

NetRexx Programming Guide

RexxLA

Version 3.01 of April 15, 2012

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

Publication Data

©Copyright The REXX Language Association, 2012

All original material in this publication is published under the Creative Commons - Share Alike 3.0 License as stated at <http://creativecommons.org/licenses/by-nc-sa/3.0/us/legalcode>.

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	i
Typographical conventions	iii
Introduction	v
1 Meet the Rexx Family	1
1.1 Once upon a Virtual Machine	1
1.2 Once upon another Virtual Machine	1
1.3 Features of NetRexx	2
2 Learning to program	3
2.1 Console Based Programs	3
2.2 Comments in programs	5
2.3 Strings	5
2.4 Clauses	6
2.5 When does a Clause End?	6
2.6 Special Variables	6
3 NetRexx as a Scripting Language	9
4 NetRexx as an Interpreted Language	11
5 NetRexx as a Compiled Language	13
6 Calling non-JVM programs	15
7 Using NetRexx classes from Java	17
8 Classes	19
8.1 Classes	19
8.2 Properties	19
8.3 Methods	19
8.4 Inheritance	19
8.5 Overriding Methods	19
8.6 Overriding Properties	19

9	Using Packages	21
9.1	The package statement	21
9.2	Translator performance consequences	21
9.3	Some NetRexx package history	21
9.4	CLASSPATH	22
10	Incorporating Class Libraries	23
10.1	The Collection Classes	23
11	Input and Output	25
11.1	The File Class	25
11.2	Streams	25
11.3	Line mode I/O	25
11.4	Byte Oriented I/O	25
11.5	Data Oriented I/O	25
11.6	Object Oriented I/O using Serialization	25
11.7	The NIO Approach	25
12	Algorithms in NetRexx	27
12.1	Factorial	27
12.2	Fibonacci	28
13	Using Parse	31
14	Using Trace	33
15	User Interfaces	35
15.1	AWT	35
15.2	Web Applets using AWT	35
15.3	Swing	39
15.4	Web Frameworks	39
16	Network Programming	41
16.1	Using Uniform Resource Locators (URL)	41
16.2	TCP/IP Socket I/O	41
16.3	RMI: Remote Method Interface	41
17	Database Connectivity with JDBC	43
18	Threads	47
19	Component Based Programming: Beans	49
20	Using the NetRexxA API	51
20.1	The NetRexxA constructor	52
20.2	The parse method	52
20.3	The getClassObject method	53
21	Interfacing to Open Object Rexx	55
21.1	BSF4ooRexx	55

22	Editor support	57
22.1	JVM - All Platforms	57
22.2	Linux	57
22.3	MS Windows	57
22.4	MacOSX	58

List of Figures	59
------------------------	-----------

List of Tables	59
-----------------------	-----------

Index	65
--------------	-----------

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 Beginnings Guide	This guide is meant for an audience that has done some programming and wants a quick start. It starts with a quick tour of the language, and a section on installing the NetRexx translator and how to run the reference implementation. 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:

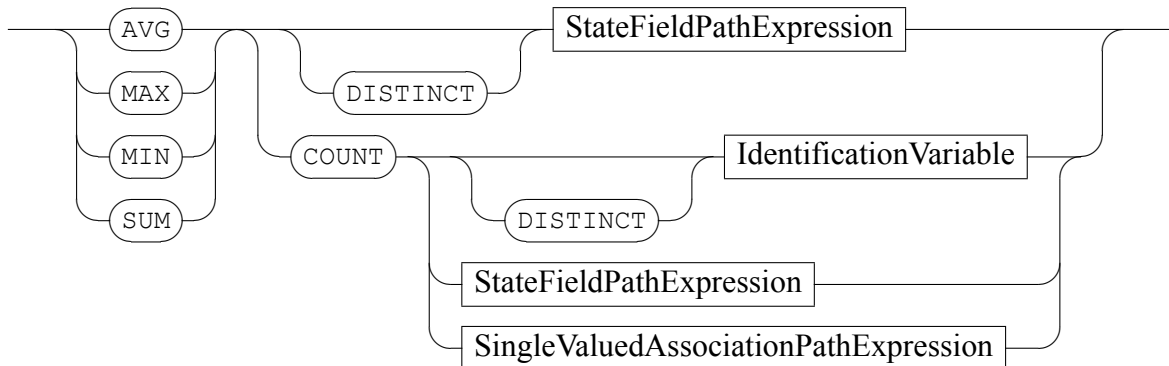
- Body text is in this font
- Examples of language statements are in a **bold** type
- Variables or strings as mentioned in source code, or things that appear on the console, are in a `typewriter` type
- Items that are introduced, or emphasized, are in an *italic* type
- Included program fragments are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

Listing 1: Example Listing

```
1 — salute the reader
2 say 'lectorem salutat'
```

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

AggregateExpression



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 technologies 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.

