NetRexx QuickStart Guide

Mike Cowlishaw and RexxLA

Version 3.06-GA of December 11, 2017

THE REXX LANGUAGE ASSOCIATION
NetRexx Programming Series
ISBN 978-90-819090-2-0

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 978-90-819090-2-0

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).
Pipelines 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:

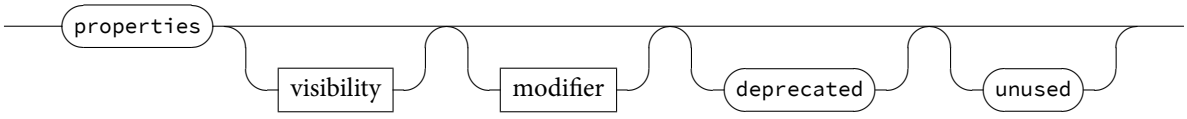
- Body text is in this font
- Examples of language statements are in a **bold** type
- Variables or strings as mentioned in source code, or things that appear on the console, are in a `typewriter` type
- Items that are introduced, or emphasized, are in an *italic* type
- Included program fragments are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

Listing 1: Example Listing

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

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

Properties



Introduction

This document is the *Quick Start Guide* for the reference implementation of NetRexx. NetRexx is a *human-oriented* programming language which makes writing and using Java[1] classes quicker and easier than writing in Java. It is part of the Rexx language family, under the governance of the Rexx Language Association.[2] NetRexx has been developed and was made available as a free download by IBM since 1995 and is free and open source since June 8, 2011.

In this Quick Start Guide, you'll find information on

1. How easy it is to write for the JVM: A Quick Tour of NetRexx
2. Installing NetRexx
3. Using the NetRexx translator as a compiler, interpreter, or syntax checker
4. Troubleshooting when things do not work as expected
5. Current restrictions.

For details of the NetRexx language, and the latest news, downloads, etc., please see the NetRexx documentation included with the package or available at: `http://www.netrexx.org`.

[1] Java is a trademark of Oracle, Inc.
[2] `http.www.rexxla.org`

1

A Quick Tour of NetRexx

This chapter summarizes the main features of NetRexx, and is intended to help you start using it quickly. It is assumed that you have some knowledge of programming in a language such as Rexx, C, BASIC, or Java, but extensive experience with programming is not needed.

This is not a complete tutorial, though – think of it more as a *taster*; it covers the main points of the language and shows some examples you can try or modify. For full details of the language, consult the NetRexx Programmer's Guide and the NetRexx Language Definition documents.

1.1 NetRexx programs

The structure of a NetRexx program is extremely simple. This sample program, "toast", is complete, documented, and executable as it stands:

Listing 1.1: Toast

```
1 /* This wishes you the best of health. */
2 say 'Cheers!'
```

This program consists of two lines: the first is an optional comment that describes the purpose of the program, and the second is a **say** instruction. **say** simply displays the result of the expression following it – in this case just a literal string (you can use either single or double quotes around strings, as you prefer). To run this program using the reference implementation of NetRexx, create a file called toast.nrx and copy or paste the two lines above into it. You can then use the NetRexxC Java program to compile it:

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

(this should create a file called toast.class), and then use the `java` command to run it:

```
java toast
```

You may also be able to use the netrexxc or nrc command to compile and run the program with a single command (details may vary – see the installation and user's guide document for your implementation of NetRexx):

```
netrexxc toast -run
```

Of course, NetRexx can do more than just display a character string. Although the language has a simple syntax, and has a small number of instruction types, it is powerful; the reference implementation of the language allows full access to the rapidly grow-

ing collection of Java programs known as class libraries, and allows new class libraries to be written in NetRexx. The rest of this overview introduces most of the features of NetRexx. Since the economy, power, and clarity of expression in NetRexx is best appreciated with use, you are urged to try using the language yourself.

1.2 Expressions and variables

Like **say** in the "toast" example, many instructions in NetRexx include expressions that will be evaluated. NetRexx provides arithmetic operators (including integer division, remainder, and power operators), several concatenation operators, comparison operators, and logical operators. These can be used in any combination within a NetRexx expression (provided, of course, that the data values are valid for those operations).

All the operators act upon strings of characters (known as *NetRexx strings*), which may be of any length (typically limited only by the amount of storage available). Quotes (either single or double) are used to indicate literal strings, and are optional if the literal string is just a number. For example, the expressions:

```
'2' + '3'
'2' + 3
 2 + 3
```

would all result in '5'.

The results of expressions are often assigned to *variables*, using a conventional assignment syntax:

Listing 1.2: Assignment

```
1  var1=5        /* sets var1 to '5' */
2  var2=(var1+2)*10 /* sets var2 to '70' */
```

You can write the names of variables (and keywords) in whatever mixture of uppercase and lowercase that you prefer; the language is not case-sensitive. This next sample program, "greet", shows expressions used in various ways:

Listing 1.3: Greet

```
1  /* A short program to greet you.        */
2  /* First display a prompt:              */
3  say 'Please type your name and then press ENTER:'
4  answer=ask         /* Get the reply into ANSWER */
5
6  /* If nothing was typed, then use a fixed greeting, */
7  /* otherwise echo the name politely.    */
8  if answer='' then say 'Hello Stranger!'
9            else say 'Hello' answer'!'
```

After displaying a prompt, the program reads a line of text from the user ("ask" is a keyword provided by NetRexx) and assigns it to the variable answer. This is then tested to see if any characters were entered, and different actions are taken accordingly; for example, if the user typed "`Fred`" in response to the prompt, then the program would display:

```
Hello Fred!
```

As you see, the expression on the last **say** (display) instruction concatenated the string "Hello" to the value of variable answer with a blank in between them (the blank is here a valid operator, meaning "concatenate with blank"). The string "!" is then directly concatenated to the result built up so far. These unobtrusive operators (the *blank operator* and abuttal) for concatenation are very natural and easy to use, and make building text strings simple and clear.

The layout of instructions is very flexible. In the "greet" example, for instance, the **if** instruction could be laid out in a number of ways, according to personal preference. Line breaks can be added at either side of the **then** (or following the **else**).

In general, instructions are ended by the end of a line. To continue a instruction to a following line, you can use a hyphen (minus sign) just as in English:

Listing 1.4: Continuation

```
say 'Here we have an expression that is quite long,' -
   'so it is split over two lines'
```

This acts as though the two lines were all on one line, with the hyphen and any blanks around it being replaced by a single blank. The net result is two strings concatenated together (with a blank in between) and then displayed. When desired, multiple instructions can be placed on one line with the aid of the semicolon separator:

Listing 1.5: Multiple Instructions

```
if answer='Yes' then do; say 'OK!'; exit; end
```

(many people find multiple instructions on one line hard to read, but sometimes it is convenient).

1.3 Control instructions

NetRexx provides a selection of *control* instructions, whose form was chosen for readability and similarity to natural languages. The control instructions include **if... then... else** (as in the "greet" example) for simple conditional processing:

Listing 1.6: Conditional

```
if ask='Yes' then say "You answered Yes"
          else say "You didn't answer Yes"
```

select... when... otherwise... end for selecting from a number of alternatives:

Listing 1.7: select - when - otherwise

```
select
  when a>0 then say 'greater than zero'
  when a<0 then say 'less than zero'
  otherwise say 'zero'
  end
select case i+1
  when 1 then say 'one'
  when 1+1 then say 'two'
  when 3, 4, 5 then say 'many'
  end
```

do... end for grouping:

Listing 1.8: do - end

```
if a>3 then do
  say 'A is greater than 3; it will be set to zero'
  a=0
end
```

and **loop... end** for repetition:

Listing 1.9: loop - end

```
/* repeat 10 times; I changes from 1 to 10 */
loop i=1 to 10
say i
end i
```

The **loop** instruction can be used to step a variable **to** some limit, **by** some increment, **for** a specified number of iterations, and **while** or **until** some condition is satisfied. **loop forever** is also provided, and **loop over** can be used to work through a collection of variables.

Loop execution may be modified by **leave** and **iterate** instructions that significantly reduce the complexity of many programs. The **select**, **do**, and **loop** constructs also have the ability to "catch" exceptions (see 1.13 on page 16.) that occur in the body of the construct. All three, too, can specify a **finally** instruction which introduces instructions which are to be executed when control leaves the construct, regardless of how the construct is ended.

1.4 NetRexx arithmetic

Character strings in NetRexx are commonly used for arithmetic (assuming, of course, that they represent numbers). The string representation of numbers can include integers, decimal notation, and exponential notation; they are all treated the same way. Here are a few:

```
'1234'
'12.03'
'-12'
'120e+7'
```

The arithmetic operations in NetRexx are designed for people rather than machines, so are decimal rather than binary, do not overflow at certain values, and follow the rules that people use for arithmetic. The operations are completely defined by the ANSI X3.274 standard for Rexx, so correct implementations always give the same results. An unusual feature of NetRexx arithmetic is the **numeric** instruction: this may be used to select the *arbitrary precision* of calculations. You may calculate to whatever precision that you wish (for financial calculations, perhaps), limited only by available memory. For example:

Listing 1.10: Digits

```
numeric digits 50
say 1/7
```

which would display

```
0.142857142857142857142857142857142857142857142857142857
```

The numeric precision can be set for an entire program, or be adjusted at will within the program. The **numeric** instruction can also be used to select the notation (*scientific* or *engineering*) used for numbers in exponential format. NetRexx also provides simple access to the native binary arithmetic of computers. Using binary arithmetic offers many opportunities for errors, but is useful when performance is paramount. You select binary arithmetic by adding the instruction:

```
options binary
```

at the top of a NetRexx program. The language processor will then use binary arithmetic (see page 14) instead of NetRexx decimal arithmetic for calculations, if it can, throughout the program.

