NetRexx Quick Beginnings

Mike Cowlishaw and RexxLA

Version 3.01 of April 7, 2012

Temporary Disclaimer

This is not official in any way yet

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 Beginnings Guide	This guide is meant for an audience that has done some programming and wants a quick start. It starts with a quick tour of the language, and a section on installing the NetRexx translator and how to run the reference implementation. It also contains help for troubleshooting if anything in the installation does not work as designed., and states current limits and restrictions of the open source reference implementation.
Programming Guide	The Programming Guide is the one manual that at the same time teaches programming, shows lots of examples as they occur in the real world, and explains about the internals of the translator and how to interface with it.
Language Reference	Referred to as the NRL, This is the formal definition for the language, documenting its syntax and semantics, and prescribing minimal functionality for language implementors. It is the definitive answer to any question on the language, and as such, is subject to approval of the NetRexx Architecture Review Board on any release of the language (including its NRL).

Typographical conventions

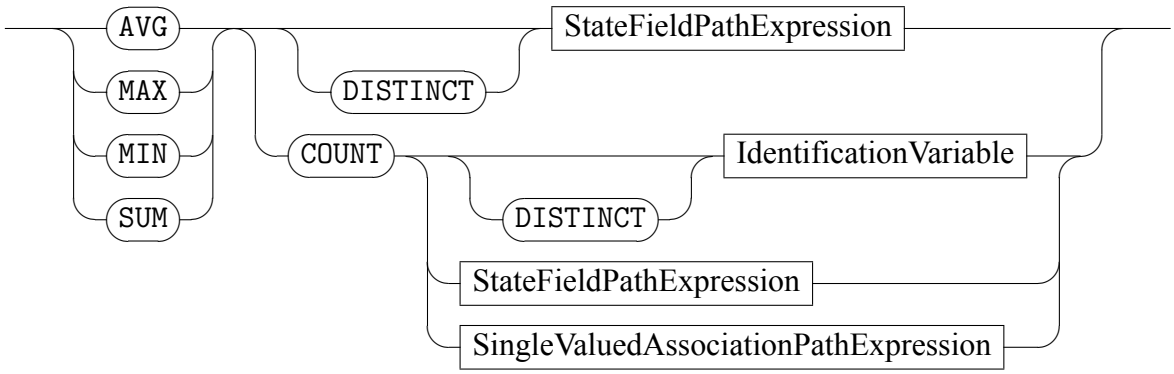
In general, the following conventions have been observed in the NetRexx publications:

- 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
- Item that are introduced, or emphasized, are in an *italic* type
- Included program fragments are listed in this fashion:

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

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

AggregateExpression



Introduction

This document is the *Quick Beginnings* 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 by IBM since 1995 and is Open Source since June, 2011.

In this Quick Beginnings 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 growing 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    /* greet.nrx — 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    /* If no name was entered, then use a fixed         */
6    /* greeting, otherwise echo the name politely.      */
7    if answer='' then say 'Hello Stranger!'
8                  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

```
1    say 'Here we have an expression that is quite long,' -
2        '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

```
1    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

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

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

Listing 1.7: select - when - otherwise

```
1    select
2      when a>0 then say 'greater than zero'
3      when a<0 then say 'less than zero'
4      otherwise say 'zero'
5      end
6    select case i+1
7      when 1 then say 'one'
8      when 1+1 then say 'two'
9      when 3, 4, 5 then say 'many'
10     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

```
0.14285714285714285714285714285714285714285714285714
```

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

```
1     comma=','
2     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

```
1    fred=0 —              initial value
2    fred[3]='abc' —  indexed value
```

or retrieved:

Listing 1.17: Retrieving

```
1    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

