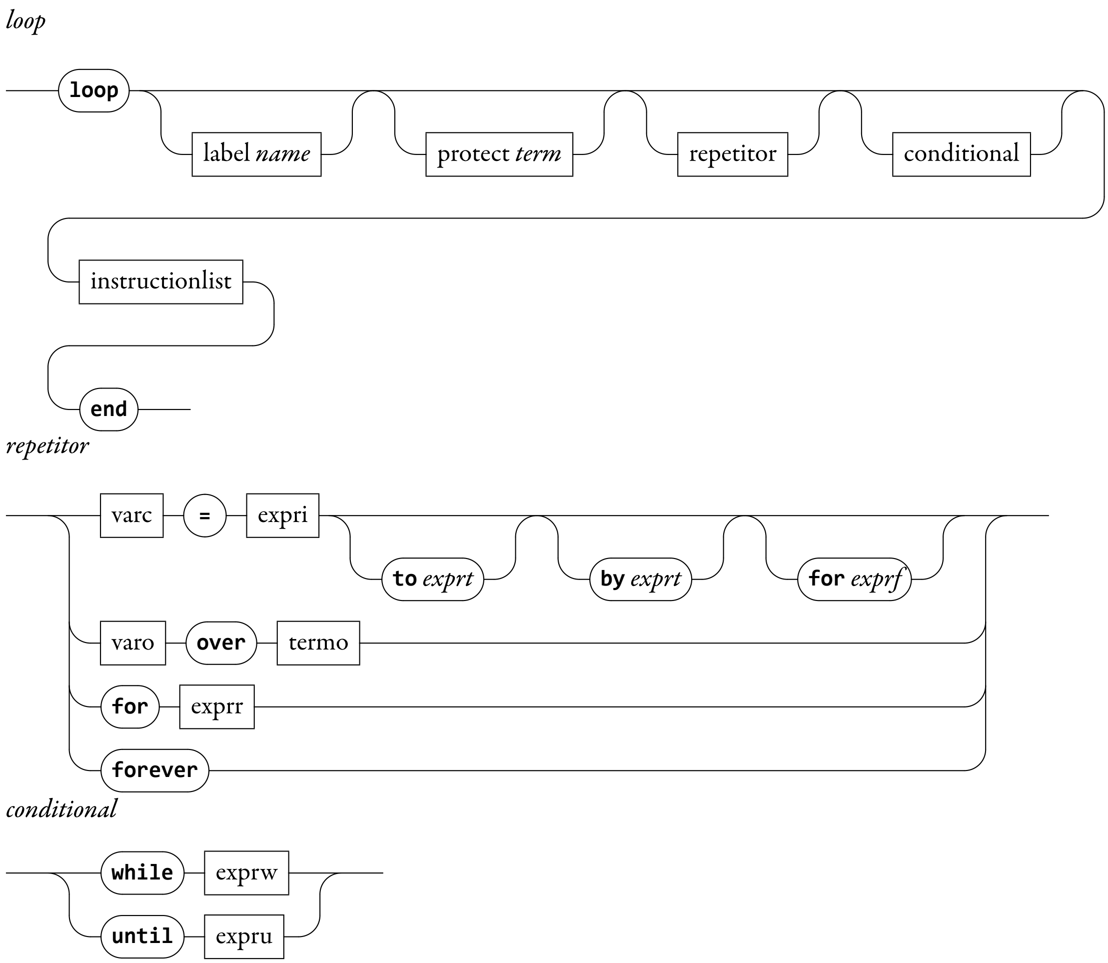
  Listing 2.8: Loop for a fixed number of times without loop index variable

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

- Looping back into the value of the loop index variable

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

FIGURE 1: Loop



Listing 2.9: Loop Forever

```
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.

Listing 2.10: NetRexx Special Variables

```
/* NetRexx */
options replace format comments java crossref savelog 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 27, 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 5 on 17, 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 Options

There are a number of options for the translator, some of which can be specified on the translator command line, and others also in the program source on the **option** statement. In the following table, c stands for *commandline only*, s stands for *source* and b stands for *both*.

TABLE 1: Options

Option	Meaning	Place
-arg words	interpret; remaining words are arguments	c
-binary	classes are binary classes	b
-classpath	specify a classpath	c
-compile	compile (default; -nocompile implies -keep)	c
-comments	copy comments across to generated .java	b
-compact	display error messages in compact form	b
-console	display messages on console (default)	c
-crossref	generate cross-reference listing	b
-decimal	allow implicit decimal arithmetic	b
-diag	show diagnostic messages	b
-exec	interpret with no argument words	c
-explicit	local variables must be explicitly declared	b
-format	format output file (pretty-print)	b
-java	generate Java source code for this program	b
-keep	keep any completed .java file (as xxx.java.keep)	c
-keepasjava	keep any completed .java file (as xxx.java)	c
-logo	display logo (banner) after starting	b
-prompt	prompt for new request after processing	c
-savelog	save messages in NetRexxC.log	c
-replace	replace .java file even if it exists	b
-sourcedir	force output files to source directory	b
-strictargs	empty argument lists must be specified as ()	b
-strictassign	assignment must be cost-free	b
-strictcase	names must match in case	b
-strictimport	all imports must be explicit	b
-strictmethods	superclass methods are not compared to local methods for best match	b