1.5 Doing things with strings

A character string is the fundamental datatype of NetRexx, and so, as you might expect, NetRexx provides many useful routines for manipulating strings. These are based on the functions of Rexx, but use a syntax that is more like Java or other similar languages:

Listing 1.11: Strings

```
1  phrase='Now is the time for a party'
2  say phrase.word(7).pos('r')
```

The second line here can be read from left to right as:

take the variable phrase, find the seventh word, and then find the position of the first "r" in that word.

This would display "3" in this case, because "r" is the third character in "party".

(In Rexx, the second line above would have been written using nested function calls:

Listing 1.12: Rexx: Nested

```
1  say pos('r', word(phrase, 7))
```

which is not as easy to read; you have to follow the nesting and then backtrack from right to left to work out exactly what's going on.)

In the NetRexx syntax, at each point in the sequence of operations some routine is acting on the result of what has gone before. These routines are called *methods*, to make the distinction from functions (which act in isolation). NetRexx provides (as methods) most of the functions that were evolved for Rexx, including:

- `changestr` (change all occurrences of a substring to another)
- `copies` (make multiple copies of a string)
- `lastpos` (find rightmost occurrence)
- `left` and `right` (return leftmost/rightmost character(s))
- `pos` and `wordpos` (find the position of string or a word in a string)
- `reverse` (swap end-to-end)

- `space` (pad between words with fixed spacing)
- `strip` (remove leading and/or trailing white space)
- `verify` (check the contents of a string for selected characters)
- `word`, `wordindex`, `wordlength`, and `words` (work with words).

These and the others like them, and the parsing described in the next section, make it especially easy to process text with NetRexx.

1.6 Parsing strings

The previous section described some of the string-handling facilities available; NetRexx also provides string parsing, which is an easy way of breaking up strings of characters using simple pattern matching.

A **parse** instruction first specifies the string to be parsed. This can be any term, but is often taken simply from a variable. The term is followed by a *template* which describes how the string is to be split up, and where the pieces are to be put.

1.6.1 Parsing into words

The simplest form of parsing template consists of a list of variable names. The string being parsed is split up into words (sequences of characters separated by blanks), and each word from the string is assigned (copied) to the next variable in turn, from left to right. The final variable is treated specially in that it will be assigned a copy of whatever is left of the original string and may therefore contain several words. For example, in:

Listing 1.13: Parsing Strings

```
1 parse 'This is a sentence.' v1 v2 v3
```

the variable v1 would be assigned the value "This", v2 would be assigned the value "is", and v3 would be assigned the value "a sentence.".

1.6.2 Literal patterns

A literal string may be used in a template as a pattern to split up the string. For example

Listing 1.14: Parse

```
1    parse 'To be, or not to be?' w1 ',' w2 w3 w4
```

would cause the string to be scanned for the comma, and then split at that point; each section is then treated in just the same way as the whole string was in the previous example.

Thus, w1 would be set to "To be", w2 and w3 would be assigned the values "or" and "not", and w4 would be assigned the remainder: "to be?". Note that the pattern itself is not assigned to any variable. The pattern may be specified as a variable, by putting the variable name in parentheses. The following instructions:

Listing 1.15: Parse with comma

```
comma=','
parse 'To be, or not to be?' w1 (comma) w2 w3 w4
```

therefore have the same effect as the previous example.

1.6.3 Positional patterns

The third kind of parsing mechanism is the numeric positional pattern. This allows strings to be parsed using column positions.

1.7 Indexed strings

NetRexx provides indexed strings, adapted from the compound variables of Rexx. Indexed strings form a powerful "associative lookup", or *dictionary*, mechanism which can be used with a convenient and simple syntax.

NetRexx string variables can be referred to simply by name, or also by their name qualified by another string (the *index*). When an index is used, a value associated with that index is either set:

Listing 1.16: Index

```
fred=0 --       initial value
fred[3]='abc' --indexed value
```

or retrieved:

Listing 1.17: Retrieving

```
say fred[3] --would say "abc"
```

in the latter case, the simple (initial) value of the variable is returned if the index has not been used to set a value. For example, the program:

Listing 1.18: Woof

```
bark='woof'
bark['pup']='yap'
bark['bulldog']='grrrrr'
say bark['pup'] bark['terrier'] bark['bulldog']
```

would display

```
yap woof grrrrr
```

Note that it is not necessary to use a number as the index; any expression may be used inside the brackets; the resulting string is used as the index. Multiple dimensions may be used, if required:

Listing 1.19: Multiple Dimensions

```
bark='woof'
bark['spaniel', 'brown']='ruff'
bark['bulldog']='grrrrr'
animal='dog'
say bark['spaniel', 'brown'] bark['terrier'] bark['bull'animal]
```

which would display

```
ruff woof grrrrr
```

Here's a more complex example using indexed strings, a test program with a function (called a *static method* in NetRexx) that removes all duplicate words from a string of words:

Listing 1.20: justonetest.nrx

```
/* justonetest.nrx --test the justone function. */
say justone('to be or not to be') /* simple testcase */
exit
/* This removes duplicate words from a string, and */
/* shows the use of a variable (HADWORD) which is */
/* indexed by arbitrary data (words).    */
method justone(wordlist) static
  hadword=0     /* show all possible words as new */
  outlist=''       /* initialize the output list */
  loop while wordlist\='' /* loop while we have data */
    /* split WORDLIST into first word and residue */
    parse wordlist word wordlist
    if hadword[word] then iterate /* loop if had word */
    hadword[word]=1 /* remember we have had this word */
    outlist=outlist word /* add word to output list */
    end
return outlist /* finally return the result */
```

Running this program would display just the four words "to", "be", "or", and "not".

1.8 Arrays

NetRexx also supports fixed-size *arrays*. These are an ordered set of items, indexed by integers. To use an array, you first have to construct it; an individual item may then be selected by an index whose value must be in the range `0` through `n-1`, where `n` is the number of items in the array:

Listing 1.21: Arrays

```
array=String[3] --  make an array of three Strings
array[0]='String one' --set each array item
array[1]='Another string'
array[2]='foobar'
loop i=0 to 2 --    display the items
  say array[i]
  end
```

This example also shows NetRexx *line comments*; the sequence "--" (outside of literal strings or "/*" comments) indicates that the remainder of the line is not part of the program and is commentary.

NetRexx makes it easy to initialize arrays: a term which is a list of one or more expressions, enclosed in brackets, defines an array. Each expression initializes an element of the array. For example:

Listing 1.22: Initializing elements

```
words=['Ogof', 'Ffynnon', 'Ddu']
```

would set words to refer to an array of three elements, each referring to a string. So, for example, the instruction:

Listing 1.23: Address Array Element

```
say words[1]
```

would then display

```
Ffynnon
```

1.9 Things that aren't strings

In all the examples so far, the data being manipulated (numbers, words, and so on) were expressed as a string of characters. Many things, however, can be expressed more easily in some other way, so NetRexx allows variables to refer to other collections of data, which are known as *objects*.

Objects are defined by a name that lets NetRexx determine the data and methods that are associated with the object. This name identifies the type of the object, and is usually called the *class* of the object.

For example, an object of class Oblong might represent an oblong to be manipulated and displayed. The oblong could be defined by two values: its width and its height. These values are called the *properties* of the Oblong class.

Most methods associated with an object perform operations on the object; for example a size method might be provided to change the size of an Oblong object. Other methods are used to construct objects (just as for arrays, an object must be constructed before it can be used). In NetRexx and Java, these *constructor* methods always have the same name as the class of object that they build ("Oblong", in this case).

Here's how an Oblong class might be written in NetRexx (by convention, this would be written in a file called `Oblong.nrx`; implementations often expect the name of the file to match the name of the class inside it):

Listing 1.24: Oblong

```
/* Oblong.nrx -- simple oblong class */
class Oblong

  width    -- size (X dimension)
  height   -- size (Y dimension)

  /* Constructor method to make a new oblong */
  method Oblong(new_width, new_height)
    -- when we get here, a new (uninitialized) object has been
    -- created. Copy the parameters we have been given to the
    -- four properties of the object:
    width=new_width; height=new_height

    /* Change the size of a Oblong */
  method size(new_width, new_height) returns Oblong
    width=new_width; height=new_height
    return this -- return the resized object

    /* Change the size of a Oblong, relatively */
  method relsize(rel_width, rel_height) returns Oblong
    width=width+rel_width; height=height+rel_height
    return this

    /* 'Print' what we know about the oblong */
  method print()
    say 'Oblong' width 'x' height
```

To summarize:

1. A class is started by the **class** instruction, which names the class.
2. The **class** instruction is followed by a list of the properties of the object. These can be assigned initial values, if required.
3. The properties are followed by the methods of the object. Each method is introduced by a **method** instruction which names the method and describes the arguments that must be supplied to the method. The body of the method is ended by the next method instruction (or by the end of the file).

The `Oblong.nrx` file is compiled just like any other NetRexx program, and should create a *class file* called `Oblong.class`. Here's a program to try out the Oblong class:

Listing 1.25: Try Oblong

```
/* tryOblong.nrx -- try the Oblong class */
first=Oblong(5,3)  -- make an oblong
first.print        -- show it
first.relsize(1,1).print -- enlarge and print again
second=Oblong(1,2) -- make another oblong
second.print       -- and print it
```

When tryOblong.nrx is compiled, you'll notice (if your compiler makes a cross-reference listing available) that the variables `first` and `second` have type `Oblong`. These variables refer to Oblongs, just as the variables in earlier examples referred to NetRexx strings.

Once a variable has been assigned a type, it can only refer to objects of that type. This helps avoid errors where a variable refers to an object that it wasn't meant to.

1.9.1 Programs are classes, too

It's worth pointing out, here, that all the example programs in this overview are in fact classes (you may have noticed that compiling them with the reference implementation creates `xxx.class` files, where `xxx` is the name of the source file). The environment underlying the implementation will allow a class to run as a stand-alone *application* if it has a static method called `main` which takes an array of strings as its argument.

