NetRexx Language Overview

23rd August 2000

Mike Cowlishaw

mfc@uk.ibm.com
IBM UK Laboratories

Version 2.00

Table of Contents

NetRexx Overview

This document summarizes the main features of NetRexx, and is intended to help you start using it quickly. It's assumed that you have some knowledge of programming in a language such as Rexx, C, BASIC, or Java, but a knowledge of "object-oriented" programming isn't 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 Language Definition* and the *NetRexx Language Supplement*.

NetRexx programs

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

```
/* This wishes you the best of health. */
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 COM.ibm.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.

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:

```
var1=5            /* sets var1 to '5'    */
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:

```
/* greet.nrx -- a short program to greet you.        */
/* First display a prompt:                           */
say 'Please type your name and then press Enter:'
answer=ask            /* Get the reply into 'answer' */

/* If no name was entered, then use a fixed          */
/* greeting, otherwise echo the name politely.       */
if answer='' then say 'Hello Stranger!'
             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 oper-*

ator 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:

```
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 semi-colon separator:

```
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).

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:

```
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:

```
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:

```
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:

```
/* 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 page 20) 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.

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:

```
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 18) instead of NetRexx decimal arithmetic for calculations, if it can, throughout the program.

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:

```
phrase='Now is the time for a party'
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:

```
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.

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.

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:

```
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.`".

Literal patterns

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

```
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:

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

therefore have the same effect as the previous example.

Positional patterns

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

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:

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

or retrieved:

```
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:

```
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:

```
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:

```
/* 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”.

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:

```
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:

```
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:

```
say words[1]
```

would then display `Ffynnon`.

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):

```
/* 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:

```
/* 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.

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:

```
/* 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.

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:

```
/* 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:

```
/* 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.

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:

```
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:

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

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

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`":

```
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 un-named 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 *-*                    returns Oblong
    >a> relwidth "1"
    >a> relheight "1"
 26 *=*     method print
Oblong 6 x 4
 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's 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`":

```
trace var a b
-- now variables a and b will be traced
a=3
b=4
c=5
trace var –b c
-- now variables a and c will be traced
a=a+1
b=b+1
c=c+1
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
```


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[1] 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.

---

[1] 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.

Explicit type assignment

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

```
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.

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:

```
/* A simple graphics Applet */
class Rainbow extends Applet
  method paint(g=Graphics)  -- called to repaint window
    maxx=size.width-1
    maxy=size.height-1
    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.

Exception and error handling

NetRexx doesn't 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:

```
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.

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://www2.hursley.ibm.com/netrexx/
```

Also at that location, you'll find the NetRexx language specification and other information, and downloadable packages containing the NetRexx software and documentation. The software should run on any platform that supports the Java Development Kit, version 1.1.2 or later.