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 (because unacknowledged in the original publications) 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 than one hundred NetRexx programming examples. The Redbook authors (Peter Heuchert, Frederik Haesbrouck, Norio Furukawa, Ueli Wahli) have provided an important document that has shown, in an early stage, how almost everything on the JVM is easier done in NetRexx. If anyone feels their copyright is violated, please do let us know, so we can take out offending passages or modify 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.

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. VMTM (now called z/VM) is the original Virtual Machine operating system, stemming from an era in which time sharing was acknowledged to be the wave of the future and when systems as CTSS (on the IBM 704) and TSS (on the IBM 360 Family of computers) were early timesharing systems, that offered the user an illusion of having a large machine for their exclusive use, but fell short of virtualising the entire hardware. The CP/CMS system changed this; CP virtualised the hardware completely and CMS was the OS running on CP. CMS knew a succession of command interpreters, called EXEC, EXEC2 and RexxTM (originally REX - until IBM Legal interfered) - the EXEC roots are the explanation why some people refer to an 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 JavaTM to OS/2TM and soon after started with an experiment to run Rexx on the JVMTM. With Rexx generally considered the first of the general purpose scripting languages, NetRexxTM 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 Cowlshaw 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, NetRexx instructions are common words, such as **say**, **ask**, **if...then...else**, **do...end**, and **exit**.

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

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

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

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

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

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

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

```
1 /* A conversation */
2 say "Hello! What's your name?"
3 who=ask
4 if who = '' then say "Hello stranger"
5 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 22, *Editor Support*, on page 57.

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 Quick Beginning Guide*, we can run it with

```
nrc -exec hello
```

and this will yield the result:

```
\nr portable processor, version \nr 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

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

2.6 Special Variables

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

Listing 2.3: NetRexx Special Variables

```
1 /* NetRexx */
2 options replace format comments java crossref savelog symbols binary
3
4 class RCSpecialVariables
5
6 method RCSpecialVariables()
7   x = super.toString
8   y = this.toString
9   say '<super>'x'</super>'
10  say '<this>'y'</this>'
11  say '<class>'RCSpecialVariables.class'</class>'
12  say '<digits>'digits'</digits>'
```

```

13 say '<form>'form'</form>'
14 say '<[1, 2, 3].length>'
15 say [1, 2, 3].length
16 say '</[1, 2, 3].length>'
17 say '<null>'
18 say null
19 say '</null>'
20 say '<source>'source'</source>'
21 say '<sourceline>'sourceline'</sourceline>'
22 say '<trace>'trace'</trace>'
23 say '<version>'version'</version>'
24
25 say 'Type an answer:'
26 say '<ask>'ask'</ask>'
27
28 return
29
30 method main(args = String[]) public static
31
32   RCSpecialVariables()
33
34   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 19, as is the case with the **class** variable.

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

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

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

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

version The **version** variable returns the version of the NetRexx translator that was in use at the time the clause we processed; in case of interpreted execution (see chapter 4 on 11, 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>\nr 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.

NetRexx as a Scripting Language

You can use NetRexx as a simple scripting language without having knowledge of, using any of the features that is needed in a Java program that runs on the JVM.

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.

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.

NetRexx as an Interpreted Language

NetRexx as a Compiled Language

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 6.1: Calling Non-JVM Programs