If necessary (that is, if there is no class instruction) NetRexx automatically adds the necessary class and method instructions for a stand-alone application, and also an instruction to convert the array of strings (each of which holds one word from the command string) to a single NetRexx string.

The automatic additions can also be included explicitly; the "toast" example could therefore have been written:

Listing 1.26: New Toast

```
  /* This wishes you the best of health. */
  class toast
    method main(argwords=String[]) static
      arg=Rexx(argwords)
      say 'Cheers!'
```

though in this program the argument string, `arg`, is not used.

1.10 Extending classes

It's common, when dealing with objects, to take an existing class and extend it. One way to do this is to modify the source code of the original class – but this isn't always available, and with many different people modifying a class, classes could rapidly get overcomplicated.

Languages that deal with objects, like NetRexx, therefore allow new classes of objects to be set up which are derived from existing classes. For example, if you wanted a different kind of Oblong in which the Oblong had a new property that would be used when printing the Oblong as a rectangle, you might define it thus:

Listing 1.27: charOblong.nrx

```
/* charOblong.nrx -- an oblong class with character */
class charOblong extends Oblong
  printchar         -- the character for display
  /* Constructor to make a new oblong with character */
  method charOblong(newwidth, newheight, newprintchar)
    super(newwidth, newheight) -- make an oblong
    printchar=newprintchar -- and set the character
  /* 'Print' the oblong */
  method print
    loop for super.height
      say printchar.copies(super.width)
      end
```

There are several things worth noting about this example:

1. The "`extends Oblong`" on the class instruction means that this class is an extension of the Oblong class. The properties and methods of the Oblong class are *inherited* by this class (that is, appear as though they were part of this class). Another common way of saying this is that "`charOblong`" is a *subclass* of "`Oblong`" (and "`Oblong`" is the *superclass* of "`charOblong`").
2. This class adds the `printchar` property to the properties already defined for Oblong.
3. The constructor for this class takes a width and height (just like Oblong) and adds a third argument to specify a print character. It first invokes the constructor of its superclass (Oblong) to build an Oblong, and finally sets the printchar for the new object.
4. The new charOblong object also prints differently, as a rectangle of characters, according to its dimension. The `print` method (as it has the same name and arguments – none – as that of the superclass) replaces (overrides) the `print'` method of Oblong.
5. The other methods of Oblong are not overridden, and therefore can be used on charOblong objects.

The `charOblong.nrx` file is compiled just like `Oblong.nrx` was, and should create a file called `charOblong.class`.

Here's a program to try it out

Listing 1.28: tryCharOblong.nrx

```
/* trycharOblong.nrx -- try the charOblong class */
first=charOblong(5,3,'#') -- make an oblong
```

```
3    first.print          -- show it
4    first.relsize(1,1).print -- enlarge and print again
5    second=charOblong(1,2,'*') -- make another oblong
6    second.print         -- and print it
```

This should create the two charOblong objects, and print them out in a simple "character graphics" form. Note the use of the method `relsize` from Oblong to resize the charOblong object.

1.10.1 Optional arguments

All methods in NetRexx may have optional arguments (omitted from the right) if desired. For an argument to be optional, you must supply a default value. For example, if the charOblong constructor was to have a default value for printchar, its method instruction could have been written

Listing 1.29: Default value X

```
1 method charOblong(newwidth, newheight, newprintchar='X')
```

which indicates that if no third argument is supplied then `'X'` should be used. A program creating a charOblong could then simply write:

Listing 1.30: Default value

```
1 first=charOblong(5,3) -- make an oblong
```

which would have exactly the same effect as if `'X'` were specified as the third argument.

1.11 Tracing

NetRexx tracing is defined as part of the language. The flow of execution of programs may be traced, and this trace can be viewed as it occurs (or captured in a file). The trace can show each clause as it is executed, and optionally show the results of expressions, etc. For example, the **trace results** in the program "`trace1.nrx`":

Listing 1.31: Trace

```
1    trace results
2    number=1/7
3    parse number before '.' after
4    say after'.'before
```

would result in:

```
      --- trace1.nrx
    2 *=* number=1/7
      >v> number "0.142857143"
    3 *=* parse number before '.' after
      >v> before "0"
      >v> after "142857143"
    4 *=* say after'.'before
      >>> "142857143.0"
  142857143.0
```

where the line marked with "---" indicates the context of the trace, lines marked with "*=*" are the instructions in the program, lines with ">v>" show results assigned to local variables, and lines with "»>" show results of unnamed expressions.

Further, **trace methods** lets you trace the use of all methods in a class, along with the values of the arguments passed to each method. Here's the result of adding `trace methods` to the Oblong class shown earlier and then running `tryOblong`:

```
        --- Oblong.nrx
      8 *=*     method Oblong(newwidth, newheight)
        >a> newwidth "5"
        >a> newheight "3"
     26 *=*     method print
    Oblong 5 x 3
     20 *=*     method relsize(relwidth, relheight)-
 21 *-*
    >a> relwidth "1"
    >a> relheight "1"
 26 *=*     method print
Oblong 6 x 4
returns Oblong
     10 *=*     method Oblong(newwidth, newheight)
        >a> newwidth "1"
        >a> newheight "2"
     26 *=*     method print
    Oblong 1 x 2
```

where lines with ">a>" show the names and values of the arguments.

It is often useful to be able to find out when (and where) a variable's value is changed. The **trace var** instruction does just that; it adds names to or removes names from a list of monitored variables. If the name of a variable in the current class or method is in the list, then **trace results** is turned on for any assignment, **loop**, or **parse** instruction that assigns a new value to the named variable.

Variable names to be added to the list are specified by listing them after the **var** keyword. Any name may be optionally prefixed by a – sign., which indicates that the variable is to be removed from the list.

For example, the program "`trace2.nrx`":

Listing 1.32: trace2.nrx

```
1   trace var a b -- now variables a and b will be traced
2   a=3
3   b=4
4   c=5
5   trace var -b c -- now variables a and c will be traced
6   a=a+1
7   b=b+1
8   c=c+1
9   say a b c
```

would result in:

```
        --- trace2.nrx
```

```
  3 *=* a=3
    >v> a "3"
  4 *=* b=4
    >v> b "4"
  8 *=* a=a+1
    >v> a "4"
 10 *=* c=c+1
    >v> c "6"
4 5 6
```

1.12 Binary types and conversions

Most programming environments support the notion of fixed-precision "primitive" binary types, which correspond closely to the binary operations usually available at the hardware level in computers. For the reference implementation, these types are:

- *byte*, *short*, *int*, and *long* – signed integers that will fit in 8, 16, 32, or 64 bits respectively
- *float* and *double* – signed floating point numbers that will fit in 32 or 64 bits respectively.
- *char* – an unsigned 16-bit quantity, holding a Unicode character
- *boolean* – a 1-bit logical value, representing 0 or 1 ("false" or "true").

Objects of these types are handled specially by the implementation "under the covers" in order to achieve maximum efficiency; in particular, they cannot be constructed like other objects – their value is held directly. This distinction rarely matters to the NetRexx programmer: in the case of string literals an object is constructed automatically; in the case of an `int` literal, an object is not constructed.

Further, NetRexx automatically allows the conversion between the various forms of character strings in implementations[3] and the primitive types. The "golden rule" that is followed by NetRexx is that any automatic conversion which is applied must not lose information: either it can be determined before execution that the conversion is safe (as in `int` to `String`) or it will be detected at execution time if the conversion fails (as in `String` to `int`).

The automatic conversions greatly simplify the writing of programs; the exact type of numeric and string-like method arguments rarely needs to be a concern of the programmer. For certain applications where early checking or performance override other considerations, the reference implementation of NetRexx provides options for different treatment of the primitive types:

1. **options strictassign** – ensures exact type matching for all assignments. No conversions (including those from shorter integers to longer ones) are applied. This option provides stricter type-checking than most other languages, and ensures that all types are an exact match.

[3] In the reference implementation, these are String, char, char[] (an array of characters), and the NetRexx string type, Rexx.

2. **options binary** – uses implementation-dependent fixed precision arithmetic on binary types (also, literal numbers, for example, will be treated as binary, and local variables will be given "native" types such as `int` or `String`, where possible).

Binary arithmetic currently gives better performance than NetRexx decimal arithmetic, but places the burden of avoiding overflows and loss of information on the programmer.

The options instruction (which may list more than one option) is placed before the first class instruction in a file; the **binary** keyword may also be used on a **class** or **method** instruction, to allow an individual class or method to use binary arithmetic.

1.12.1 Explicit type assignment

You may explicitly assign a type to an expression or variable:

Listing 1.33: Assigning Type

```
i=int 3000000 -- 'i' is an 'int' with value 3000000
j=int 4000000 -- 'j' is an 'int' with value 4000000
k=int -- 'k' is an 'int', with no initial value
say i*j -- multiply and display the result
k=i*j -- multiply and assign result to 'k'
```

This example also illustrates an important difference between **options nobinary** and **options binary**. With the former (the default) the **say** instruction would display the result "`1.20000000E+13`" and a conversion overflow would be reported when the same expression is assigned to the variable k.

With **options binary**, binary arithmetic would be used for the multiplications, and so no error would be detected; the say would display "–138625024" and the variable `k` takes the incorrect result.

1.12.2 Binary types in practice

In practice, explicit type assignment is only occasionally needed in NetRexx. Those conversions that are necessary for using existing classes (or those that use **options binary**) are generally automatic. For example, here is an Applet for use by Java-enabled browsers:

Listing 1.34: A Simple Applet

```
  /* A simple graphics Applet */
  class Rainbow extends Applet
   method paint(g=Graphics) -- called to repaint window
     maxx=size.-width1
     maxy=size.-height1
     loop y=0 to maxy
       col=Color.getHSBColor(y/maxy, 1, 1) -- new colour
       g.setColor(col)            -- set it
       g.drawLine(0, y, maxx, y)  -- fill slice
    end y
```

In this example, the variable `col` will have type `Color`, and the three arguments to the method `getHSBColor` will all automatically be converted to type float. As no overflows are possible in this example, **options binary** may be added to the top of the program with no other changes being necessary.