```
1    bark='woof'
2    bark['pup']='yap'
3    bark['bulldog']='grrrrr'
4    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

```
1    bark='woof'
2    bark['spaniel', 'brown']='ruff'
3    bark['bulldog']='grrrrr'
4    animal='dog'
5    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

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(newwidth, newheight)—
         when we get here, a new (uninitialized) object—
         has been created.  Copy the parameters we have—
         been given to the properties of the object:
        width=newwidth; height=newheight
      /* Change the size of an Oblong */
      method size(newwidth, newheight) returns Oblong
        width=newwidth; height=newheight
        return this —   return the resized object
      /* Change the size of an Oblong, relatively */
      method relsize(relwidth, relheight)–
                    returns Oblong
        width=width+relwidth; height=height+relheight
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
first.print --                show it
first.relsize(1,1).print --   enlarge and print again
second=charOblong(1,2,'*') -- make another oblong
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

```
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

```
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

```
    trace results
    number=1/7
    parse number before '.' after
    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—
2    now variables a and b will be traced
3   a=3
4   b=4
5   c=5
6   trace var –b c—
7    now variables a and c will be traced
8   a=a+1
9   b=b+1
10  c=c+1
11  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 hard- ware 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.
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.

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

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
say i*j
k=i*j—
 'k' is an 'int', with no initial value—
 multiply and display the result—
 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

```
say 'Please enter a number:'
number=ask
do
  say 'The reciprocal of' number 'is:' 1/number
catch Exception
  say 'Sorry, could not divide "'number'" into 1'
  say 'Please try again.'
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

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, as of version 3.01, will run on Java version 1.5 or later[4]. To do anything more than run NetRexx programs with the runtime package, a Java software development kit is required. 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 "1.6.0_26"
Java(TM) SE Runtime Environment (build 1.6.0_26-b03-383-11A511)
Java HotSpot(TM) 64-Bit Server VM (build 20.1-b02-383, mixed mode)
```

For more information on Java installation:

1. For some operating environments
2. For other operating systems, see the Oracle Java web page[5] – or other suppliers of Java toolkits.

2.1 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.

2.1.1 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. Choose where you want the NetRexx directory tree to reside, and unpack the zip file in the directory which will be the parent of the NetRexx tree. Here are some tips: The syntax for unzipping NetRexx.zip is simply

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

```
unzip NetRexx
```

which should create the files and directory structure directly.

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

  ```
  jar xf NetRexx.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. Note that the extension (.zip) is required.

After unpacking, the following directories1 should have been created: -TODO-

2.2 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 briefly as follows[6] (full details are given later):

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

   •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[7].
2. Copy all the files in the NetRexx/bin directory to a directory in your PATH (perhaps the /bin directory in the Java installation tree). 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.
4. Test the installation by making the /bin directory the current directory and issuing the following two commands exactly as written:

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

   The first of these should translate the test program and then invoke the javac compiler to generate the class file (hello.class) for the program. The second should run the program and display a simple greeting.

[6] For Windows operating system, forward slashes are backslashes.

[7] This has serious drawbacks, however: This breaks NetRexx applications running in custom class loader environments such as jEdit and NetRexxScript, as well as some JSP containers. As soon as the Java version is updates, NetRexx applications may mysteriously – due to the now obsolete path - fail. Running multiple versions of Java and NetRexx for testing purposes will become very hard when this way of installing is chosen.

If you have any problems or errors in the above process, please read the detailed instructions and problem-solving tips that follow.

2.3 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.

2.4 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 there may be more than one way). Please refer to Appendix 1 for your specific platform.

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

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

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

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

  Changes for re-boot or opening of a new window should be placed in your .cshrc file. If you are unsure of how to do this, check the documentation you have for installing the Java toolkit.

2.5 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 Installation Problems section. To test your installation,

make the directory to which you copied the executables the current directory, then (being very careful to get the case of letters correct):

1. Enter the command

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

   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), to compile the file hello.java to make the binary class file hello.class. The intermediate .java file is then deleted, unless an error occurred or you asked for it to be kept.4
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.

3

Installing on an EBCDIC system

The NetRexx binaries are identical for all operating systems; the same NetRexxC.jar runs everywhere [8]. 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) are 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 EBCDIC machine. Specifically:

- The NetRexxC.jar file should be copied as-is, that is, use FTP or other file transfer with the BINARY option. The CLASSPATH should be set to include this NetRexxC.jar file.
- Other files (documentation, etc.) should be copied as Text (that is, they will be translated from ASCII to EBCDIC).

In general, files with extension *.au*, *.class*, *.gif*, *.jar*, or *.zip* are binary files; all others are text files. Setting the classpath might look like this:

```
export CLASSPATH=\$CLASSPATH:/u/j390/j1.1.8/lib/NetRexxC.jar
```

[8] Many thanks to Mark Cathcart and John Kearney for contributing the details to the original version of this section.

4

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 2.5 on page 19.

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.

4.1 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. 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.

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

```
nrc hello d:\textbackslash myprograms\textbackslash 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. [9] 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.

