Copyright © Mike Cowlishaw 1979, 2009.
All rights reserved.
# Table of Contents

**Introduction** 7
- Language objectives 7
- Language concepts 10
- Acknowledgements 15

**NetRexx Overview** 17
- NetRexx programs 18
- Expressions and variables 19
- Control instructions 21
- NetRexx arithmetic 22
- Doing things with strings 23
- Parsing strings 24
- Indexed strings 25
- Arrays 26
- Things that aren’t strings 27
- Extending classes 29
- Tracing 31
- Binary types and conversions 33
- Exception and error handling 35

**NetRexx Language Definition** 37
- Notations 38
- Characters and Encodings 39
- Structure and General Syntax 40
  - Blanks and White Space 40
  - Comments 40
  - Tokens 41
  - Implied semicolons and continuations 44
  - The case of names and symbols 44
  - Hexadecimal and binary numeric symbols 45
- Types and Classes 46
- Terms 48
  - Evaluation of terms 49
- Methods and Constructors 53
  - Method call instructions 53
  - Method resolution (search order) 54
  - Method overriding 55
  - Constructor methods 56

**Version 2.00** 3
Appendix A – A Sample NetRexx Program 171

Appendix B – JavaBean Support 175
Indirect properties 176

Appendix C – The netrexx.lang Package 179
Exception classes 180
The Rexx class 181
Rexx constructors 181
Rexx arithmetic methods 182
Rexx miscellaneous methods 184
The RexxOperators interface class 185
The RexxSet class 186
Public properties 186
Constructors 186
Methods 187

Index 189
Introduction

NetRexx is a general-purpose programming language inspired by two very different programming languages, Rexx and Java™. It is designed for people, not computers. In this respect it follows Rexx closely, with many of the concepts and most of the syntax taken directly from Rexx or its object-oriented version, Object Rexx. From Java it derives static typing, binary arithmetic, the object model, and exception handling. The resulting language not only provides the scripting capabilities and decimal arithmetic of Rexx, but also seamlessly extends to large application development with fast binary arithmetic.

The reference implementation of NetRexx produces classes for the Java Virtual Machine, and in so doing demonstrates the value of that concrete interface between language and machine: NetRexx classes and Java classes are entirely equivalent – NetRexx can use any Java class (and vice versa) and inherits the portability and robustness of the Java environment.

This document is in three parts:
1. The objectives of the NetRexx language and the concepts underlying its design, and acknowledgements.
2. An overview and introduction to the NetRexx language.
3. The definition of the language.

Appendices include a sample NetRexx program, a description of an experimental feature, and some details of the contents of the netrexx.lang package.

Language objectives

This document describes a programming language, called NetRexx, which is derived from both Rexx and Java. NetRexx is intended as a dialect of Rexx that can be as efficient and portable as languages such as Java, while preserving the low threshold to learning and the ease of use of the original Rexx language.

Features of Rexx

The Rexx programming language¹ was designed with just one objective: to make programming easier than it was before. The design achieved this by emphasizing readability and usability, with a minimum of special notations and restrictions. It was consciously designed to make life easier for its users, rather than for its implementers.

One important feature of Rexx syntax is keyword safety. Programming languages invariably need to

evolve over time as the needs and expectations of their users change, so this is an essential requirement for languages that are intended to be executed from source.

Keywords in Rexx are not globally reserved but are recognized only in context. This language attribute has allowed the language to be extended substantially over the years without invalidating existing programs. Even so, some areas of Rexx have proved difficult to extend – for example, keywords are reserved within instructions such as `do`. Therefore, the design for NetRexx takes the concept of keyword safety even further than in Rexx, and also improves extensibility in other areas.

The great strengths of Rexx are its human-oriented features, including

- simplicity
- coherent and uncluttered syntax
- comprehensive string handling
- case-insensitivity
- arbitrary precision decimal arithmetic.