```

1  /* script\NonJava.nrx
2
3      This program starts an UNZIP program, redirect its output,
4      parses the output and shows the files stored in the zipfile */
5
6  parse arg unzip zipfile .
7
8  — check the arguments — show usage comments
9  if zipfile = '' then do
10     say 'Usage: Process unzipcommand zipfile'
11     exit 2
12 end
13
14 do
15     say "Files stored in" zipfile
16     say "____".left(39,"_") "____".left(39,"_")
17     child = Runtime.getRuntime().exec(unzip ' -v' zipfile) — program start
18
19     — read input from child process
20     in = BufferedReader(InputStreamReader(child.getInputStream()))
21     line = in.readLine
22
23     start = 0 — listing of files are not available yet
24     count = 0
25     loop while line \= null
26         parse line sep program
27         if sep = '____' then start = \start
28         else
29             if start then do
30                 count = count + 1
31                 if count // 2 > 0 then say program.word(program.words).left(39) '\-'
32                 else say program.word(program.words)
33             end
34             line = in.readLine()
35         end
36
37     — wait for exit of child process and check return code
38     child.waitFor()
39     if child.exitValue() \= 0 then
40         say 'UNZIP return code' child.exitValue()
41
42     catch IOException
43         say 'Sorry cannot find' unzip
44     catch e2=InterruptedException
45         e2.printStackTrace()
46 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.¹ 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.

¹This is akin to what one would do with *queue* on z/VM CMS and *outtrap* on z/OS TSO in Classic Rexx.

Using NetRexx classes from Java

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

A NetRexx method without a `returns` keyword can return nothing, which is the void type in Java, or a Rexx string. NetRexx is case independent²; 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.

²With the default of `options nostrictcase` in effect.

Classes

8.1 Classes

8.2 Properties

8.3 Methods

8.4 Inheritance

8.5 Overriding Methods

8.6 Overriding Properties

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 names³, 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 probably 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.

9.1 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.

9.2 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.

9.3 Some NetRexx package history

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

`COM.ibm.netrexx.process`

³Willard Van Orman Quine, *Word and Object*, MIT Press, 1960, ISBN 0-262-67001-1

The official, SUN ordained convention for package names was, to prepend the reversed domain name of the vendor to the package name, while uppercasing the top level domain. NetRexx, being one of the first programs to make use of packages, followed this convention, that was quickly dropped by SUN afterwards, probably because someone experienced what trouble it could cause with version management software that adapted to case-*sensitive* and case-*insensitive* file systems. For NetRexx, which had started out keenly observing the rules, this insight came late, 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.

9.4 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).

Incorporating Class Libraries

10.1 The Collection Classes

Input and Output

11.1 The File Class

11.2 Streams

11.3 Line mode I/O

11.3.1 Line mode I/O using BufferedReader and PrintWriter

11.3.2 Line mode I/O using BufferedReader and BufferedWriter

11.4 Byte Oriented I/O

11.5 Data Oriented I/O

11.6 Object Oriented I/O using Serialization

11.7 The NIO Approach

Algorithms in NetRexx

12.1 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 12.1: Factorial

```

1  /* NetRexx */
2
3  options replace format comments java crossref savelog symbols nobinary
4
5  numeric digits 64 — switch to exponential format when numbers become larger than 64
   digits
6
7  say 'Input a number: \-'
8  say
9  do
10   n_ = long ask — Gets the number, must be an integer
11
12   say n_ '! =' factorial(n_) '(using iteration)'
13   say n_ '! =' factorial(n_, 'r') '(using recursion)'
14
15   catch ex = Exception
16     ex.printStackTrace
17 end
18
19 return
20
21 method factorial(n_ = long, fmethod = 'I') public static returns Rexx signals
   IllegalArgumentException
22
23 if n_ < 0 then —
24   signal IllegalArgumentException('Sorry, but' n_ 'is not a positive integer')
25
26 select
27   when fmethod.upper = 'R' then —
28     fact = factorialRecursive(n_)
29   otherwise —
30     fact = factorialIterative(n_)
31 end
32
33 return fact
34
35 method factorialIterative(n_ = long) private static returns Rexx
36
37 fact = 1
38 loop i_ = 1 to n_
39   fact = fact * i_
40 end i_
41
42 return fact
43
44 method factorialRecursive(n_ = long) private static returns Rexx

```

```

45
46   if n_ > 1 then -
47       fact = n_ * factorialRecursive(n_ - 1)
48   else -
49       fact = 1
50
51   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 12.2: Factorial Recursive

```

1 class Factorial
2   numeric digits 64
3
4   method main(args=String[]) static
5       say factorial_(42)
6
7   method factorial_(number) static
8       if number = 0 then return 1
9       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.