Continued on next page

Table 1 – *continued from previous page*

-strictprops	even local properties must be qualified	b
-strictsignal	signals list must be explicit	b
-symbols	include symbols table in generated .class files	b
-time	display timings	c
-trace[n]	trace stream [1 or 2], or 0 for NOTRACE	b
-utf8	source file is in UTF8 encoding	b
-verbose[n]	verbosity of progress reports [0-5]	b
-warnexit0	exit with a zero returncode on warnings	c

Options valid for the options statement and on the commandline

These are the options that can be used on the **options** statement:

-binary All classes in this program will be binary classes. In binary classes, literals are assigned binary (primitive) or native string types, rather than NetRexx types, and native binary operations are used to implement operators where appropriate, as described in “Binary values and operations”. In classes that are not binary, terms in expressions are converted to the NetRexx string type, Rexx, before use by operators.

-comments Comments from the NetRexx source program will be passed through to the Java output file (which may be saved with a .java.keep or .java extension by using the -keep and -keepasjava command options, respectively).

-compact Requests that warnings and error messages be displayed in compact form. This format is more easily parsed than the default format, and is intended for use by editing environments. Each error message is presented as a single line, prefixed with the error token identification enclosed in square brackets. The error token identification comprises three words, with one blank separating the words. The words are: the source file specification, the line number of the error token, the column in which it starts, and its length. For example (all on one line):

```
[D:\test\test.nrx 3 8 5] Error: The external name
'class' is a Java reserved word, so would not be
usable from Java programs
```

Any blanks in the file specification are replaced by a null ('\0') character. Additional words could be added to the error token identification later.

-crossref Requests that cross-reference listings of variables be prepared, by class.

-decimal Decimal arithmetic may be used in the program. If nodecimal is specified, the language processor will report operations that use (or, like normal string comparison, might use) decimal arithmetic as an error. This option is intended for performance-critical programs where the overhead of inadvertent use of decimal arithmetic is unacceptable.

-diag Requests that diagnostic information (for experimental use only) be displayed. The diag option word may also have side-effects.

-explicit Requires that all local variables must be explicitly declared (by assigning them a type but no value) before assigning any value to them. This option is intended to permit the enforcement of “house styles” (but note that the NetRexx compiler always checks for variables which are referenced before their first assignment, and warns of variables which are set but not used).

-format Requests that the translator output file (Java source code) be formatted for improved readability. Note that if this option is in effect, line numbers from the input file will not be preserved (so run-time errors and exception trace-backs may show incorrect line numbers).

-java Requests that Java source code be produced by the translator. If nojava is specified, no Java source code will be produced; this can be used to save a little time when checking of a program is required without any compilation or Java code resulting.

-logo Requests that the language processor display an introductory logotype sequence (name and version of the compiler or interpreter, etc.).

-sourcedir Requests that all .class files be placed in the same directory as the source file from which they are compiled. Other output files are already placed in that directory. Note that using this option will prevent the -run command option from working unless the source directory is the current directory.

-strictargs Requires that method invocations always specify parentheses, even when no arguments are supplied. Also, if strictargs is in effect, method arguments are checked for usage – a warning is given if no reference to the argument is made in the method.

-strictassign Requires that only exact type matches be allowed in assignments (this is stronger than Java requirements). This also applies to the matching of arguments in method calls.

-strictcase Requires that local and external name comparisons for variables, properties, methods, classes, and special words match in case (that is, names must be identical to match).

-strictimport Requires that all imported packages and classes be imported explicitly using import instructions. That is, if in effect, there will be no automatic imports, except those related to the package instruction.

-strictmethods Superclass methods are not compared to local methods for best match.

-strictprops Requires that all properties, including those local to the current class, be qualified in references. That is, if in effect, local properties cannot appear as simple names but must be qualified by this. (or equivalent) or the class name (for static properties).

-strictsignal Requires that all checked exceptions signalled within a method but not caught by a catch clause be listed in the signals phrase of the method instruction.

-symbols Symbol table information (names of local variables, etc.) will be included in any generated .class file. This option is provided to aid the production of classes that are easy to analyse with tools that can understand the symbol table information. The use of this option increases the size of .class files.

-trace, -traceX If given as **-trace**, **-trace1**, or **-trace2**, then trace instructions are accepted. The trace output is directed according to the option word: **-trace1** requests that trace output is written to the standard output stream, **-trace** or **-trace2** imply that the output should be written to the standard error stream (the default).

-utf8 If given, clauses following the options instruction are expected to be encoded using UTF-8, so all Unicode characters may be used in the source of the program. In UTF-8 encoding, Unicode characters less than '\u0080' are represented using one byte (whose most-significant bit is 0), characters in the range '\u0080' through '\u07FF' are encoded as two bytes, in the sequence of bits:

```
110xxxxx 10xxxxxx
```

where the eleven digits shown as x are the least significant eleven bits of the character, and characters in the range '\u0800' through '\uFFFF' are encoded as three bytes, in the sequence of bits:

```
1110xxxx 10xxxxxx 10xxxxxx
```

where the sixteen digits shown as x are the sixteen bits of the character. If noutf8 is given, following clauses are assumed to comprise only Unicode characters in the range '\x00' through '\xFF', with the more significant byte of the encoding of each character being 0. Note: this option only has an effect as a compiler option, and applies to all programs being compiled. If present on an options instruction, it is checked and must match the compiler option (this allows processing with or without utf8 to be enforced).