Care has been taken to preserve these. Conversely, its interpretive nature has always entailed a lack of efficiency: excellent Rexx compilers do exist, from IBM and other companies, but cannot offer the full speed of statically-scoped languages such as C\(^2\) or Java.\(^3\)

**Influence of Java**

The system-independent design of Rexx makes it an obvious and natural fit to a system-independent execution environment such as that provided by the Java Virtual Machine (JVM). The JVM, especially when enhanced with “just-in-time” bytecode compilers that compile bytecodes into native code just before execution, offers an effective and attractive target environment for a language like Rexx.

Choosing the JVM as a target environment does, however, place significant constraints on the design of a language suitable for that environment. For example, the semantics of method invocation are in several ways determined by the environment rather than by the source language, and, to a large extent, the object model (class structure, etc.) of the Java environment is imposed on languages that use it.

Also, Java maintains the C concept of primitive datatypes; types (such as `int`, a 32-bit signed integer) which allow efficient use of the underlying hardware yet do not describe true objects. These types are pervasive in classes and interfaces written in the Java language; any language intending to use Java classes effectively must provide access to these types.

Equally, the *exception* (error handling) model of Java is pervasive, to the extent that methods must check certain exceptions and declare those that are not handled within the method. This makes it difficult to fit an alternative exception model.

The constraints of safety, efficiency, and environment necessitated that NetRexx would have to differ in some details of syntax and semantics from Rexx; unlike Object Rexx, it could not be a fully upwards-compatible extension of the language.\(^4\) The need for changes, however, offered the opportunity to make some significant simplifications and enhancements to the language, both to

---


improve its keyword safety and to strengthen other features of the original Rexx design.⁵ Some additions from Object Rexx and ANSI Rexx ⁶ are also included.

Similarly, the concepts and philosophy of the Rexx design can profitably be applied to avoid many of the minor irregularities that characterize the C and Java language family, by providing suitable simplifications in the programming model. For example, the NetRexx looping construct has only one form, rather than three, and exception handling can be applied to all blocks rather than requiring an extra construct. Also, as in Rexx, all NetRexx storage allocation and de-allocation is implicit – an explicit new operator is not required.

Further, the human-oriented design features of Rexx (case-insensitivity for identifiers, type deduction from context, automatic conversions where safe, tracing, and a strong emphasis on string representations of common values and numbers) make programming for the Java environment especially easy in NetRexx.

**A hybrid or a whole?**

As in other mixtures, not all blends are a success; when first designing NetRexx, it was not at all obvious whether the new language would be an improvement on its parents, or would simply reflect the worst features of both.

The fulcrum of the design is perhaps the way in which datatyping is automated without losing the static typing supported by Java. Typing in NetRexx is most apparent at interfaces – where it provides most value – but within methods it is subservient and does not obscure algorithms. A simple concept, binary classes, also lets the programmer choose between robust decimal arithmetic and less safe (but faster) binary arithmetic for advanced programming where performance is a primary consideration.

The “seamless” integration of types into what was previously an essentially typeless language does seem to have been a success, offering the advantages of strong typing while preserving the ease of use and speed of development that Rexx programmers have enjoyed.

The end result of adding Java typing capabilities to the Rexx language is a single language that has both the Rexx strengths for scripting and for writing macros for applications and the Java strengths of robustness, good efficiency, portability, and security for application development.

---


Language concepts

As described in the last section, NetRexx was created by applying the philosophy of the Rexx language to the semantics required for programming the Java Virtual Machine (JVM). Despite the assumption that the JVM is a “target environment” for NetRexx, it is intended that the language not be environment-dependent; the essentials of the language do not depend on the JVM. Environment-dependent details, such as the primitive types supported, are not part of the language specification.

The primary concepts of Rexx have been described before, in *The Rexx Language*, but it is worth repeating them and also indicating where modifications and additions have been necessary to support the concepts of statically-typed and object-oriented environments. The remainder of this section is therefore a summary of the principal concepts of NetRexx.

Readability

One concept was central to the evolution of Rexx syntax, and hence NetRexx syntax: readability (used here in the sense of perceived legibility). Readability in this sense is a somewhat subjective quality, but the general principle followed is that the tokens which form a program can be written much as one might write them in Western European languages (English, French, and so forth). Although NetRexx is more formal than a natural language, its syntax is lexically similar to everyday text.