12.2 Fibonacci

Listing 12.3: Fibonacci

```

1  /* NetRexx */
2  options replace format comments java crossref savelog symbols
3
4  numeric digits 210000                                /*prepare for some big ones.    */
5  parse arg x y .                                       /*allow a single number or range.*/
6  if x == '' then do                                    /*no input? Then assume -30-->+30*/
7      x = -30
8      y = -x
9  end
10
11 if y == '' then y = x                                  /*if only one number, show fib(n)*/
12 loop k = x to y                                       /*process each Fibonacci request.*/
13     q = fib(k)

```

```

14 w = q.length                                     /*if wider than 25 bytes, tell it*/
15 say 'Fibonacci' k="q
16 if w > 25 then say 'Fibonacci' k "has a length of" w
17 end k
18 exit
19
20 /*-----FIB subroutine (non-recursive)-----*/
21 method fib(arg) private static
22   parse arg n
23   na = n.abs
24
25   if na < 2 then return na                       /*handle special cases.      */
26   a = 0
27   b = 1
28
29   loop j = 2 to na
30     s = a + b
31     a = b
32     b = s
33   end j
34
35   if n > 0 | na // 2 == 1 then return s /*if positive or odd negative... */
36   else return -s /*return a negative Fib number. */

```

13

Using Parse

Using Trace

User Interfaces

15.1 AWT

15.2 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 15.1: Nervous Texxt

```

1  /* NervousText applet in NetRexx: Test of text animation.
2     Algorithms, names, etc. are directly from the Java version by
3     Daniel Wyszynski and kwalrath, 1995
4  */
5  options binary
6
7  class NervousTexxt extends Applet implements Runnable
8
9      separated = char[]
10     s = String
11     killme = Thread
12     threadSuspended = boolean 0
13
14     method init
15         resize(300,50)
16         setFont(Font("TimesRoman", Font.BOLD, 30))
17         s = getParameter("text")
18         if s = null then s = "NetRexx"
19
20         separated = char[s.length]
21         s.getChars(0, s.length, separated,0)
22
23     method start
24         if killme \= null then return
25         killme = Thread(this)
26         killme.start
27
28     method stop
29         killme = null
30
31     method run
32         loop while killme \= null
33             Thread.sleep(100)
34             this.repaint
35         catch InterruptedException
36         end
37         killme = null
38
39     method paint(g=Graphics)

```

```

40     loop i=0 to s.length-1
41         x_coord = int Math.random*10+15*i
42         y_coord = int Math.random*10+36
43         g.drawChars(separated, i, 1, x_coord, y_coord)
44     end
45
46     method mouseDown(evt=Event, x=int, y=int) returns boolean
47         if threadSuspended then killme.resume
48             else killme.suspend
49     threadSuspended = \threadSuspended
50     return 1

```

Listing 15.2: ArchText

```

1  /* ArchText applet: multi-coloured text on a white background */
2  /* Mike Cowlishaw April 1996, December 1996 */
3  options binary
4
5  class ArchText extends Applet implements Runnable
6
7  text = "NetRexx" /* default text */
8  tick = 0 /* display counter */
9  timer = Thread null /* timer thread */
10 shadow=Image /* shadow image */
11 draw =Graphics /* where we can draw */
12
13 method init
14     s=getParameter("text") /* get any provided text */
15     if s\=null then text=s
16     shadow=createImage(getSize.width, getSize.height) /* image */
17     draw=shadow.getGraphics
18     draw.setColor(Color.white) /* background */
19     draw.fillRect(0, 0, getSize.width, getSize.height) /* .. */
20     draw.setFont(Font("TimesRoman", Font.BOLD, 30)) /* font */
21
22 method start
23     if timer=null then timer=Thread(this) /* new thread */
24     timer.setPriority(Thread.MAX_PRIORITY) /* time matters */
25     timer.start /* start the thread */
26
27 method stop
28     if timer=null then return /* have no thread */
29     timer.stop /* else stop it */
30     timer=null /* .. and discard */
31
32 method run /* this runs as thread */
33     loop while timer\=null
34         tick=tick+1 /* next update */
35         hue=((tick+133)//191)/191
36         draw.setColor(Color.getHSBColor(hue, 1, 0.7))
37         draw.drawString(text, 0, 30)
38         this.repaint /* .. and redraw */
39         Thread.sleep(119) /* wait awhile */
40     catch InterruptedException
41     end
42     timer=null /* discard */
43
44 method update(g=Graphics) /* override Applet's update */
45     paint(g) /* method to avoid flicker */
46
47 method paint(g=Graphics)
48     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 15.3: WordClock