-verbose, -verboseX Sets the "noisiness" of the language processor. The digit X may be any of the digits 0 through 5; if omitted, a value of 3 is used. The options **-noverbose** and **verbose0** both suppress all messages except errors and warnings

Options valid on the commandline

The translator also implements some additional option words, which control compilation features. These cannot be used on the **options** instruction[3], and are:

-arg The **-arg** *words* option is used when interpreting programs, it indicates that after the **-arg** statement, commandline arguments for ther interpreted program follow

-classpath The -classpath option allows dynamic specification of the classpath used by the NetRexxC compiler without having to depend on the CLASSPATH environment variable. (since: NetRexx 3.02) .

-exec The **-exec** *words* option is used when interpreting programs. With this option, no commandline arguments are possible.

-keep keep the intermediate *.java* file for each program. It is kept in the same directory as the NetRexx source file as *xxx.java.keep*, where *xxx* is the source file name. The file will also be kept automatically if the *javac* compilation fails for any reason.

-keepasjava keep the intermediate *.java* file for each program. It is kept in the same directory as the NetRexx source file as *xxx.java*, where *xxx* is the source file name. Implies -replace. Note: use this option carefully in mixed-source projects where you might have .java source files around.

-nocompile do not compile (just translate). Use this option when you want to use a different Java compiler. The *.java* file for each program is kept in the same directory as the NetRexx source file, as the file *xxx.java.keep* (where *xxx* is the source file name).

-noconsole do not display compiler messages on the console (command display screen). This is usually used with the *savelog* option.

-savelog write compiler messages to the file *NetRexxC.log*, in the current directory. This is often used with the *noconsole* option.

-time display translation, *javac* or *ecj* compile, and total times (for the sum of all programs processed).

-run run the resulting Java class as a stand-alone application, provided that the compilation had no errors.

-warnexit0 Exit the translator with returncode 0 even if warnings are issued. Useful with build tools that would otherwise exit a build.

[3] Although at the moment, there will be no indication of this

4

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 both cases, interpreted or compiled, the NetRexx translator adds the necessary overhead 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 using scripting mode, this error message nearly always indicates an error that occurred before the first class statement.

5

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 way to explain this feature, because historically, the compiler was first (1996) and the interpretation facility was added later (in 2000).

6

NetRexx as a Compiled Language

6.1 Compiling from another program

The translator may be called from a NetRexx or Java program directly, by invoking the *main* method in the *org.netrexx.process.NetRexxC* class described as follows:

Listing 6.1: Invoking NetRexxC.main

```
method main(arg=Rexx, log=PrintWriter null) static returns int
```

The *Rexx* string passed to the method can be any combination of program names and options (except *-run*), as described above. Program names may optionally be enclosed in double-quote characters (and must be if the name includes any blanks in its specification).

A sample NetRexx program that invokes the NetRexx compiler to compile a program called *test* is:

Listing 6.2: Compiletest

```
/* compiletest.nrx */
s='test -keep -verbose4 -utf8'
say org.netrexx.process.NetRexxC.main(s)
```

Alternatively, the compiler may be called using the method:

Listing 6.3: Calling with Array argument

```
method main2(arg=String[], log=PrintWriter null) static returns int
```

in which case each element of the *arg* array must contain either a name or an option (except *-run*, as before). In this case, names must *not* be enclosed in double-quote characters, and may contain blanks.

For both methods, the returned *int* value will be one of the return values described above, and the second argument to the method is an optional *PrintWriter* stream. If the *PrintWriter* stream is provided, translator messages will be written to that stream (in addition to displaying them on the console, unless *-noconsole* is specified). It is the responsibility of the caller to create the stream (autoflush is recommended) and to close it after calling the compiler. The *-savelog* compiler option is ignored if a *PrintWriter* is provided (the *-savelog* option normally creates a *PrintWriter* for the file *NetRexxC.log*).

Note: NetRexxC is thread-safe (the only static properties are constants), but it is not known whether *javac* is thread-safe. Hence the invocation of multiple instances of NetRexxC on different threads should probably specify *-nocompile*, for safety.

6.2 Compiling from memory strings

Programs may also be compiled from memory strings by passing an array of strings containing programs to the translator using these methods:

Listing 6.4: From Memory

```
method main(arg=Rexx, programarray=String[], log=PrintWriter null) static returns int
method main2(arg=String[], programarray=String[], log=PrintWriter null) static returns int
```

Any programs passed as strings must be named in the arg parameter before any programs contained in files are named. For convenience when compiling a single program, the program can be passed directly to the compiler as a String with this method:

Listing 6.5: With String argument

```
method main(arg=Rexx, programstring=String, logfile=PrintWriter null) constant returns int
```

Here is an example of compiling a NetRexx program from a string in memory:

Listing 6.6: Example of compiling from String

```
import org.netrexx.process.NetRexxC
program = "say 'hello there via NetRexxC'"
NetRexxC.main("myprogram",program)
```

7

Calling non-JVM programs

Although NetRexx currently misses the **Address** facility that Classic Rexx and Object Rexx do have, 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.