The structure of the syntax means that the language is readily adapted to a variety of programming styles and layouts. This helps satisfy user preferences and allows a lexical familiarity that also increases readability. Good readability leads to enhanced understandability, thus yielding fewer errors both while writing a program and while reading it for information, debugging, or maintenance. Important factors here are:

1. Punctuation and other special notations are required only when absolutely necessary to remove ambiguity (though punctuation may often be added according to personal preference, so long as it is syntactically correct). Where notations are used, they follow established conventions.
2. The language is essentially case-insensitive. A NetRexx programmer may choose a style of use of uppercase and lowercase letters that he or she finds most helpful (rather than a style chosen by some other programmer).
3. The classical constructs of structured and object-oriented programming are available in NetRexx, and can undoubtedly lead to programs that are easier to read than they might otherwise be. The simplicity and small number of constructs also make NetRexx an excellent language for teaching the concepts of good structure.
4. Loose binding between the physical lines in a program and the syntax of the language ensures that even though programs are affected by line ends, they are not irrevocably so. A clause may be spread over several lines or put on just one line; this flexibility helps a programmer lay out the program in the style felt to be most readable.

Natural data typing and decimal arithmetic

“Strong typing”, in which the values that a variable may take are tightly constrained, has been an attribute of some languages for many years. The greatest advantage of strong typing is for the interfaces between program modules, where errors are easy to introduce and difficult to catch. Errors within modules that would be detected by strong typing (and which would not be detected from context) are much rarer, certainly when compared with design errors, and in the majority of cases do
not justify the added program complexity.

NetRexx, therefore, treats types as unobtrusively as possible, with a simple syntax for type description which makes it easy to make types explicit at interfaces (for example, when describing the arguments to methods).

By default, common values (identifiers, numbers, and so on) are described in the form of the symbolic notation (strings of characters) that a user would normally write to represent those values. This natural datatype for values also supports decimal arithmetic for numbers, so, from the user’s perspective, numbers look like and are manipulated as strings, just as they would be in everyday use on paper.

When dealing with values in this way, no internal or machine representation of characters or numbers is exposed in the language, and so the need for many data types is reduced. There are, for example, no fundamentally different concepts of `integer` and `real`; there is just the single concept of `number`. The results of all operations have a defined symbolic representation, and will therefore act consistently and predictably for every correct implementation.

This concept also underlies the BASIC language; indeed, Kemeny and Kurtz’s vision for BASIC included many of the fundamental principles that inspired Rexx. For example, Thomas E. Kurtz wrote:

> “Regarding variable types, we felt that a distinction between ‘fixed’ and ‘floating’ was less justified in 1964 than earlier ... to our potential audience the distinction between an integer number and a non-integer number would seem esoteric. A number is a number is a number.”

For Rexx, intended as a scripting language, this approach was ideal; symbolic operations were all that were necessary.

For NetRexx, however, it is recognized that for some applications it is necessary to take full advantage of the performance of the underlying environment, and so the language allows for the use and specification of binary arithmetic and types, if available. A very simple mechanism (declaring a class or method to be `binary`) is provided to indicate to the language processor that binary arithmetic and types are to be used where applicable. In this case, as in other languages, extra care has to be taken by the programmer to avoid exceeding limits of number size and so on.

**Emphasis on symbolic manipulation**

Many values that NetRexx manipulates are (from the user’s point of view, at least) in the form of strings of characters. Productivity is greatly enhanced if these strings can be handled as easily as manipulating words on a page or in a text editor. NetRexx therefore has a rich set of character manipulation operators and methods, which operate on values of type `Rexx` (the name of the class of NetRexx strings).

Concatenation, the most common string operation, is treated specially in NetRexx. In addition to a conventional concatenate operator (“|”), the novel `blank operator` from Rexx concatenates two data strings together with a blank in between. Furthermore, if two syntactically distinct terms (such as a string and a variable name) are abutted, then the data strings are concatenated directly. These operators make it especially easy to build up complex character strings, and may at any time be combined with the other operators.