1.13 Exception and error handling

NetRexx does not have a **goto** instruction, but a **signal** instruction is provided for abnormal transfer of control, such as when something unusual occurs. Using **signal** raises an *exception*; all control instructions are then "unwound" until the exception is caught by a control instruction that specifies a suitable catch instruction for handling the exception.

Exceptions are also raised when various errors occur, such as attempting to divide a number by zero. For example:

Listing 1.35: Exception

```
1  say 'Please enter a number:'
2  number=ask
3  do
4    say 'The reciprocal of' number 'is:' 1/number
5  catch Exception
6    say 'Sorry, could not divide "'number'" into 1'
7    say 'Please try again.'
8  end
```

Here, the **catch** instruction will catch any exception that is raised when the division is attempted (conversion error, divide by zero, *etc.*), and any instructions that follow it are then executed. If no exception is raised, the **catch** instruction (and any instructions that follow it) are ignored.

Any of the control instructions that end with **end** (**do**, **loop**, or **select**) may be modified with one or more **catch** instructions to handle exceptions.

1.14 Summary and Information Sources

The NetRexx language, as you will have seen, allows the writing of programs for the Java environment with a minimum of overhead and "boilerplate syntax"; using NetRexx for writing Java classes could increase your productivity by 30% or more. Further, by simplifying the variety of numeric and string types of Java down to a single class that follows the rules of Rexx strings, programming is greatly simplified. Where necessary, however, full access to all Java types and classes is available.

Other examples are available, including both stand-alone applications and samples of applets for Java-enabled browsers (for example, an applet that plays an audio clip, and another that displays the time in English). You can find these from the NetRexx web pages, at `http://www.netrexx.org`. Also at that location, you'll find the NetRexx language specification and other information, and downloadable packages containing the NetRexx software and documentation. There is a large selection of NetRexx examples available at `http://www.rosettacode.org`. The software should run on any platform that has a Java Virtual Machine (JVM) available.

2

Requirements

NetRexx 3.06-GA runs on a wide variety of hardware and operating systems; all releases are tested on (non-exhaustive):

1. Windows Desktop and Server editions, with OpenJDK, Oracle and IBM JVMs
2. Linux, with OpenJDK, Oracle and IBM JVMs, including z/Linux
3. MacOSX with OpenJDK and Oracle JVM; Apple JVM 1.6 is supported.
4. Android on ARM hardware with Dalvik virtual machine
5. z/OS, z/Linux with IBM JVM.
6. eComstation 2.x (OS/2) with eComstation Java 1.6
7. The Raspberry Pi, using Raspbian Linux and its included Oracle JDK

NetRexx runs equally well on 32- or 64-bit JVMs. As the translator is a command line tool, no graphics configuration is required, and headless operation is supported. Care has been taken to keep the NetRexx runtime small.

The class file format, however, of the current release distribution, is

.; for older formats, you can build NetRexx yourself or request assistance from the development team[4] for a special build.

Since release 3.01, NetRexx requires only a JRE[5] for program development, where previously a Java SDK[6] (earlier name: JDK) was required. For serious development purposes a Java SDK is recommended, as the tools found therein might assist the development process.

[4] developers@netrexx.kenai.com; you will need to be member of the Kenai NetRexx project

3

Installation

This chapter of the document tells you how to unpack, install, and test the NetRexx translator package. This will install documentation, samples, and executables. It will first state some generic steps that are sufficient for most users. The appendices contain very specific instructions for a range of platforms that NetRexx is used on. Note that to run any of the samples, or use the NetRexx translator, you must have already installed the Java runtime (and toolkit, if you want to compile NetRexx programs using the default compiler). The NetRexx samples and translator, version 3.06-GA, are guaranteed to run on Java version 1.6 or later; the programs using the NetRexxR.jar runtime library will run on earlier versions of many JVM's.[7] For ease of development and the availability of additional Java tools, a Java SDK can be installed, but NetRexx programs can be interpreted or compiled on a Java JRE installation[8]. By default the built-in (same compiler classes as javac uses) compiler of the Java SDK is used. You can test whether Java is installed, and its version, by trying the following command at a command prompt:

```
java -version
```

which should display a response similar to this:

```
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
```

For more information on Java installation, see the Oracle Java web page[9] – or other suppliers of Java toolkits.

3.1 Unpacking the NetRexx package

The NetRexx package is shipped as a collection of files compressed into the file NetRexx<version>.zip. Most modern operating environments can uncompress a .zip package by doubleclicking.

3.1.1 Unpacking the NetRexx.zip file

An unzip command is included in most Linux distributions, and Mac OSX. You can also use the jar command which comes with all Java development kits, with the options xvf. Choose where you want the NetRexx directory tree to reside, and unpack the zip file

[7] For earlier versions of Java, NetRexx 2.05 is available from the NetRexx.org website.
[8] See chapter 5
[9] at `http://www.javasoft.com`

in the directory which will be the parent of the NetRexx tree. Here are some tips: The syntax for unzipping NetRexx3.06-GA.zip is simply

```
unzip  NetRexx3.06-GA.zip
```

which should create the files and directory structure directly.

- WinZip: all versions may be used
- Linux unzip: use the syntax: unzip –a `NetRexx3.06-GA.zip`. The "–a" flag will automatically convert text files to Unix format if necessary
- jar: The syntax for unzipping the package is

  ```
  jar xvf  NetRexx3.06-GA.zip
  ```

which should create the files and directory structure directly. The "x" indicates that the contents should be extracted, and the "f" indicates that the zip file name is specified, the "v" is for verbose. Note that the extension (.zip) is required.

After unpacking, the following directories should have been created:

3.2 The NetRexx packages

In the *lib* subdirectory, there are three java archive files (*jars*), which are called:

NetRexxF.jar The translator (and runtime) package including the ecj[10] java compiler
NetRexxC.jar The translator (and runtime) package without java compiler
ecj-4.4.2.jar The eclipse java compiler package

The *runlib* directory contains one java archive:

NetRexxR.jar A minimal package including only the runtime NetRexx classes - for distribution with NetRexx programs

It is advised to start with the NetRexxF.jar archive package. This can be used for your first NetRexx activities in a way that is independent of the Java *classpath*, or the Java installation - a development installation (JDK) or just the java runtime (JRE). This enables you to interpret, or compile NetRexx programs to .class files. The NetRexxC.jar package is used by experienced NetRexx users; it requires a correct setting of the *classpath* environment variable (explicitly, or implicitly by adding it to the JVM standard extension directory) to find the java compiler (either the JDK included *javac* classes or the included eclipse compiler) - on a JDK or JRE installation. The NetRexxR.jar contains only the runtime of the NetRexx language. It can be added to compiled NetRexx applications if a small footprint is required. The following paragraph discusses getting the compiler to translate your first program using the NetRexxF.jar - after that the process of adding the translator to your environment is shown, what we will call 'installing' here. There is no requirement for a 'setup' type of install, and when you can execute Java on your system, there is no need to be 'Administrator' or 'root' on your system - NetRexx runs fine from your home directory.

[10] Eclipse Compiler for Java

3.3 First steps with NetRexx

1. Verify the working of java on your system with the command: java -version
   If this does not work, obtain a version at `http://java.com` and install it.
2. Create a file named hello.nrx in the directory that contains NetRexxF.jar, that contains the line:

   ```
   say 'hello, netrexx world!'
   ```

   You can copy this file from the ../bin directory.
3. For Windows environments, add the bin directory to your PATH environment variable. The nrc.bat command takes care of adding the NetRexxF.jar library to your CLASSPATH environment variable, so you can just run with:

   ```
   nrc -exec hello
   ```

   To compile to a java .class file, leave out the -exec option. If this works, you can skip the other steps (or read on, to get a feel for the working of the CLASSPATH environment).
4. In this directory, verify the working of the interpreter with:

   ```
   java -jar NetRexxF.jar -exec hello
   ```
5. Verify the creating of a .class file using the compiler with:

   ```
   java -jar NetRexxF.jar hello
   ```

   This should create hello.class, to be executed with the command:

   ```
   java -cp NetRexxF.jar:. hello
   ```

   (on windows, the colon should be a semicolon)

The -jar directive tells the JVM to ignore the set classpath and to start a method that is indicated in the jar metadata.This method is, for the NetRexxF.jar:

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

just as shown in 3.7 on page 23. Now that you have seen that it works, you can use this method of execution[11], or proceed with installing a more flexible way of using NetRexx.

When a class calls another class that is located in the same directory, we need to add this directory to the *classpath*. For example, if we want to compile the charOblong.nrx example from page 11, which extends the Oblong class, we need to invoke it as:

```
java -jar NetRexxF.jar -cp NetRexxF.jar;. charOblong.nrx
```

This can be done in a more straightforward way, by installing the NetRexxC.jar on the classpath and using the provided `nrc` script; this is the subject of the next section.

3.4 Installing the NetRexx Translator

The NetRexx package includes the NetRexx translator – a Java application which can be used for compiling, interpreting, or syntax-checking NetRexx programs. The procedure for installation is as follows[12]:

[11] Taking into account that you will have to add additional entries to the -jar argument for all but the most trivial applications.
[12] For Windows operating systems, forward slashes are backslashes.

1. Make the translator visible to the Java Virtual Machine (JVM) - either:
   - Add the full path and filename of the NetRexx/lib/NetRexxC.jar to the CLASSPATH environment variable for your operating system.[13]
   - Or (deprecated): Copy the file NetRexx/lib/NetRexxC.jar to the jre/lib/ext directory in the Java installation tree. The JVM will automatically find it there and make it available[14].
2. Copy all the files in the NetRexx/bin directory to a directory in your PATH. This is not essential, but makes shorthand scripts and a test case available.
3. Make the file [...]/lib/tools.jar (which contains the javac compiler) in the Java tree visible to the JVM. You can do this either by adding its path and filename to the CLASSPATH environment variable, or by moving it to the jre/lib/ext directory in the Java tree. This file sometime goes under different names, that will be mentioned in the platform-specific appendices.