Listing 7.1: Calling Non-JVM Programs

```
/* script\NonJava.nrx

   This program starts an UNZIP program, redirect 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.[5] 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.

Listing 7.2: Use of ProcessBuilder

```
/**
 * 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

  /**
   * Method run 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 = in.readLine()
      loop while line <> null
   this.outList.add(line)
   line = 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)
    this.commandList = command_
    do
```

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

```
    pb = ProcessBuilder(command_)
    pb.redirectErrorStream(1)
    this.pid = pb.start()
    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()
  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 11.1 on page 31.

Listing 7.3: Output Line Event

```
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
```

Listing 7.4: Output Event Listener

```
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:

Listing 7.5: Example of calling the OSProcess class - registering an eventhandler

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

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

Listing 7.6: Example of implementing the listener method

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

8

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. NetRexxis case independent[6]; 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.

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

9

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[7] 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.

9.1 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.

[7] Cfront

9.2 Dependent Classes

9.3 Properties

9.4 Methods

9.5 Inheritance

9.6 Overriding Methods

9.7 Overriding Properties

10

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[8], 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.

10.1 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.

10.2 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.

10.3 Some NetRexx package history

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

`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

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

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, and it is a sober fact that as a result some needlessly profane language was uttered on occasion by some in some projects that suffered the consequences of this. 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**, because some major other languages also follow this convention.

10.4 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
```

- 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).

11

Programming Patterns

Much has been made of patterns as aggregations of higher level embodiments of programming solutions. It has been observed[9] that of a number of the C++ oriented patterns in Design Patterns[10], 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.

11.1 Events

11.2 Recursive Parse

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

11.3 Observer

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 7.4 on page 23) 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 23) uses an event approach to consume output lines from a subprocess - in the version that puts the output in an

[9] This observation from a Java patterns book.

[10] Gamma, Helm, Johnson, Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional; 1994

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.

12

Incorporating Class Libraries

12.1 The Collection Classes

13

Input and Output

13.1 The File Class

13.2 Streams

13.3 Line mode I/O

13.3.1 Line mode I/O using BufferedReader and PrintWriter

13.3.2 Line mode I/O using BufferedReader and FileOutputStream

Listing 13.1: Buffered I?O

```
/* 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'
```

13.4 Byte Oriented I/O

13.5 Data Oriented I/O

13.6 Object Oriented I/O using Serialization

13.7 The NIO Approach

14

Algorithms in NetRexx

14.1 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.

Listing 14.1: Factorial

```
/* NetRexx */

options replace format comments java crossref savelog 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:

Listing 14.2: Factorial Recursive

```
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.

14.2 Fibonacci

Listing 14.3: Fibonacci

```
/* NetRexx */
options replace format comments java crossref savelog 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. */
```

15 Using Parse

The `Parse` statement is one of the stalwarts of the Rexx family of languages.

16

Using Trace

17

Concurrency

17.1 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.

Listing 17.1: Thread sample 1

```
/* 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
```

Listing 17.2: Thread sample 2

```
/* 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.

18

User Interfaces

18.1 AWT

18.2 Web Applets using AWT

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 optimizing webpages which may be accessed by people using a slow internet connection. Several examples using this style are included in the NetRexx package like the two listed below.

Listing 18.1: Nervous Texxt

```
/* NervousText applet in NetRexx: Test of text animation.
   Algorithms, names, etc. are directly from the Java version by
   Daniel Wyszynski and kwalrath, 1995
*/
options binary

class NervousTexxt extends Applet implements Runnable

  separated = char[]
  s = String
  killme = Thread
  threadSuspended = boolean 0

  method init
    resize(300,50)
    setFont(Font("TimesRoman", Font.BOLD, 30))
    s = getParameter("text")
    if s = null then s = "NetRexx"

    separated = char[s.length]
    s.getChars(0, s.length, separated,0)

  method start
    if killme \= null then return
    killme = Thread(this)
    killme.start

  method stop
    killme = null

  method run
    loop while killme \= null
      Thread.sleep(100)
      this.repaint
    catch InterruptedException
    end
    killme = null

  method paint(g=Graphics)
    loop i=0 to s.length-1
      x_coord = int Math.random*10+15*i
      y_coord = int Math.random*10+36
```

```
    g.drawChars(separated, i, 1, x_coord, y_coord)
  end

 method mouseDown(evt=Event, x=int, y=int) returns boolean
  if threadSuspended then killme.resume
                     else killme.suspend
  threadSuspended = \threadSuspended
  return 1
```

Listing 18.2: ArchText

```
/* ArchText applet: multi-coloured text on a white background */
/* Mike Cowlishaw April 1996, December 1996              */
options binary

class ArchText extends Applet implements Runnable

 text  ="NetRexx"                          /* default text */
 tick  =0                                /* display counter */
 timer =Thread null                        /* timer thread */
 shadow=Image                              /* shadow image */
 draw  =Graphics                         /* where we can draw */

 method init
  s=getParameter("text")           /* get any provided text */
  if s\=null then text=s
  shadow=createImage(getSize.width, getSize.height) /* image */
  draw=shadow.getGraphics
  draw.setColor(Color.white)                /* background */
  draw.fillRect(0, 0, getSize.width, getSize.height) /* .. */
  draw.setFont(Font("TimesRoman", Font.BOLD, 30)) /* font */

 method start
  if timer=null then timer=Thread(this)     /* new thread */
  timer.setPriority(Thread.MAX_PRIORITY)  /* time matters */
  timer.start                           /* start the thread */