---

For example, the `say` instruction consists of the keyword `say` followed by any expression. In this instance of the instruction, if the variable `n` has the value “6” then

```netrexx
say 'Sorry,' n*100/50,'% were rejected'
```

would display the string

```
Sorry, 12% were rejected
```

Concatenation has a lower priority than the arithmetic operators. The order of evaluation of the expression is therefore first the multiplication, then the division, then the concatenate-with-blank, and finally the direct concatenation.

Since the concatenation operators are distinct from the arithmetic operators, very natural coercion (automatic conversion) between numbers and character strings is possible. Further, explicit type-casting (conversion of types) is effected by the same operators, at the same priority, making for a very natural and consistent syntax for changing the types of results. For example,

```netrexx
i=int 100/7
```

would calculate the result of 100 divided by 7, convert that result to an integer (assuming `int` describes an integer type) and then assign it to the variable `i`.

**Nothing to declare**

Consistent with the philosophy of simplicity, NetRexx does not require that variables within methods be declared before use. Only the `properties` of classes – which may form part of their interface to other classes – need be listed formally.

Within methods, the type of variables is deduced statically from context, which saves the programmer the menial task of stating the type explicitly. Of course, if preferred, variables may be listed and assigned a type at the start of each method.

**Environment independence**

The core NetRexx language is independent of both operating systems and hardware. NetRexx programs, though, must be able to interact with their environment, which implies some dependence on that environment (for example, binary representations of numbers may be required). Certain areas of the language are therefore described as being defined by the environment.

Where environment-independence is defined, however, there may be a loss of efficiency – though this can usually be justified in view of the simplicity and portability gained.

As an example, character string comparison in NetRexx is normally independent of case and of leading and trailing blanks. (The string “Yes” means the same as “yes” in most applications.) However, the influence of underlying hardware has often subtly affected this kind of design decision, so that many languages only allow trailing blanks but not leading blanks, and insist on exact case matching. By contrast, NetRexx provides the human-oriented relaxed comparison for strings as default, with optional “strict comparison” operators.

**Limited span syntactic units**

The fundamental unit of syntax in the NetRexx language is the clause, which is a piece of program text terminated by a semicolon (usually implied by the end of a line). The span of syntactic units is

---

9 Class variables and instance variables.
therefore small, usually one line or less. This means that the syntax parser in the language processor can rapidly detect and locate errors, which in turn means that error messages can be both precise and concise.

It is difficult to provide good diagnostics for languages (such as Pascal and its derivatives) that have large fundamental syntactic units. For these languages, a small error can often have a major or distributed effect on the parser, which can lead to multiple error messages or even misleading error messages.

**Dealing with reality**

A computer language is a tool for use by real people to do real work. Any tool must, above all, be reliable. In the case of a language this means that it should do what the user expects. User expectations are generally based on prior experience, including the use of various programming and natural languages, and on the human ability to abstract and generalize.

It is difficult to define exactly how to meet user expectations, but it helps to ask the question “Could there be a high astonishment factor associated with this feature?” If a feature, accidentally misused, gives apparently unpredictable results, then it has a high astonishment factor and is therefore undesirable.

Another important attribute of a reliable software tool is consistency. A consistent language is by definition predictable and is often elegant. The danger here is to assume that because a rule is consistent and easily described, it is therefore simple to understand. Unfortunately, some of the most elegant rules can lead to effects that are completely alien to the intuition and expectations of a user who, after all, is human.

These constraints make programming language design more of an art than a science, if the usability of the language is a primary goal. The problems are further compounded for NetRexx because the language is suitable for both scripting (where rapid development and ease of use are paramount) and for application development (where some programmers prefer extensive checking and redundant coding). These conflicting goals are balanced in NetRexx by providing automatic handling of many tasks (such as conversions between different representations of strings and numbers) yet allowing for “strict” options which, for example, may require that all types be explicit, identifiers be identical in case as well as spelling, and so on.