3.5 Installing just the NetRexx Runtime

If you only want to run NetRexx programs and do not wish to compile or interpret them, or if you would like to use the NetRexx string (Rexx) classes from other languages, you can install just the NetRexx runtime classes.
To do this, follow the appropriate instructions for installing the compiler, but use the NetRexxR.jar instead of NetRexxC.jar. The NetRexxR.jar file can be found in the *NetRexx/runlib* directory.
You do not need to use or copy the executables in the *NetRexx/bin* directory.
The NetRexx class files can then be referred to from Java or NetRexx programs by importing the package *netrexx.lang*. For example, a string might be of class *netrexx.lang.Rexx*.
For information on the *netrexx.lang.Rexx* class and other classes in the runtime, see the *NetRexx Language Reference* document.

note If you have already installed the NetRexx translator (NetRexxC.jar) then you do not need to install NetRexxR.jar; the latter contains only the NetRexx runtime classes, and these are already included in NetRexxC.jar.

3.6 Setting the 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 for Unix versions, which shell you are using.

[13] if you have a NetRexxC.zip in your CLASSPATH from an earlier version of NetRexx, remove it (NetRexxC.jar replaces NetRexxC.zip).

[14] This has serious drawbacks, however: As soon as the Java version is updated, NetRexx applications may mysteriously – due to the now obsolete path - fail. The contents of the extensions directory are unversioned. Running multiple versions of Java and NetRexx for testing purposes, or with an application that included another version of NetRexx will become very hard when this way of installing is chosen.

- For Linux, MacOSX and other Unix versions (BASH (bash), Korn (ksh), or Bourne (sh) shell), use:

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

- This should be placed in your /.bash_profile, /etc/profile, .login, or .profile file, as appropriate. The environment changes can be made active by running, for example,

```
. .bash_profile
```

  in your home directory, when this location is where you made the changes.
- For Linux, MacOSX and other Unix versions (C shell (csh and tcsh)), use:

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

  These should be set in your .cshrc file (csh) or .tcshrc (tcsh). The `rehash` command can be used to activate these changes in the environment. 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).
- In Windows *Powershell*, limitations set by the administrator can determine which kind of scripting (using Powershell, not NetRexx) can be undertaken. It might be difficult to modify the enviroment, and a different from scripting under the `cmd.exe` processor is that the environment is local to an execution unit of a line. When changing the environment is allowed, and a Powershell script is used to start the NetRexxtranslator, this is how it can be done:

```
$env:path = "c:\program files\java\jdk1.7.0_02\bin;\Users\rvj\bin;"
$env:classpath = ".;\Users\rvj\lib\NetRexxC.jar"
```

- For pre 3.04 versions of NetRexx, when using an IBM JVM or JRE, make sure that the file vm.jar is on the CLASSPATH - NetRexx will complain about missing java.lang.Object when it is not. NetRexx 3.04 and later are looking up the boot-classpath in a correct manner to avoid this problem.

In case of encountering difficulties in getting the classpath settings to work, the following remarks can be helpful:

- Spaces in directory names are OK, but these paths must be surrounded by double quotes in most environments, like Windows and Unix
- Non-existing directories in classpaths can hurt - move the NetRexxC.jar path to the beginning of classpath to eliminate the risk of non-existing directories.

3.7 Testing the NetRexx Installation

After installing NetRexx, it is recommended that you test that it is working correctly. If there are any problems, check the *Troubleshooting* section of this document, chapter 14 on page 57.

Test the installation by typing in a file named 'hello.nrx' containing the line:

```
say 'hello, world'
```

If you want to avoid typing in the file yourself,

```
./examples/ibm-historic/hello.nrx
```

has the original version of this program.

1. Enter the command

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

   Make sure that the userid that you are using for this has write authorization for the directory that contains the source.[15] This should run the NetRexx compiler, which first translates the NetRexx program hello.nrx to the Java program hello.java. It then invokes the default Java compiler (javac[16]), to compile the file hello.java to make the binary class file hello.class. The intermediate hello.java file is then deleted, unless an error occurred or you asked for it to be kept. You can also specify the source filename as 'hello.nrx' - for convenience, the translator will look for a file with a '.nrx' suffix if this is not specified.
2. Enter the command

   ```
   java hello
   ```

   This runs (interprets the bytecodes in) the hello.class file, which should display a simple greeting. On some systems, you may first have to add the directory that contains the hello.class file to the CLASSPATH setting so Java can find it.
3. With the sample scripts provided (NetRexxC.cmd, NetRexxC.bat, or NetRexxC.sh), or the equivalent in the scripting language of your choice, the steps above can be combined into a simple single command such as:

   ```
   NetRexxC.sh –run hello
   ```

   This package also includes a trivial nrc, and matching nrc.cmd and nrc.bat scripts, which simply pass on their arguments to NetRexxC; "nrc" is just a shorter name that saves keystrokes, so for the last example you could type:

   ```
   nrc –run hello
   ```

   Note that scripts may be case-sensitive, and you will probably have to spell the name of the program exactly as it appears in the filename. Also, to use –run, you may need to omit the .nrx extension. You could also edit the appropriate nrc.cmd, nrc.bat, or nrc script and add your favourite "default" NetRexxC options there. For example, you might want to add the –prompt flag (described later) to save reloading the translator before every compilation. If you do change a script, keep a backup copy so that if you install a new version of the NetRexx package you won't overwrite your changes. On Unix versions, do not forget to make the scripts nrc and NetRexxC.sh executable with the command `chmod +x` *scriptname*. Also on Unix versions, it is better to use a command alias to start java classes; it avoids problems with the splitting of strings on the command line. This is a workable set of aliases to go into a `.bash_profile` script:

[15] For example, more modern versions of Windows do not allow non-admin userids to write into the program files directories. In this case, make a directory under your home directory and copy the hello.nrx file there, and start the nrc command from the same location. Running it from the examples directory will work.

[16] In fact, the class that the javac program also calls for compilation - but you can use other java compilers

```
alias nrc=”java -cp $CLASSPATH org.netrexx.process.NetRexxC”
alias pipe=”java org.netrexx.njpipes.pipes.compiler”
alias nrs=”jrunscript -l netrexx -cp ~/lib/NetRexxC.jar”
```

4

Unicode

The JVM works with Unicode as a string representation; for this reason the display of characters in alphabets other than the latin alphabet does not pose a problem. To work with Unicode and internationalization in a straightforward way, a combination of factors must be present. The operating system, your editor, shell and character set support must be compatible with Unicode. A set fonts very seldom contains glyphs[17] for all Unicode code points (values). Be certain to save the program file as the right type; some editors can save as ASCII, UTF-8 and UTF-16. Some editors seem to support Unicode but have made mistakes in the implementation. The NetRexx translator has a `-utf8` option that makes it accept this encoding in the source. This option is not necessary for the use of Unicode in *variables* - this always works, it being the native encoding of the JVM. The option is rather meant to enable specification of NetRexx syntax elements in Unicode. This makes it possible to use Class names, Method names and variable names composed of Unicode characters.

Some things to think of when using the `-utf8` option:

- It is not the default.
- The option `-utf8` can be specified in the program source, but the value of this option on the compiler command line must be equal to the value of the program option. Here the rule that the last specified value for an option is applicable, does not count
- When method names are specified in Unicode, they need to be *symbols* and not escaped Unicode characters
- When Unicode is used in a Class name, the program file name must match the class name.
- A filename in Unicode might still spell trouble when using it in conjunction with version management software, sharing it using email or other usages that are not limited to one file system and encoding method.

[17] This is a typographical term for character form

5

Running on a JRE-only environment

5.1 Eclipse Batch Compiler

NetRexx can be used on a JRE-only environment; it does not need an SDK (JDK) when the included ecj (Eclipse Compiler for Java) jar file is available on the classpath. This compiler is a part of the Eclipse JDT Core, which is the Java infrastructure of the Java IDE. This is an incremental Java compiler. It is based on technology evolved from the VisualAge for Java compiler and maintained by IBM and the Eclipse Foundation. In particular, it allows one to run and debug code which still contains unresolved errors. Future releases of NetRexx might be exploring more of the features of this compiler, like the extensive error reporting. Currently, the ecj-4.4.2.jar level of the core compiler jar is delivered with NetRexx. There are other standalone Java compilers, but after extensive research we have chosen to include this one. Using the *–nocompile* and *–keepasjava* options it is always possible to substitute your own compilers as subsequent stages in the build process. 3.01

5.2 The -ecj and -javac translator options

The NetRexx language processor is a translator package that either interprets or executes NetRexx language source, and (by default) compiles the generated Java language source code with the SDK-included *javac* compiler, or rather, the Java compiler class sun.tools.javac.Main class that is delivered (in most implementations) in the tools.jar file, which is also called by the javac executable. An option is introduced to make the language processor choose the ecj compiler. 3.04

```
nrc -ecj sourcefile.nrx
```

This directs the NetRexxC processor to use the ecj compiler to do the java compile step instead of javac. This option can also be set to javac - which is still the default when the option is not specified. The `NetRexxC` command script can, on systems that do not have a javac compiler installed, be changed to state

```
java org.netrexx.process.NetRexxC -ecj $*
```

In this case all compiles started with the `nrc` command will use the Eclipse compiler. Only in case of Java compiler errors, when the compiler output will be shown, will the difference be apparent. Installer support is planned to include this property automatically when during NetRexx installation the javac compiler jar is not detected. When compiling using the `-time` option, the right compiler name will be indicated.