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). 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.

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

-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.

-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* compile, and total times (for the sum of all programs processed).

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

-run run the resulting Java class as a stand-alone application, provided that the compilation had no errors. (See note below.)

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.

4.3 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 4.1: Dependencies

```
1 /* X.nrx */
2 class X
3   why=Y null
4
5 /* Y.nrx */
6 class Y
7   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's a restriction, though, if the classes you are compiling *are* in packages (that is, they include a **package** instruction). Currently, NetRexxC uses the *javac* compiler to generate the *.class* files, and for mutually-dependent files like these, *javac* requires that the source files 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:\textbackslash myproject`

if the two programs above specified a package, thus:

Listing 4.2: 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).

4.4 Compiling from another program

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

Listing 4.3: Invoking NetRexxC.main

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

The *Rexx* string passed to the method can be any combination of program names and options (except *-run*), as described above. Program names may optionally be enclosed

in double-quote characters (and must be if the name includes any blanks in its specification).

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

Listing 4.4: Compiletest

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

Alternatively, the compiler may be called using the method:

Listing 4.5: Calling with Array argument

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

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

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

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

4.5 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 4.6: 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 4.7: 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 4.8: Example of compiling from String

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

5

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.

5.1 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.

5.1.1 Interpreting programs

The NetRexx interpreter is currently designed to be fully compatible with NetRexx programs compiled conventionally. There are some minor restrictions (see section 9 on page 41), 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*[10] or *-nocompile* was specified.
For example, to interpret the hello world program without compilation, the command:

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

[10] 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.

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 *Using the NetRexxA API* in chapter 6 on page 33.

5.2 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. This is discussed further in the

5.3 Interpreting – Performance

The initial release of the interpreter, in the NetRexx 2.0 reference 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 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.

6

Using the NetRexxA API

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

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

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

Listing 6.1: Try the NetRexxA interface

```
1 options binary
2 import org.netrexx.process.NetRexxA
3
4 interpreter=NetRexxA()              -- make interpreter
5
6 files=['hello.nrx']                 -- a file to interpret
```

```
7  flags=['nocrossref', 'verbose0']    -- flags, for example
8  interpreter.parse(files, flags)     -- parse the file(s), using the flags
9
10 helloClass=interpreter.getClassObject(null, 'hello') -- find the hello Class
11
12 -- find the 'main' method; it takes an array of Strings as its argument
13 classes=[interpreter.getClassObject('java.lang', 'String', 1)]
14 mainMethod=helloClass.getMethod('main', classes)
15
16 -- now invoke it, with a null instance (it is static) and an empty String
       array
17 values=[Object String[0]]
18
19 loop for 10     -- let's call it ten times, for fun...
20   mainMethod.invoke(null, values)
21 end
```

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

6.1 The NetRexxA constructor

Listing 6.2: Constructor

```
1 NetRexxA()
```

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

6.2 The parse method

Listing 6.3: parse

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

The parse method takes two arrays of Strings. The first array contains a list of one or more file specifications, one in each element of the array; these specify the files that are to be parsed and made ready for interpretation.
The second array is a list of zero or more option words; these may be any option words understood by the interpreter (but excluding those known only to the NetRexxC command interface, such as *time*). [11] The parse method prefixes the *nojava* flag automatically, to prevent *.java* files being created inadvertently. In the example, *nocrossref* is

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

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

6.3 The getClassObject method

Listing 6.4: getClassObject

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

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

7

Using NetRexx for Web applets

Web applets can be written one of two styles:

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

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

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