**Be adaptable**

Wherever possible NetRexx allows for the extension of instructions and other language constructs, building on the experience gained with Rexx. For example, there is a useful set of common characters available for future use, since only small set is used for the few special notations in the language.

Similarly, the rules for keyword recognition allow instructions to be added whenever required without compromising the integrity of existing programs. There are no reserved keywords in NetRexx; variable names chosen by a programmer always take precedence over recognition of keywords. This ensures that NetRexx programs may safely be executed, from source, at a time or place remote from their original writing – even if in the meantime new keywords have been added to the language.

A language needs to be adaptable because it certainly will be used for applications not foreseen by the designer. Like all programming languages, NetRexx may (indeed, probably will) prove inadequate for certain future applications; room for expansion and change is included to make the language more adaptable and robust.
**Keep the language small**

NetRexx is designed as a small language. It is not the sum of all the features of Rexx and of Java; rather, unnecessary features have been omitted. The intention has been to keep the language as small as possible, so that users can rapidly grasp most of the language. This means that:

- the language appears less formidable to the new user
- documentation is smaller and simpler
- the experienced user can be aware of all the abilities of the language, and so has the whole tool at his or her disposal
- there are few exceptions, special cases, or rarely used embellishments
- the language is easier to implement.

Many languages have accreted “neat” features which make certain algorithms easier to express; analysis shows that many of these are rarely used. As a rough rule-of-thumb, features that simply provided alternative ways of writing code were added to Rexx and NetRexx only if they were likely to be used more often than once in five thousand clauses.

**No defined size or shape limits**

The language does not define limits on the size or shape of any of its tokens or data (although there may be implementation restrictions). It does, however, define a few minimum requirements that must be satisfied by an implementation. Wherever an implementation restriction has to be applied, it is recommended that it should be of such a magnitude that few (if any) users will be affected.

Where arbitrary implementation limits are necessary, the language requires that the implementer use familiar and memorable decimal values for the limits. For example 250 would be used in preference to 255, 500 to 512, and so on.
Acknowledgements

Much of NetRexx is based on earlier work, and I am indebted to the hundreds of people who contributed to the development of Rexx, Object Rexx, and Java.

In the 1990s I gained many insights from the deliberations of the members of the X3J18 technical committee, which, under the remarkable chairmanship of Brian Marks, led to the 1996 ANSI Standard for Rexx. Many of the committee’s suggestions are incorporated in NetRexx.

Equally important have been the comments and feedback from the pioneering users of NetRexx, and all those people who sent me comments on the language either directly or in the NetRexx mailing list or forum. I would especially like to thank Ian Brackenbury, Barry Feigenbaum, Davis Foulger, Norio Furukawa, Dion Gillard, Martin Lafai, Max Marsiglietti, and Trevor Turton for their insightful comments and encouragement.

I also thank IBM; my appointment as an IBM Fellow made it possible to make the implementation of NetRexx a reality in months rather than years.

Finally, this document has relied on old but trusted technology for its creation: its GML markup was processed using macros originally written by Bob O’Hara, and formatted using SCRIPT/VS, the IBM Document Composition Facility. Geoff Bartlett provided critical advice on character sets and fonts for the NetRexx book. This version, for NetRexx 2, uses a set of Rexx programs that convert that same GML markup into OpenOffice Document Text format (XML files).

Mike Cowlishaw, 1997 and 2009
NetRexx Overview

This part of the document summarizes the main features of NetRexx, and is intended to help you start using the language quickly. It is 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 is not needed.