 method stop
  if timer=null then return              /* have no thread */
  timer.stop                              /* else stop it */
  timer=null                             /* .. and discard */

 method run                          /* this runs as thread */
  loop while timer\=null
    tick=tick+1                            /* next update */
    hue=((tick+133)//191)/191
    draw.setColor(Color.getHSBColor(hue, 1, 0.7))
    draw.drawString(text, 0, 30)
    this.repaint                          /* .. and redraw */
    Thread.sleep(119)                      /* wait awhile */
  catch InterruptedException
  end
  timer=null                                  /* discard */

 method update(g=Graphics)      /* override Applet's update */
  paint(g)                      /* method to avoid flicker */

 method paint(g=Graphics)
  g.drawImage(shadow, 0, 0, null)
```

- Full-function, where decimal arithmetic is used, and advantage is taken of the full power of the NetRexx runtime *Rexx* class.

  An example using this style is the below *WordClock.nrx*.

Listing 18.3: WordClock

```
/* WordClock -- an applet that shows the time in English. */
/*                                                        */
/*   Parameters:                                          */
/*                                                        */
/*     face -- the font face to use                       */
/*     size -- the font size to use                       */
/*                                                        */
/* ------------------------------------------------------ */
/* Based on the ancient QTIME.REXX, and typical Java applets. */
```

```
class WordClock extends Applet implements Runnable

 timer=Thread null                     /* the timer thread */
 offsetx; offsety                        /* text position */
 now                                      /* current time */

 method init
  /* Get parameters from the <applet> markup */
  face=getParameter("face")                    /* font face */
  if face=null then face="TimesRoman"
  size=getParameter("size")
  if size=null then size="20"                  /* font size */

  setFont(Font(face, Font.BOLD, size))
  resize(size*20, size*2)               /* set window size */
  offsetx=size/2              /* and where text will start */
  offsety=size*3/2                   /* note Y is from top */
  parse Date() . . . now .  /* initial time is fourth word */

 method start
  if timer=null then timer=Thread(this)     /* new thread */
  timer.setPriority(Thread.MAX_PRIORITY)  /* time matters */
  timer.start                           /* start the thread */

 method stop
  if timer\=null then do                     /* have thread */
    timer.stop                             /* .. so stop it */
    timer=null                            /* .. and discard */
    end

 method run
  /* Use the Java Date class to get the time */
  loop while timer\=null
    parse Date() . . . now .        /* time is fourth word */
    this.repaint                               /* redisplay */
    parse now ':' ':'secs               /* where in minute */
    wait=30-secs             /* calculate delay in seconds */
    if wait<=0 then wait=wait+60
    /* say 'secs, wait:' secs wait */
    Thread.sleep(1000*wait)       /* wait for milliseconds */
  catch InterruptedException
    say 'Interrupted...'
  end
  timer=null                                        /* done */

 method paint(g=Graphics)
  g.drawString(wordtime(now), offsetx, offsety) /* show it */

 /* WORDTIME -- a cut-down version of QTIME.REXX
    Arg1 is the time string (hh:mm:ss)
    Returns the time in english, as a Rexx string
    */
 method wordtime(arg) static returns Rexx
  /* Extract the hours, minutes, and seconds from the time. */
  parse arg hour':'min':'sec
  if sec>29 then min=min+1            /* round up minutes */

  /* Nearness phrases - this time using an array        */
  near=Rexx[5]                               /* five items */
  near[0]=''                                      /* exact */
  near[1]=' just gone'; near[2]=' just after'    /* after */
  near[3]=' nearly';   near[4]=' almost'        /* before */

  mod=min//5            /* where we are in 5 minute bracket */
  out="It's"near[mod]         /* start building the result */
  if min>32 then hour=hour+1      /* we are TO the hour... */
  min=min+2    /* shift minutes to straddle a 5-minute point */

  /* Now special-case the result for Noon and Midnight hours */
  if hour//12=0 & min//60<=4 then do
    if hour=12 then return out 'Noon.'
    return 'Midnight.'
    end

  min=min-(min//5)                  /* find nearest 5 mins */
  if hour>12
   then hour=hour-12           /* get rid of 24-hour clock */
   else
    if hour=0 then hour=12     /* .. and allow for midnight */
```

```
  /* Determine the phrase to use for each 5-minute segment */
  select
    when min=0 then nop            /* add "o'clock" later */
    when min=60 then min=0                       /* ditto */
    when min= 5 then out=out 'five past'
    when min=10 then out=out 'ten past'
    when min=15 then out=out 'a quarter past'
    when min=20 then out=out 'twenty past'
    when min=25 then out=out 'twenty-five past'
    when min=30 then out=out 'half past'
    when min=35 then out=out 'twenty-five to'
    when min=40 then out=out 'twenty to'
    when min=45 then out=out 'a quarter to'
    when min=50 then out=out 'ten to'
    when min=55 then out=out 'five to'
    end

  numbers='one two three four five six'-  /* continuation */
    'seven eight nine ten eleven twelve '
  out=out numbers.word(hour)       /* add the hour number */
  if min=0 then out=out "o'clock" /* .. and o'clock if exact */