5.3 The netrexx_java environment variable

The NetRexxC compile scripts pass the environment variable `netrexx_java` to the Java VM at start. The compiler selection can be placed in the environment (in a slightly adapted and more historic form) and no change to the NetRexxC script is required. In Windows for example:

```
set netrexx_java=-Dnrx.compile=ecj
```

5.4 Passing options to the Java Compiler

3.04 A scan will be performed for a suitable compiler when the preferred one is not found.

The Java system property ”nrx.compiler” can be used to provide options for the Java compiler called by NetRexx. This property is set on starting the NetRexx translator as in this example:

```
java -Dnrx.compiler=”-target 1.6” org.netrexx.process.NetRexxC myprogram
```

If the first option specified is ”javac” or ”ecj”, NetRexx will use that option to prefer selection of a compiler although the ”-javac” and ”-ecj” translator options will override it. Other options are passed to the Java compiler unchanged. If you are using the Windows script ”nrc.bat” to compile programs, you can place the system property in the Windows environment to make it automatic as in this example:

```
set netrexx_java=-Dnrx.compiler=”ecj -source 1.6 -target 1.6”
```

The nrx.compiler property can also be set directly in Ant builds or via the Ant project property ”ant.netrexxc.javacompiler”.

5.5 Interpreting

For completeness, it is confirmed here that interpretative execution also works on a JRE-only system, and does not require a Java compiler. The NetRexx translator produces the required bytecode and proxy classes without any need for a Java compiler.

6

Using the translator

This section of the document tells you how to use the translator package. It assumes you have successfully installed Java and NetRexx, and have tested that the *hello.nrx* testcase can be compiled and run, as described in the *Testing the NetRexx Installation* (section 3.7 on page 23).

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

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

6.1 Using the translator as a compiler

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

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

6.2 The translator command

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

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

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

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

or by using a system-specific script (such as *NetRexxC.cmd*. or *nrc.bat*). In either case, the compiler invocation is followed by one or more file specifications (these are the

names of the files containing the NetRexx source code for the programs to be compiled).

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

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

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

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

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

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

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

Note that when more than one program is specified, they are all compiled within the same class context. That is, they can see the classes, properties, and methods of the other programs being compiled, much as though they were all in one file. [18] This allows mutually interdependent programs and classes to be compiled in a single operation. Note that if you use the **package** instruction you should also read the more detailed *Compiling multiple programs* section.

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

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

6.2.1 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*. On the commandline, options are prefixed with a *dash* ("-"), while in

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

programsource they are not - there they are preceded by the `option` statement.

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
ecj	prefer the ecj compiler	c
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
javac	prefer the javac compiler	c
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
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[19], 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.

ecj prefer the ecj compiler when available

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.

javac prefer the javac compiler when available

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.

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

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

Here are some examples:

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

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

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

6.3 Compiling multiple programs and using packages

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

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

Listing 6.1: Dependencies

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

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

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

```
nrc X Y
```

the cross-references will be resolved correctly.

The total elapsed time will be significantly less, too, as the classes on the CLASSPATH need to be located only once, and the class files used by the NetRexxC compiler or the

programs themselves will also only be loaded (and JIT-compiled) once.

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

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

```
D:\myproject
```

if the two programs above specified a package, thus:

Listing 6.2: Package Dependencies

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

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

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

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

Notes:

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

7

Programmatic use of the NetRexxC translator

NetRexxC can be used in a program, to compile NetRexx programs from files, or to compile from strings in memory.

7.1 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 7.1: 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 7.2: 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 7.3: Example of compiling from String

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

7.2 JSR199

NetRexx uses the jsr-199 way of invoking the compiler, and a .java file is not written to disk by default. The following program illustrates how a complete program can be compiled and executed without anything being written to disk. 3.04

Listing 7.4: JSR199 example

```
import org.netrexx.

pname="jsr199hello"                    --      program name
nrp=' say "hello"\n say arg \n say "program complete" ' -- NetRexx program code

```

```
6 classlist=ArrayList() --  this requests a class loader and class files returned in
      memory
7 NetRexxC.main(pname "-verbose0", nrp, null, classlist) -- ask NetRexxC to compile from
       string nrp
8 loader=ClassLoader classlist.get(0) -- find class loader build by NetRexx translator
9 pclass=loader.loadClass(pname) --   load our class file into the jvm
10 pclass.getMethod("main", [Class -
11 String[0].getClass()]).invoke(null,[Object [String 'argument
12 string']])    --      locate main, call it with reflection = all done!
```

Other uses of the NetRexxA API are beyond the scope of this Quick Start Guide and are documented in the *Programming Guide*.

8

Using the prompt option

The **prompt** option may be be used for interactive invocation of the translator. This requests that the processor not be ended after a file (or set of files) has been processed. Instead, you will be prompted to enter a new request. This can either repeat the process (perhaps if you have altered the source in the meantime), specify a new set of files, or alter the processing options.
On the second and subsequent runs, the processor will re-use class information loaded on the first run. Also, the classes of the processor itself (and the *javac* compiler, if used) will not need to be verified and JIT-compiled again. These savings allow extremely fast processing, as much as fifty times faster than the first run for small programs.
When you specify *-prompt* on a NetRexxC command, the NetRexx program (or programs) will initially be processed as usual, according to the other flags specified. Once processing is complete, you will be prompted thus:

```
Enter new files and additional options, '=' to repeat, 'exit' to end:
```

.
At this point, you may enter:

- One or more file names (with or without additional flags): the previous process, modified by any new flags, is repeated using the source file or files specified. Files named previously are not included in the process (unless they are named again in the new list of names).
- Additional flags (without any new files): the previous process, modified by the new flags, is repeated, on the same files as before. Note that flags are accumulated; that is, flags are not reset to defaults between prompts.
- The character = this simply repeats the previous process, on the same file or files (which may have had their contents changed since the last process) and using the same flags. This is especially useful when you simply wish to re-compile (or re-interpret, see below) the same file or files after editing.
- The word *exit*, which causes NetRexxC to cease execution without any more prompts.
- Nothing (just press Enter or the equivalent) – usage hints, including the full list of possible options, etc., are displayed and you are then prompted again.

9

Using the translator as an Interpreter

In addition to being used as a compiler, the translator also includes a true NetRexx interpreter, allowing NetRexx programs to be run on the Java 2 (1.2) platform without needing a compiler or generating .class files.
The startup time for running programs can therefore be significantly reduced as no Java source code or compilation is needed, and also the interpreter can give better runtime support (for example, exception tracebacks are localized to the programs being interpreted, and the location of an exception will be identified often to the nearest token in a term or expression).
Further, in a single run, a NetRexx program can be both interpreted and then compiled. This shares the parsing between the two processes, so the *.class* file is produced without the overhead of re-translating and re-checking the source.

9.0.1 Interpreting programs

The NetRexx interpreter is currently designed to be fully compatible with NetRexx programs compiled conventionally. There are some minor restrictions (see section 15 on page 59), but in general any program that NetRexxC can compile without error should run. In particular, multiple programs, threads, event listeners, callbacks, and Minor (inner) classes are fully supported.
To use the interpreter, use the NetRexxC command as usual and specify either of the following command options (flags):

-exec after parsing, execute (interpret) the program or programs by calling the static *main(String[])* method on the first class, with an empty array of strings as the argument. (If there is no suitable *main* method an error will be reported.)

-arg words... as for *-exec*, except that the remainder of the command argument string passed to NetRexxC will be passed on to the main method as the array of argument strings, instead of being treated as file specifications or flags. Specifying *-noarg* is equivalent to specifying *-exec*; that is, an empty array of argument strings will be passed to the main method (and any remaining words in the command argument string are processed normally).

When any of *-exec*, *-arg*, or *-noarg* is specified, NetRexxC will first parse and check the programs listed on the command. If no error was found, it will then run them by invoking the main method of the first class interpretively.
Before the run starts, a line similar to:

```
===== Exec: hello =====
```

will be displayed (you can stop this and other progress indicators being displayed by using the *-verbose0* flag, as usual).

Finally, after interpretation is complete, the programs are compiled in the usual way, unless *-nojava*[20] or *-nocompile* was specified.
For example, to interpret the hello world program without compilation, the command:

```
nrc hello -exec -nojava
```

can be used. If you are likely to want to re-interpret the program (for example, after changing the source file) then also specify the *-prompt* flag, as described above. This will give very much better performance on the second and subsequent interpretations.
Similarly, the command:

```
nrc hello -nojava -arg Hi Fred!
```

would invoke the program, passing the words *Hi Fred!* as the argument to the program (you might want to add the line *say arg* to the program to demonstrate this).
You can also invoke the interpreter directly from another NetRexx or Java program, as described in The NetRexx Programming Guide.

9.1 Interpreting – Hints and Tips

When using the translator as an interpreter, you may find these hints useful:

- If you can, use the *-prompt* command line option (see above). This will allow very rapid re-interpretation of programs after changing their source.
- If you don't want the programs to be compiled after interpretation, specify the *-nojava* option, unless you want the Java source code to be generated in any case (in which case specify *-nocompile*, which implies *-keep*).
- By default, NetRexxC runs fairly noisily (with a banner and logo display, and progress of parsing being shown). To turn off these messages during parsing (except error reports and warnings) use the *-verbose0* flag.
- If you are watching NetRexx trace output while interpreting, it is often a good idea to use the *-trace1* flag. This directs trace output to the standard output stream, which will ensure that trace output and other output (for example, from **say** instructions) are synchronized.
- Use the NetRexx **exit** instruction (rather than the *System.exit()* method call) to end windowing (AWT) applications which are to be interpreted. This will allow the interpreter to correctly determine when the application has ended.

9.2 Interpreting – Performance

The interpreter, in the current implementation, directly and efficiently interprets NetRexx instructions. However, to assure the stability of the code, terms and expressions within instructions are currently fully re-parsed and checked each time they are executed. This

[20] The *-nojava* flag stops any Java source being produced, so prevents compilation. This flag may be used to force syntax-checking of a program while preventing compilation, and with optional interpretation.

has the effect of slowing the execution of terms and expressions significantly; performance measurements on the initial release are therefore unlikely to be representative of later versions that might be released in the future.
For example, at present a loop controlled using *loop for 1000* will be interpreted around 50 times faster than a loop controlled by *loop i=1 to 1000*, even in a binary method, because the latter requires an expression evaluation each time around the loop.