```

1  /* WordClock — an applet that shows the time in English. */

```

```

2  /*                                                                 */
3  /*   Parameters:                                                  */
4  /*                                                                 */
5  /*     face — the font face to use                                */
6  /*     size — the font size to use                                */
7  /*                                                                 */
8  /* -----                                                        */
9  /* Based on the ancient QTIME.REXX, and typical Java applets. */
10
11 class WordClock extends Applet implements Runnable
12
13 timer=Thread null          /* the timer thread */
14 offsetx; offsety          /* text position */
15 now                        /* current time */
16
17 method init
18   /* Get parameters from the <applet> markup */
19   face=getParameter("face")      /* font face */
20   if face=null then face="TimesRoman"
21   size=getParameter("size")
22   if size=null then size="20"     /* font size */
23
24   setFont(Font(face, Font.BOLD, size))
25   resize(size*20, size*2)        /* set window size */
26   offsetx=size/2                 /* and where text will start */
27   offsety=size*3/2               /* note Y is from top */
28   parse Date() . . . now .      /* initial time is fourth word */
29
30 method start
31   if timer=null then timer=Thread(this)      /* new thread */
32   timer.setPriority(Thread.MAX_PRIORITY)      /* time matters */
33   timer.start                                /* start the thread */
34
35 method stop
36   if timer\=null then do                    /* have thread */
37     timer.stop                               /* .. so stop it */
38     timer=null                              /* .. and discard */
39   end
40
41 method run
42   /* Use the Java Date class to get the time */
43   loop while timer\=null
44     parse Date() . . . now .                /* time is fourth word */
45     this.repaint                             /* redisplay */
46     parse now ':' ':'secs                    /* where in minute */
47     wait=30-secs                             /* calculate delay in seconds */
48     if wait<=0 then wait=wait+60
49     /* say 'secs, wait:' secs wait */
50     Thread.sleep(1000*wait)                  /* wait for milliseconds */
51   catch InterruptedException
52     say 'Interrupted...'
53   end
54   timer=null                                /* done */
55
56 method paint(g=Graphics)
57   g.drawString(wordtime(now), offsetx, offsety) /* show it */
58
59 /* WORDTIME — a cut-down version of QTIME.REXX
60   Arg1 is the time string (hh:mm:ss)
61   Returns the time in english, as a REXX string
62   */
63 method wordtime(arg) static returns REXX
64   /* Extract the hours, minutes, and seconds from the time. */
65   parse arg hour ':' 'min' ':' 'sec
66   if sec>29 then min=min+1                    /* round up minutes */
67
68   /* Nearness phrases — this time using an array */
69   near=REXX[5]                                /* five items */
70   near[0]=''                                  /* exact */
71   near[1]=' just gone'; near[2]=' just after' /* after */
72   near[3]=' nearly'; near[4]=' almost'        /* before */
73
74   mod=min//5                                /* where we are in 5 minute bracket */
75   out="It's"near[mod]                        /* start building the result */

```