  return out'.'                /* return the final result */

/* Mike Cowlishaw, December 1979 - January 1985. */
/* NetRexx version March 1996; applet April 1996. */
```

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.

18.3 Swing

Swing is the most commonly used name for the second attempt from the SUN engineers to provide a graphical user interface library for the JVM. With AWT also acknowledged by SUN to be a quick attempt that was made just before release of the first Java package, it became clear that it was rather taxing on system resources without compensation by a pretty look. A case in point is the event mechanism, that indiscriminately sends around mouse and keyboard events even when nobody is listening to them. The architecture for Swing prescribes registering for events before they are produced, and tries to have the drawing done by the Java graphics engine instead of leaning heavily on the operating system's native GUI functionality. The user interface widgets that are produced by Java are called 'light' and their looks can be changed by applying different skins, called *'look-and-feel'* (LAF) libraries.

In the first months of its existence Swing gathered quite a bad reputation because it made the Java 1.2 releases that contained it very slow in starting up programs that used the library. Consequently, much was invested in performance studies by SUN engineers and these problems were solved. One of the things that came out is that dividing the libraries in a great many classes, done for performance reasons, worked counterproductive. All these problems were solved over the years, and developments in hardware and multithreading took care of the rest, and nowadays Swing is a valid way of producing a rich client user interface.

For esthetical reasons, it is best to research a bit in the third party look-and-feel libraries that can be obtained. Swing can be made to look beautiful, but it takes some care and the defaults are not helping.

18.3.1 Creating NetRexx Swing interfaces with NetBeans

18.4 Web Frameworks

18.4.1 JSF

19

Network Programming

19.1 Using Uniform Resource Locators (URL)

19.2 TCP/IP Socket I/O

19.3 RMI: Remote Method Interface

20

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.

Listing 20.1: A JDBC Query example

```
/* 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' -
```

```
48          'FROM' prefix'.DEPARTMENT dep,' prefix'.EMPLOYEE emp'-
49          'WHERE dep.mgrno=emp.empno ORDER BY dep.deptno'
50    stmt = Statement jdbcCon.createStatement()
51    say 'Executing query:'
52    loop i=0 to (query.length()-1)%75
53      say '   ' query.substr(i*75+1,75)
54    end
55    rs = ResultSet stmt.executeQuery(query)
56    say 'Results:'
57    loop row=0 while rs.next()
58      say rs.getString('deptno') rs.getString('deptname') -
59           'is directed by' rs.getString('lastname') rs.getString('firstnme')
60    end
61    rs.close()                       -- close the ResultSet
62    stmt.close()                     -- close the Statement
63    jdbcCon.close()                  -- close the Connection
64    say 'Retrieved' row 'departments.'
65  catch e3 = SQLException
66    say 'SQLException(s) caught !'
67    loop while (e3 \= null)
68      say 'SQLState:' e3.getSQLState()
69      say 'Message:  ' e3.getMessage()
70      say 'Vendor:  ' e3.getErrorCode()
71      say
72      e3 = e3.getNextException()
73    end
74  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 line 32 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**.

Listing 20.2: A JDBC Update example

```
/* 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...'
```

```
23   Class.forName('COM.ibm.db2.jdbc.app.DB2Driver').newInstance()
24   -- Class.forName('COM.ibm.db2.jdbc.net.DB2Driver').newInstance()
25 catch e1 = Exception
26   say 'The DB2 driver classes could not be found and loaded !'
27   say 'Exception (' e1 ') caught : \n' e1.getMessage()
28   exit 1
29 end                                  -- end : loading DB2 support
30
31 do                                   -- connecting to DB2 host
32   say 'Connecting to:' url
33   jdbcCon = Connection DriverManager.getConnection(url, 'userid', 'password')
34 catch e2 = SQLException
35   say 'SQLException(s) caught while connecting !'
36   loop while (e2 \= null)
37      say 'SQLState:' e2.getSQLState()
38      say 'Message: ' e2.getMessage()
39      say 'Vendor: ' e2.getErrorCode()
40      say
41      e2 = e2.getNextException()
42    end
43    exit 1
44 end                                  -- end : connecting to DB2 host
45
46 do                                   -- retrieve employee, update firstname
47
48   say 'Preparing update...'                       -- prepare UPDATE
49   updateQ = 'UPDATE' prefix'.EMPLOYEE SET firstnme = ? WHERE empno = ?'
50   updateStmt = PreparedStatement jdbcCon.prepareStatement(updateQ)
51   say 'Creating query...'                         -- create SELECT
52   query = 'SELECT firstnme, lastname, empno FROM' prefix'.EMPLOYEE'
53   stmt = Statement jdbcCon.createStatement()
54   rs = ResultSet stmt.executeQuery(query)         -- execute select
55
56   loop row=0 while rs.next()                      -- loop employees
57     firstname = String rs.getString('firstnme')
58     if lowup = 'U' then firstname = firstname.toUpperCase()
59     else do
60        dChar = firstname.charAt(0)
61        firstname = dChar || firstname.substring(1).toLowerCase()
62     end
63     updateStmt.setString(1, firstname)            -- parms for update
64     updateStmt.setString(2, rs.getString('empno'))
65     say 'Updating' rs.getString('lastname') firstname ': \0'
66     say updateStmt.executeUpdate() 'row(s) updated' -- execute update
67   end
68
69   rs.close()                         -- close the ResultSet
70   stmt.close()                       -- close the Statement
71   updateStmt.close()                 -- close the PreparedStatement
72   jdbcCon.close()                    -- close the Connection
73   say 'Updated' row 'employees.'
74 catch e3 = SQLException
75   say 'SQLException(s) caught !'
76   loop while (e3 \= null)
77      say 'SQLState:' e3.getSQLState()
78      say 'Message: ' e3.getMessage()
79      say 'Vendor: ' e3.getErrorCode()
80      say
81      e3 = e3.getNextException()
82    end
83 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.

21

WebSphere MQ

WebSphere MQ (also and maybe better known as MQ Series) is IBM's messaging and queing middleware, 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.

Listing 21.1: MQ Sample