10

Installing on an IBM Mainframe

10.0.1 EBCDIC Systems: z/OS, z/VM

Prerequisites for z/OS

To use NetRexx on z/OS you must have access to an OMVS prompt (z/OS Unix Systems Services[21] shell for 3270 terminals), or have access using ssh or telnet; Java must be installed.

Access to the OMVS command can be regulated through a security profile, so your userid must be in the right RACF, ACF2 or TOP SECRET class. You will need a home directory specified in this OMVS class, and this directory needs to be mounted, preferably as a permanent mount.

If this is arranged and working, you need to verify if there is a Java runtime available. Test this with the command

```
java -version
```

Java 1.6, or more recent, is needed for NetRexx 3.06-GA.

Uploading the NetRexx translator jar

The NetRexx binaries are identical for all operating systems; the same NetRexxC.jar runs everywhere[22]. However, during installation it is important to ensure that binary files are treated as binary files, whereas text files (such as the accompanying HTML and sample files) need to be translated to the local code page as required.

The simplest way to do this is to first install the package on a workstation, following the instructions above, then copy or FTP the files you need to the mainframe. The files need to be placed in an HFS to be used by OMVS; FTP and sftp can directly place the files in an HFS or ZFS home directory, while IND$FILE can place them into a traditional data set.

Specifically:

- The NetRexxC.jar file should be copied as-is, that is, use FTP or other file transfer with the BINARY option. Note that `sftp` defaults to binary, while `scp` to z/OS translates ASCII to EBCDIC and is not usable for this purpose. The CLASSPATH should be set to include this NetRexxC.jar file. When using IND$FILE as a file transfer mechanism to a traditional MVS data set, make sure it is allocated as a load library

[21] IBM Manuals SA22-7801-12 "Unix System Services User's Guide" and SA22-7802-12 "Unix System Services Reference"
[22] Thanks to Mark Cathcart and John Kearney for contributing the details to the original version of this section.

with `lrecl 0` and a large blocksize. A variable length record also works, for example, a dataset defined as `dsorg=ps, recfm=vb, lrecl=1250, blksize=12500` works without a problem.

- Other files (documentation, etc.) should be copied as Text (that is, they will be translated from ASCII to EBCDIC). This can be done by specifying type TEXT on the ftp command, or use the ASCII CRLF option on the IND$FILE command.

In general, files with extension *.au*, *.class*, *.gif*, *.jar*, or *.zip* are binary files; all others are text files. You may opt to leave the additional files on a workstation, the mainframe really only needs the .jar file, NetRexxC.jar (or NetRexxR.jar if you are only planning to run already compiled classfiles). Setting the classpath might look like this for a Java 1.6 installation on a recent z/OS:

```
JAVA_HOME=/opt/ibm/java-s390x-60
export JAVA_HOME
CLASSPATH=$CLASSPATH:$JAVA_HOME/lib/tools.jar
CLASSPATH=$CLASSPATH:$JAVA_HOME/jre/lib/s390x/default/jclSC160/vm.jar
CLASSPATH=$CLASSPATH:/u/[your userid]/lib/NetRexxC.jar
export CLASSPATH
```

For a Java 1.6.1 installation, the following settings were encountered:

```
JAVA_HOME=/usr/lpp/java/J6.0.1
export JAVA_HOME
CLASSPATH=$CLASSPATH:$JAVA_HOME/lib/tools.jar
CLASSPATH=$CLASSPATH:$JAVA_HOME/lib/s390/default/jclSC160/vm.jar
CLASSPATH=$CLASSPATH:/u/[your userid]/lib/NetRexxC.jar
export CLASSPATH
```

For a 64 bits Java 1.7.0 installation, these settings work:

```
JAVA_HOME=/usr/lpp/java/J7.0_64
export JAVA_HOME
CLASSPATH=$CLASSPATH:$JAVA_HOME/lib/tools.jar
CLASSPATH=$CLASSPATH:$JAVA_HOME/lib/s390x/default/jclSC170/vm.jar
CLASSPATH=$CLASSPATH:/u/[your userid]/lib/NetRexxC.jar
export CLASSPATH
```

Note that you are free to put the NetRexxC.jar archive in any location, as long as the classpath correctly refers to it. The vm.jar has to be on the classpath because otherwise Object.class will not be found by the NetRexxC translator.

The `OCOPY` command can be used under TSO to copy the uploaded NetRexxC.jar to a path in an HFS dataset:

```
/* rexx */
"free fi(pathname)"
"free fi(sysut1)"
"alloc fi(pathname) path('/u/[your userid]/lib/NetRexxC.jar')"
"alloc fi(sysut1) dsn('netrexx.new')"
"ocopy indd(sysut1) outdd(pathname) binary"
```

This works when the NetRexxC.jar file already exists, if that is not the case, just issue `touch NetRexxC.jar` in that directory, the copy command will overwrite that empty file.

Be sure to add the `-Xquickstart` option to the java command in the nrc binary file in your path, or add it as an alias.

```
java -Xquickstart org.netrexx.process.NetRexxC $*
```

because this will shorten the startup time required to a more or less acceptable time.

When this is done, we can run some tests with it and see that everything works. Edit a program source file with `oedit`, which works just like the ISPF/PDF editor and compile or interpret it like we do on other versions of Unix. NetRexx programs can access HFS (and ZFS) files the same way it does on Windows and Unix, and also network programming with TCP/IP works in the same way from OMVS.

For a description how NetRexx can be used in a traditional MVS workload environment, with batch JCL and using VSAM and sequential data sets and PDS directories, you are referred to the *NetRexx Programming Guide)*.

A note on character sets

z/OS USS is an EBCDIC Unix version, do note that the `-utf8` option does only work when your source file actually is encoded in utf8.

10.0.2 z/Linux

Installing on z/Linux is straightforward. Make sure the NetRexxC.jar is copied untranslated to the z/Linux file system using ftp, scp or some other file transfer technology, and take into account that the IBM JVM has Object.class in the vm.jar archive. At the moment, if not installed already, Java for z/Linux is a free download from the IBM website. With z/Linux versions that have a VNC server installed and available, Java Graphical User Interfaces (GUI) can be used without installing X client software.

11

ARM ABI Remarks

For the next two chapters, it is relevant to know about a specific issue with ARM processors, as used in both the Raspberry Pi and the Beaglebone Black, with regard to the JVM distribution that is used. ARM processors are available in many different configurations, and because of considerations of pricing and power requirements, not all of these include hardware floating point units. The difference between these is the reason of the existence of two Embedded Application Binary Interfaces or EABIs for ARM: soft float and VFP (Vector Floating Point). Although there is forward compatibility between soft and hard float, there is no backward compatibility. In the Linux community, releases built using these EABI's are called *armel* based distributions.

Unfortunately, VFP has an inefficient way of passing floating point values through intermediate integer registers to the floating point registers where they can be used. This has given rise to a third EABI, which is called *armhf*, also called *hard float*. This architecture can be seen as the future, because the important Linux distributions are moving towards it. Depending on the release of your operating system, your Raspberry Pi or Beaglebone Black's software can be operating in *armel* or *armhf* mode. The consequence of this is that the JVM implementation must match the architecture, or it will not work.

The JVM that are installed using the package manager that is native to the operating system will choose the right architecture. For the Oracle Java versions, it is important to know that the released version 7 JVM is soft-float *armel* and that there is currently a JVM 8 preview that is hard-float. The recommended OS for the Raspberry Pi is Debian Wheezy "Raspbian", which is hard float. The Beaglebone Black comes with Ångstrom Linux, which is soft-float and cannot run the Oracle Java 8 preview.

The easiest way to spot the architecture is to look for these components (armel of armhf) in the package names when installing software. There is a way to determine which EABI conventions were used, which is mentioned here for completeness: the command

```
readelf -A /proc/self/exe | grep Tag_ABI_VFP_args
```

returns:

```
Tag_ABI_VFP_args: VFP registers
```

when the OS distribution is *armhf* and nothing, when it is *armel*.

12

Installing and running on the BeagleBone Black

12.0.1 Starting with an unmodified system

The following instructions assume a new system, running the default Ångstrom Linux distribution. Since NetRexx is an alternative language for the JVM, you must first have Java installed on the BeagleBone Black.

12.0.2 Install Java

- From the Ångstrom repository.
  Login as *root*

  ```
  opkg update
  opkg install openjdk-6-jdk
  ```

  If that fails (for one reason or another), install the pieces of Java step-by-step:

  ```
  opkg install openjdk-6-common
  opkg install openjdk-6-java
  opkg install openjdk-6-jre
  opkg install openjdk-6-jdk
  opkg install openjdk-6-vm-zero
  ```

  And if *that* fails (for whatever reason), go directly to the repositories at `http://www.angstrom-distribution.org/repo/` and fetch the packages individually *by direct URL*, using the list above in order:

  ```
  opkg install <URL>
  ```

- From Oracle
  Download the JDK from `http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.htm`. Ensure that:
  - You accept the license agreement, and
  - select the "Linux ARM" version

  As this is written, the file to download is `http://download.oracle.com/otn-pub/java/jdk/7u25-b15/jdk-7u25-linux-arm-sfp.tar.gz`
  Then, while logged in as root:
  - `mkdir /usr/java`
  - Move the downloaded file to /usr/java
  - tar zxvf jdk-7u25-linux-arm-sfp.tar.gz
  - Delete the downloaded file (optional, but saves space)

  Add */usr/java/jdk1.7.0_25/bin* to the PATH:
  - Edit /etc/profile

- Insert PATH=$PATH:/usr/bin/jdk1.7.0_25/bin

Somewhere between the existing PATH and the final export statements.