```
D:\mydata
```

you might put your applets into

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

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

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

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

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

8

Troubleshooting

1. Can't find class org.netrexx.process.NetRexxC... message 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.
2. Can't find class hello... message may mean that the directory with the hello.class file is not in your CLASSPATH (you may need to add a ?.;' 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.
3. The compiler appears to work, but towards the end fails with Exception ... NoClassDefFoundError: sun/tools/javac/Main. This indicates that you are running Java 1.2 or later but did not add the Java tools to your CLASSPATH (hence Java could not find the javac compiler). See the Installing for Java 1.2+ section for more details, and an alternative action. Alternatively, you may be trying to use NetRexx under Visual J++, which needs a different procedure. You can check whether javac is available and working by issuing the javac command at a command prompt; it should respond with usage instructions.
4. 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.
5. 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.
6. 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.
7. You didn't install on a file system that supports long file names (for example, on OS/2 or Windows you should use an HPFS or FAT32 disk or equivalent). Like most Java applications, NetRexx uses long file names.
8. You have a down-level unzip utility, or changed the name of the NetRexxC.jar file so that it does not match the spelling in the classpath. For example, check that the

name of the file ?NetRexxC.jar' is exactly that, with just three capital letters.

9. 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.
10. An Out of environment space message when trying to set CLASSPATH under Win9x-DOS can be remedied by adding /e:4000 to the ?Cmd line' entry for the MS-DOS prompt properties (try command /? for more information).
11. An exception, apparently in the RexxUtil.translate method, when compiling with Microsoft Java SDK 3.1 (and possibly later SDKs) is caused by a bug in the Just In Time compiler (JIT) in that SDK. Turn off the JIT using Start -> Settings -> Control Panel -> Internet to get to the Internet Properties dialog, then select Advanced, scroll to the Java VM section, and uncheck ?Java JIT compiler enabled'. Alternatively, turn of the JIT by setting the environment variable: SET MSJAVA_ENABLE_JIT=0 (this can be placed in a batch file which invokes NetRexxC, if desired).
12. 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) 24 MegaBytes by setting the environment variable: SET NETREXX_JAVA=-mx24M In Java 1.2.2 or later, use: SET NETREXX_JAVA=-Xmx24M
13. 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.
14. You have a down-level version of Java installed. NetRexxC will run only on Java version 1.1.2 (and later versions). You can check the version of Java you have installed using the command ?java -version'.
15. Included in the documentation collection are a number of examples and samples (Hello, HelloApplet, etc.). To run any of these, you must have Java installed.
16. Further, some of the samples must be viewed using the Java toolkit applet-viewer 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.

9

Current Restrictions

The NetRexx translator is now functionally complete, though work continues on usability and performance improvements. As of this version there are still a number of restrictions, listed below. 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).

9.1 General restrictions

1. The translator requires that Java 1.1.2 or later be installed. To use the interpreter functions, at least Java 1.2 (Java 2) is required. Note that Java 6 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.

9.2 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.
8. If you have a loop construct with a large number (perhaps hundreds) of instructions inside it, running the compiled class may fail with an illegal target of jump or branch verification error (or, under Java 1.1, simply terminate execution after one iteration of the loop). This is due to a bug in javac one workaround is to move some of the code out of the loop, perhaps into a private method. (The following problem may occur in larger methods, with Java 1.1.2; it seems to have been fixed in later versions of Java): NetRexxC does not restrict the number of local variables used or generated. However, the 1.1.2 javac compiler fails with unrelated error messages (such as statement unreachable or variable may be uninitialized) if asked to handle more than 63 local variables.

9.3 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. For interpretation to proceed, when any of –exec, –arg, or –noarg is specified, you must be running a Java 2 JVM (Java Virtual Machine). That is, the command “java –version” should report a version of 1.2 or later. Parsing and compilation, however, only require Java 1.1.2.
2. Certain “built-in” Java classes (notably java.lang.Object, java.lang.String, and java.lang.Throwable) 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).
3. 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): • 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 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.

List of Figures

List of Tables

Index