```
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!"
```

```
50   catch ex=MQException
51     say "A WebSphere MQ Error occured : Completion Code " ex.completionCode "Reason Code " ex.reasonCode
52   catch ex2=java.io.IOException
53     say "An IOException occured whilst writing to the message buffer: " ex2
54   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 exceptionhandler 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.

Listing 21.2: MQ Message Reader

```
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"
```

```
46      queue.close()
47      say "Disconnecting from the Queue Manager"
48      qMgr.disconnect()
49      say "Done!"
50    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.

Listing 21.3: MQ Java Channel Exit

```
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
```

Listing 21.4: WatchdogTimer

```
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.

Listing 21.5: Publish/Subscribe

```
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.

22

MQTT

22.1 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.

Listing 22.1: MQTT Publish Sample

```
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=MqttTopic,m=MqttMessage)
    log("Message Arrived: " t m)

  method deliveryComplete(t=MqttDeliveryToken)
    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.

Listing 22.2: MQTT Subscribe Sample

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

class Subscribe implements MqttCallback

  properties
  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,MqttPersistenceException
    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=MqttTopic,m=MqttMessage)
    log("Message Arrived: " t m)

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

  method connectionLost(t=Throwable)
    do
    connectAndSubscribe()
    catch mqx=MqttException
      log(mqx.getMessage())
    end

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

In the home setup, there is a Raspberry PI running the client while a server in the attic runs the Mosquitto broker. On the Raspberry, which runs Debian wheezy with the soft-float ABI that, at the moment of writin, is still necessary for the Oracle ARM Java implementation; everything done in NetRexx runs unchanged; I just move the classes to it using scp. The broker on the laptop takes care of the scenario in which I suddenly can do some development while not connected to the net, like when I have some moments to reflect on the code in the IKEA restaurant while my spouse runs the serious shopping business.

Security is outside of the scope of this introduction which shows you the sourcecode 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.

23

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.

24

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:

Listing 24.1: Try the NetRexxA interface

```
options binary
import org.netrexx.process.NetRexxA

interpreter=NetRexxA()         -- 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.

24.1 The NetRexxA constructor

Listing 24.2: 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.

24.2 The parse method

Listing 24.3: parse

```
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*). [11] The parse method prefixes the *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.

24.3 The getClassObject method

Listing 24.4: getClassObject

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

```
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.

24.4 The exiting method

Syntax:

Listing 24.5: exiting

```
exiting() returns boolean
```

If this method returns true, an interpreted program has invoked the NetRexx “exit” instruction to shut down the interpreter. If more programs need to be interpreted, a new instance of the interpreter will need to be created with the NetRexxA() constructor.

24.5 Interpreting programs contained in memory strings

RexxLA’s NetRexx 3.01 release adds some extensions to the NetRexxA API to support interpreting programs from memory strings. The first extension adds an optional array of strings containing programs to the standard parse API. It is mainly useful to IDE developers. It also serves as the basis to support two other extensions documented below.

Listing 24.6: parse program in memory buffer

```
method parse(filestrings=String[], programstrings=String[], flagstrings=String[], logfile=PrintWriter
    null, outfile=PrintStream System.out) returns boolean
```

Parses a set of files, under specified flags:

- filestrings is a list of program names,
- programstrings is a list of program strings[12],
- flagstrings is a list of flags,
- logfile is a PrintWriter for parse output messages (optional),
- outfile is a PrintStream for console output messages (optional),

This method returns 1 if no error

The second extension is a new easy to use method to parse and interpret a program contained in a string.

[12] Note that program strings which are not named in the name list are ignored.

Listing 24.7: parse program in string

```
method interpret(programname=String, programstring=String, argstring=String "", flagstring=String "",
    logfile=PrintWriter null, outfile=PrintStream System.out) returns boolean
```

A convenience method to interpret a single NetRexx program in a string:

- programname is the program name,
- programstring is the program string,
- argstring is the argument string (optional),
- flagstring is the translator flags string (optional),
- logfile is a PrintWriter for parse output messages (optional),
- outfile is a PrintStream for console output messages (optional),

This method returns 1 if no parse error. The default flag is -verbose0

Here is a simple example using the interpret method:

Listing 24.8: interpret from string

```
import org.netrexx.process.
netrexxapi=NetRexxA()
myprog="say 'argument string is' arg"
netrexxapi.interpret("myprog",myprog,"a passed argument")
```

The third extension is a slightly more complex "eval" method that allows a program to call a static method in a program string and receive an object back.

Listing 24.9: eval