```

76  if min>32 then hour=hour+1          /* we are TO the hour... */
77  min=min+2      /* shift minutes to straddle a 5-minute point */
78
79  /* Now special-case the result for Noon and Midnight hours */
80  if hour//12=0 & min//60<=4 then do
81      if hour=12 then return out 'Noon.'
82      return 'Midnight.'
83  end
84
85  min=min-(min//5)          /* find nearest 5 mins */
86  if hour>12
87  then hour=hour-12        /* get rid of 24-hour clock */
88  else
89      if hour=0 then hour=12    /* .. and allow for midnight */
90
91  /* Determine the phrase to use for each 5-minute segment */
92  select
93      when min=0 then nop          /* add "o'clock" later */
94      when min=60 then min=0      /* ditto */
95      when min= 5 then out=out 'five past'
96      when min=10 then out=out 'ten past'
97      when min=15 then out=out 'a quarter past'
98      when min=20 then out=out 'twenty past'
99      when min=25 then out=out 'twenty-five past'
100     when min=30 then out=out 'half past'
101     when min=35 then out=out 'twenty-five to'
102     when min=40 then out=out 'twenty to'
103     when min=45 then out=out 'a quarter to'
104     when min=50 then out=out 'ten to'
105     when min=55 then out=out 'five to'
106  end
107
108  numbers='one two three four five six'— /* continuation */
109  'seven eight nine ten eleven twelve '
110  out=out numbers.word(hour)          /* add the hour number */
111  if min=0 then out=out "o'clock" /* .. and o'clock if exact */
112
113  return out'.'          /* return the final result */
114
115  /* Mike Cowlishaw, December 1979 — January 1985. */
116  /* 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.

15.3 Swing

15.3.1 Creating NetRexx Swing interfaces with NetBeans

15.4 Web Frameworks

15.4.1 JSF

Network Programming

16.1 Using Uniform Resource Locators (URL)

16.2 TCP/IP Socket I/O

16.3 RMI: Remote Method Interface

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 17.1: A JDBC Query example

```

1  /* jdbc\JdbcQry.nrx
2
3  This NetRexx program demonstrate DB2 query using the JDBC API.
4  Usage: Java JdbcQry [<DB-URL>] [<userprefix>] */
5
6  import java.sql.
7
8  parse arg url prefix          — process arguments
9  if url = '' then
10     url = 'jdbc:db2:sample'
11 else do                      — check for correct URL
12     parse url p1 ':' p2 ':' rest
13     if p1 \= 'jdbc' | p2 \= 'db2' | rest = '' then do
14         say 'Usage: java JdbcQry [<DB-URL>] [<userprefix>]'
15         exit 8
16     end
17 end
18 if prefix = '' then prefix = 'userid'
19
20 do                            — loading DB2 support
21     say 'Loading DB2 driver classes...'
22     Class.forName('COM.ibm.db2.jdbc.app.DB2Driver').newInstance()
23     — Class.forName('COM.ibm.db2.jdbc.net.DB2Driver').newInstance()
24 catch e1 = Exception
25     say 'The DB2 driver classes could not be found and loaded !'
26     say 'Exception ( ' e1 ' ) caught : \n' e1.getMessage()
27     exit 1
28 end                            — end : loading DB2 support
29
30 do                            — connecting to DB2 host
31     say 'Connecting to:' url
32     jdbcCon = Connection DriverManager.getConnection(url, 'userid', 'password')
33 catch e2 = SQLException
34     say 'SQLException(s) caught while connecting !'
35     loop while (e2 \= null)
36         say 'SQLState:' e2.getSQLState()
37         say 'Message: ' e2.getMessage()
38         say 'Vendor: ' e2.getErrorCode()
39         say
40         e2 = e2.getNextException()
41     end
42     exit 1

```

```

43 end                                     — end : connecting to DB2 host
44
45 do                                     — get list of departments with the managers
46 say 'Creating query...'
47 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 long 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 17.2: A JDBC Update example

```

1  /* jdbc\JdbcUpd.nrx
2
3      This NetRexx program demonstrate DB2 update using the JDBC API.
4      Usage: Java JdbcUpd [<DB-URL>] [<userprefix>] [U] */
5
6  import java.sql.
7
8  parse arg url prefix lowup             — process arguments
9  if url = '' then
10     url = 'jdbc:db2:sample'
11 else do                                — check for correct URL
12     parse url p1 ':' p2 ':' rest
13     if p1 \= 'jdbc' | p2 \= 'db2' | rest = '' then do

```



```

14         say 'Usage: java JdbcUpd [<DB-URL>] [<userprefix>] [U]'
15         exit 8
16     end
17 end
18 if prefix = '' then prefix = 'userid'
19 if lowup \= 'U' then lowup = 'L'
20
21 do                                     — loading DB2 support
22     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: employees

```

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.

Threads

Component Based Programming: Beans

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 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 20.1: Try the NetRexxA interface

```

1 options binary
2 import org.netrexx.process.NetRexxA
3
4 interpreter=NetRexxA()           — make interpreter
5
6 files=['hello.nrx']              — a file to interpret
7 flags=['nocrossref', 'verbose0'] — flags, for example

```