12.0.3 Install NetRexx

Download the NetRexx distribution

```
wget http://netrexx.org/files/ NetRexx3.06-GA.zip
```

Create or select a destination directory (like */usr/netrexx/*), move the downloaded file there, and

```
unzip  NetRexx3.06-GA.zip
```

Then simply follow the NetRexx recommendations to finalize the installation.

13

Installing and running on the Raspberry Pi

13.0.1 Running NetRexx in 10 minutes on the Raspberry Linux/ARM system

This install guide is different, in the sense that it describes the entire setup of the Raspberry Pi system, including NetRexx.

Linux on ARM

The Raspberry Pi is an inexpensive computer, containing an ARM architecture CPU on a board the size of a credit card, which sells for $35. It boots from an SD card, the kind you have in your digital camera. In a few small steps you can be up and running with NetRexx. Recent Raspbian distributions already contain Java.

- Use an SD card of suitable size (and known brand)[23], at least 2GB but 8 or 16 is advisable
- Download the raspbian image from `http://www.raspberrypi.org/downloads`
- Hook up an SD Card writer (the one in your digital camera probably also works) to the USB port of your computer
- While taking good care not to overwrite your harddisk, use *dd* or, on Windows, *Win32DiskManager* to write the image to the SD card. This takes a minute. Good instructions are at `http://elinux.org/RPi_Easy_SD_Card_Setup`
- Now unpack the Raspberry Pi, connect the hdmi to a tv or via an hdmi-monitor cable to a monitor, connect a keyboard (mouse can be attached later, if at all), and connect the mini-usb adapter to the power socket. I used a spare plug from an old phone. It boots and gives a lot of Unix messages. The first boot is not very quick. Connect an ethernet cable to your router[24].
- You land in the raspi-config system. Resize the partitions to fill your SD card. Change the password for the pi user, set the default locale, and enable ssh. You can worry with the other options later.
- Note the IP address that the system received from DHCP
- Login from another system, for example using Putty (for Windows) or use ssh pi@your.ip.add.ress (these are the numbers of an IP4 address)
- Use scp or ftp (binary mode) to transmit the NetRexxC.jar to the system, or install the whole NetRexx package. There is an unzip command available
- Set path and classpath as indicated earlier, and run NetRexx. You have the option to develop and compile on the Raspberry, or just upload class files to it.

[23] Not all cards work; the large brands do. SanDisk Ultra SDHC 16Gig cards are verified to work.

[24] The entire installation can be done without connection a monitor if so desired. You can find the Raspberry on your network by using `nmap`, or looking at your router interface. Be sure to re-enable ssh when running raspi-config.

14

Troubleshooting

Can't find class org.netrexx.process.NetRexxC probably means that the NetRexxC.jar file has not been specified in your CLASSPATH setting, or is misspelled, or is in the wrong case, or (for Java 1.2 or later) is not in the Java \lib\ext directory. Note that in the latter case there are two lib directories in the Java tree; the correct one is in the Java Runtime Environment directory (jre). The Setting the CLASSPATH section contains information on setting the CLASSPATH.

+++ Error: The class 'java.lang.Object' cannot be found. You are running with an IBM JVM or JRE. The java.lang.Object class is packaged in the file vm.jar, which needs to be on your CLASSPATH

Can't find class hello may mean that the directory with the hello.class file is not in your CLASSPATH (you may need to add a . (dot) to the CLASSPATH, signifying the current directory), or either the filename or name of the class (in the source) is spelled wrong (the java command is [very] case-sensitive). Note that the name of the class must not include the .class extension.

Exception ... NoClassDefFoundError: sun/tools/javac/Main This indicates that you did not add the Java tools to your CLASSPATH (hence Java could not find the javac compiler). Your system might not have tools.jar: use the -ecj option on the compile command, and use `NetRexxF.jar`.

Error opening the file 'hello.java' [C:\Program Files(86) \javajdk1 7.0.05 jrebinhello.java (Access is denied)] - your userid needs write authorization on the current directory. Please copy the source file to a writeable directory and try again.

Extra blanks You have an extra blank or two in the CLASSPATH. Blanks should only occur in the middle of directory names (and even then, you probably need some double quotes around the SET command or the CLASSPATH segment with the blank). The JVM is sensitive about this.

Permission Denied You are trying the NetRexxC.sh or nrc scripts under Linux or other Unix system, and are getting a Permission denied message. This probably means that you have not marked the scripts as being executable. To do this, use the chmod command, for example: chmod 751 NetRexxC.sh.

No such file You are trying the NetRexxC.sh or nrc scripts under Linux or other Unix system, and are getting a No such file or syntax error message from bash. This probably means that you did not use the unzip -a command to unpack the NetRexx package, so CRLF sequences in the scripts were not converted to LF.

You have only the Java runtime installed, and not the toolkit. If the toolkit is installed, you should have a program called javac on your computer. You can check whether javac is available and working by issuing the javac command at a command prompt; it should respond with usage information.

java.lang.OutOfMemoryError when running the compiler probably means that the maximum heap size is not sufficient. The initial size depends on your Java virtual machine; you can change it to (say) 128 MegaBytes by setting the environment variable:

```
SET NETREXX_JAVA=-Xmx128M
```

The NetRexxC.cmd and .bat files add the value of this environment variable to the options passed to java.exe. If you're not using these, modify your java command or script appropriately.

Down-level Java You have a down-level version of Java installed. Java 1.6, or more recent, is needed for NetRexx 3.06-GA. The level of the JVM can be checked with the command:

```
java -version'
```

applet viewer needed Some of the samples must be viewed using the Java toolkit appletviewer or a Java-enabled browser. Please see the hypertext pages describing these for detailed instructions. In general, if you see a message from Java saying:

```
void main(String argv[]) is not defined
```

this means that the class cannot be run using just the *java* command; it must be run from another Java program, probably as an applet.

15

Current Restrictions

This chapter lists the restrictions for the current release. Please note that the presence of an item in this section is not a commitment to remove a restriction in some future update; NetRexx enhancements are dependent on on-going research, your feedback, and available resources. You should treat this list as a "wish-list" (and please send in your wishes, preferable as an RFE on the `http://kenai.com/projects/netrexx` website).

15.1 General restrictions

1. The translator requires that Java 1.6 or later be installed. Note that Java 8 is the current version, so the chance that you will be impacted by this is minimal.
2. Certain forward references (in particular, references to methods later in a program from the argument list of an earlier method) are not handled by the translator. For these, try reordering the methods.

15.2 Compiler restrictions

The following restrictions are due to the use of a translator for compiling, and would probably only be lifted if a direct-to-bytecodes NetRexx compiler were built. Externally-visible names (property, method, and class names) cannot be Java reserved words (you probably want to avoid these anyway, as people who have to write in Java cannot refer to them), and cannot start with "$0".

1. There are various restrictions on naming and the contents of programs (the first class name must match the program name, etc.), required to meet Java rules.
2. The javac compiler requires that mutually-dependent source files be on the CLASSPATH, so it can find the source files. NetRexxC does not have this restriction, but when using javac for the final compilation you will need to follow the convention described in the Compiling multiple programs and using packages section (see page 23).
3. The symbols option (which requests that debugging information be added to generated .class files) applies to all programs compiled together if any of them specify that option.
4. Some binary floating point underflows may be treated as zero instead of being trapped as errors.
5. When trace is used, side-effects of calls to this() and super() in constructors may be seen before the method and method call instructions are traced – this is because

the Java language does not permit tracing instructions to be added before the call to this() or super().
6. The results of expressions consisting of the single term "null" are not traced.
7. When a minor (inner) class is explicitly imported, its parent class or classes must also be explicitly imported, or javac will report that the class cannot be found.

15.3 Interpreter restrictions

Interpreting Java-based programs is complex, and is constrained by various security issues and the architecture of the Java Virtual Machine. As a result, the following restrictions apply; these will not affect most uses of the interpreter.

1. Certain "built-in" Java classes[25] are constrained by the JVM in that they are assumed to be pre-loaded. An attempt to interpret them is allowed, but will cause the later loading of any other classes to fail with a class cast exception. Interpreted classes have a stub which is loaded by a private class loader. This means that they will usually not be visible to external (non-interpreted) classes which attempt to find them explicitly using reflection, Class.forName(), etc. Instead, these calls may find compiled versions of the classes from the classpath. Therefore, to find the "live" classes being interpreted, use the NetRexxA interpreter API interface (described below).
2. An interpreter cannot completely emulate the actions taken by the Java Virtual Machine as it closes down. Therefore, special rules are followed to determine when an application is assumed to have ended when interpreting (that is, when any of –exec, –arg, or –noarg is specified):
3. If the application being interpreted invokes the exit method of the java.lang.System class, the run ends immediately (even if –prompt was specified). The call cannot be intercepted by the interpreter, and is assumed to be an explicit request by the application to terminate the process and release all resources. In other cases, NetRexxC has to decide when the application ends and hence when to leave NetRexxC (or display the prompt, if –prompt was specified). The following rules apply:
   (a) If any of the programs being interpreted contains the NetRexx exit instruction and the application leaves extra user threads active after the main method ends then NetRexxC will wait for an exit instruction to be executed before assuming the application has ended and exiting (or re-prompting). Otherwise (that is, there are no extra threads, or no exit instruction was seen) the application is assumed to have ended as soon as the main method returns and in this case the run ends (or the prompt is shown) immediately. This rule allows a program such as "hello world" to be run after a windowing application (which leaves threads active) without a deadlocked wait. These rules normally "do the right thing". Applications which create windows may, however, appear to exit prematurely unless they use the NetRexx exit instruction to end their execution, because of the last rule.
   (b) Applications which include both thread creation and an exit instruction which is never executed will wait indefinitely and will need to be interrupted by an

[25] notably java.lang.Object, java.lang.String, and java.lang.Throwable

external "break" request, or equivalent, just as they would if run from compiled classes.

(c) Interpreting programs which set up their own security managers may prevent correct operation of the interpreter.

Index

ISBN 978-90-819090-2-0