```
method eval(programname=String, programstring=String, methodname=String, argstring=String "",
    flagstring=String "", logfile=PrintWriter null, outfile=PrintStream System.out) returns Object
```

A convenience method to interpret a method from a NetRexx program in a string and return an object:

- programname is the program name,
- programstring is the program string,
- methodname is the method name to call - the method must accept a String array like main methods,
- argstring is the argument string (optional),
- flagstring is the translator flags string (optional),
- logfile is a PrintWriter for parse output messages (optional),
- outfile is a PrintStream for console output messages (optional),

This method returns an object if no error. The default flag is -verbose0

Here is a simple example using the eval method:

Listing 24.10: eval example

```
import org.netrexx.process.
netrexxapi=NetRexxA()
termpgmstring='method term(sa=String[]) static returns rexx;i=Rexx(sa);return 1/i'
say netrexxapi.eval("termpgm",termpgmstring,"term",99)
```

25

Interfacing to Open Object Rexx

25.1 BSF4ooRexx

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/`). Chapter 27 on page 79 discusses the setup of Eclipse to build the translator itself; and has instructions for the setup of the NetRexx plugin.

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 MacOSX

Aquamacs	A version of Emacs that is integrated with the MacOSX 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 MacOSX. `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 **`http://kenai.org/NetRexx/contrib/java2nrx`**

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

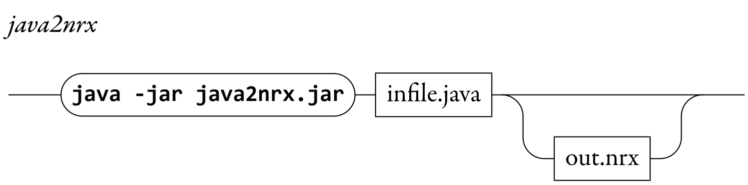
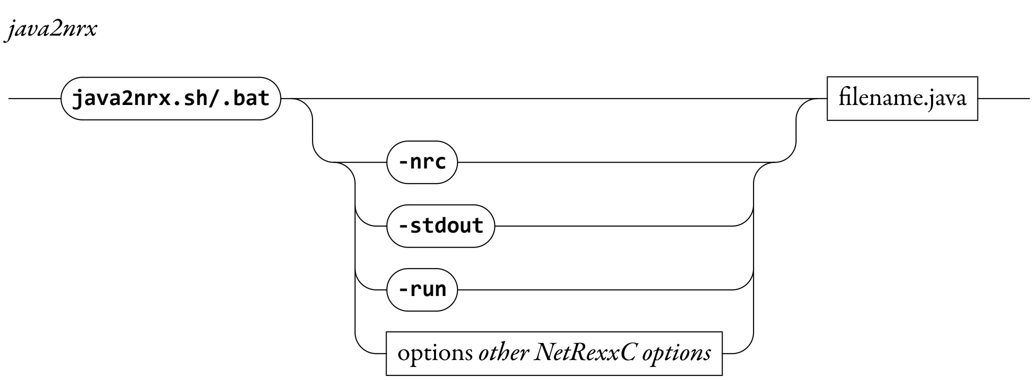


FIGURE 3: Java2nrx 2



-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[13].

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 namedECLIPSE 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.

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.

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

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 SVN

This version of Eclipse comes with CVS and Git support built in, but the SVN support must be installed.

1. Click on Main Menu -> Help -> Eclipse MarketPlace.
2. Type SVN in the search box and hit Enter.
3. Locate Subversive - it will probably be the first entry - and click the Install button.
4. Click Next, I Accept the License and Finish. The SVN plugin will be downloaded.
5. Click Yes to restart Eclipse.
6. The SVN "Install connectors" dialog will start.
7. Select the SVN Kit 1.75.
8. Click Next, Accept the License, Finish, OK to unsigned content, and Yes to restart Eclipse.

27.5 Downloading the NetRexx project from the SVN repository

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

1. The NetRexx SVN repository name is: **`https://svn.kenai.com/svn/netrexx~netrexxc-repo`**
2. Copy it (for pasting) from above, or get it from the kenai or netrexx.org site.
3. You do not need a period at the end.
4. Click on Main Menu -> File -> New -> Other -> SVN -> Project from SVN, then Next or double click.
5. Select Create a New Repository location, click Next
6. Paste (or type if you must) the repository name into the URL field and click Next
7. The Checkout from SVN - Select Resource dialog will come up. Click Browse
8. Double click on "netrexxc", and then single click on "trunk" to select it. Click OK
9. Now click Finish in the checkout dialog to bring up the "Checkout As" dialog
10. Leave the selection at the default of "Checkout ... using the New ProjectWizard", and Finish
11. The New Project dialog comes up - double click on Java and then Java Project (or use Next)
12. The New Java Project dialog comes up. Enter a project name, perhaps something like NetRexx301.
13. Click Finish, and the project is downloaded. It will show up in the Package Explorer on the left.

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 and 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, unfethered 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

Nonewithstanding the current policy of Apple to only allow Objective-C as a programming language on the iPhone and iPad, NetRexx on IOS works fine. This is what one should do to make it work:

1. Jailbreak[14] 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

[14] Note that jailbreaking an iPhone is against your eula (well - Apple's eula) and might be illegal in some jurisdictions.

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.

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 $*
```

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.

29

Translator inner workings

29.1 Method resolution

Until version 3.01 of the NetRexx translator a slightly different way of method resolution was used. The chances that this will ever impact your program are very small, but for the sake of history preservation (and to clarify the process that is used) the way in which the translator looks up and decides to find methods in the inheritance tree are documented here.

Index

ISBN 978-90-819090-0-6
9 789081 909006 >