```

8 interpreter.parse(files, flags) — parse the file(s), using the flags
9
10 helloClass=interpreter.getClassObject(null, 'hello') — find the hello Class
11
12 — find the 'main' method; it takes an array of Strings as its argument
13 classes=[interpreter.getClassObject('java.lang', 'String', 1)]
14 mainMethod=helloClass.getMethod('main', classes)
15
16 — now invoke it, with a null instance (it is static) and an empty String array
17 values=[Object String[0]]
18
19 loop for 10 — let's call it ten times, for fun...
20   mainMethod.invoke(null, values)
21 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.

20.1 The NetRexxA constructor

Listing 20.2: Constructor

```
1 NetRexxA()
```

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

20.2 The parse method

Listing 20.3: parse

```
1 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*).⁴ 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

⁴Note that the option words are not prefixed with a -.

result an take appropriate action; it will not be possible to interpret a program or class whose parsing failed with an error.

20.3 The getClassObject method

Listing 20.4: getClassObject

```
1 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.

Interfacing to Open Object Rexx

21.1 BSF4ooRexx

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).

22.1 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/).

22.2 Linux

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

22.3 MS Windows

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

22.4 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 <code>tools</code> directory.
Emacs	<code>netrexx-mode.el</code> (in the NetRexx package) runs on GNU Emacs for MacOSX. http://www.gnu.org/software/emacs .
Vim	Vi with extensions

List of Figures

List of Tables

Listings

Example Listing	iii
Hello Stranger	3
RAH Exec	6
NetRexx Special Variables	6
Calling Non-JVM Programs	15
Factorial	27
Factorial Recursive	28
Fibonacci	28
NetRexx Text	35
ArchText	36
WordClock	36
ADJDBC Query example	43
ADJDBC Update example	44
ADJDBC the NetRexxA interface	51
ADJDBC constructor	52
ADJDBC	52
ADJDBCClassObject	53

Index

Class, 43, 45, 52, 53
Rexx, 27, 37
SELECT, 45
arg, 15, 28, 29, 37, 43, 44
binary, 6, 35, 36, 51
catch, 15, 27, 35-37, 43-45
class, 6, 28, 35-37
digits, 6, 27, 28
do, 15, 27, 28, 37, 38, 43-45
else, 3, 15, 28, 29, 36, 38, 43-45
end, 15, 27-29, 35-38, 43-45
exit, 15, 29, 43, 45
extends, 35-37
for, 15, 43-45, 51, 52
form, 7
if, 3, 15, 27-29, 35-38, 43-45
implements, 35-37
import, 43, 44, 51
interpret, 51
loop, 15, 27-29, 35-37, 43-45, 52
method, 6, 7, 27-29, 35-37, 52
nop, 38
numeric, 27, 28
options, 6, 27, 28, 35, 36, 51
otherwise, 27
package, 53
parse, 15, 28, 29, 37, 43, 44, 52
private, 27, 29
public, 7, 27
return, 7, 15, 27-29, 35, 36, 38
returns, 27, 36, 37, 52, 53
say, iii, 3, 6, 7, 15, 27-29, 37, 43-45
select, 27, 38, 45
signal, 27
signals, 27
sourceline, 7
static, 7, 27-29, 37, 52
super, 6
then, 3, 15, 27-29, 35-38, 43-45
this, 6, 35-37
to, 27-29, 36, 43-45, 51
trace, 7
upper, 27
when, 27, 38
while, 15, 35-37, 43-45

application programming interface, for
 interpreting, 51
ArchText example, 35

binary arithmetic, used for Web applets,
 35

constructor, in NetRexxA API, 52

getClassObject method, in NetRexxA API, 53

HTTP server setup, 38

interpreting, API, 51
interpreting, using the NetRexxA API, 51
interpreting/API example, 51

NervousText example, 35
NetRexxA, API, 51
NetRexxA, class, 51
NetRexxA/constructor, 52

parse method, in NetRexxA API, 52

ref /API/application programming
 interface, 51
runtime/web server setup, 38

Web applets, writing, 35
Web server setup, 38
WordClock example, 36

applets for the Web, writing, 35

ISBN 978-90-819090-0-6