This is not a complete tutorial, however – 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 third part of this document, the *NetRexx Language Definition* (see page 37).
NetRexx programs

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

```plaintext
/* 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:

```plaintext
java COM.ibm.netrexx.process.NetRexxC toast
```

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

```plaintext
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):

```plaintext
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 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:

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

This acts as though the two lines were all on one line, with the hyphen and any blanks around it being replaced by a single blank. The net result is two strings concatenated together (with a blank in between) and then displayed.

When desired, multiple instructions can be placed on one line with the aid of the semicolon separator:

    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:

```plaintext
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:

```plaintext
select
   when a>0 then say 'greater than zero'
   when a<0 then say 'less than zero'
   otherwise say 'zero'
end
```

```plaintext
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:

```plaintext
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:

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

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 35) 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:

```netrex
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:

```netrex
options binary
```

at the top of a NetRexx program. The language processor will then use binary arithmetic (see page 33) 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:

```netrexx
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:

```netrexx
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=','
p 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 */
    /parse wordlist\\"' /* split WORDLIST into first word and residue */
    hadword[word]=1 /* remember we have this word */
    outlist=wordlist ++ /* 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”. 

Version 2.00 NetRexx Overview 25
**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:

```netrexx
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:

```netrexx
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:

```netrexx
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):

```netrexx
/* 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
  return this

  /* Change the size of an Oblong */
  method size(newwidth, newheight) returns Oblong
    width=newwidth; height=newheight
    return this

  /* 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
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:

```plaintext
/* 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:

```plaintext
/* 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 over-complicated.

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:

```netrexx
/* 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
  end

  /* 'Print' the oblong */
  method print
    loop for super.height
      say printchar.copies(super.width)
    end
  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:

```plaintext
/* trycharOblong.nrx -- try the charOblong class */

first=charOblong(5,3,'#')  -- make an oblong
first.print                -- show it
first-resize(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:

```plaintext
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:

```plaintext
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 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.

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:

---

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

```rexx
/* 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:

```netrexx
say 'Please enter a number:'
number=ask
do
  say 'The reciprocal of' number 'is:' 1/number
catch RuntimeException
  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.
NetRexx Language Definition

This part of the document describes the NetRexx language, version 2.00. Version 2 includes the original NetRexx language\textsuperscript{11} together with additions made from 1997 through 2000 and previously published in the *NetRexx Language Supplement*.

The language is described first in terms of the characters from which it is composed and its low-level syntax, and then progressively through more complex constructions. Finally, special sections describe the semantics of the more complicated areas.

Some features of the language, such as *options* keywords and binary arithmetic, are implementation-dependent. Rather than leaving these important aspects entirely abstract, this description includes summaries of the treatment of such items in the *reference implementation* of NetRexx. The reference implementation is based on the Java environment and class libraries.

*Paragraphs that refer to the reference implementation, and are therefore not strictly part of the language definition, are shown in italics, like this one.*

**Notations**

In this part of the document, various notations such as changes of font are used for clarity. Within the text, a sans-serif bold font is used to indicate **keywords**, and an italic font is used to indicate **technical terms**. An italic font is also used to indicate a reference to a **technical term defined elsewhere** or a **word** in a syntax diagram that names a segment of syntax.

Similarly, in the syntax diagrams in this document, words (symbols) in the sans-serif bold font also denote keywords or sub-keywords, and words (such as *expression*) in italics denote a token or collection of tokens defined elsewhere. The brackets [ and ] delimit optional (and possibly alternative) parts of the instructions, whereas the braces { and } indicate that one of a number of alternatives must be selected. An ellipsis ( . . . ) following a bracket indicates that the bracketed part of the clause may optionally be repeated.

Occasionally in syntax diagrams (e.g., for indexed references) brackets are “real” (that is, a bracket is required in the syntax; it is not marking an optional part). These brackets are enclosed in single quotes, thus: ’[ or ’].

Note that the keywords and sub-keywords in the syntax diagrams are not case-sensitive: the symbols “IF” “If” and “iF” would all match the keyword shown in a syntax diagram as *if*.

Note also that most of the clause delimiters (“; ”) shown can usually be omitted as they will be implied by the end of a line.
Characters and Encodings

In the definition of a programming language it is important to emphasize the distinction between a character and the coded representation (encoding) of a character. The character “A”, for example, is the first letter of the English (Roman) alphabet, and this meaning is independent of any specific coded representation of that character. Different coded character sets (such as, for example, the ASCII and EBCDIC codes) use quite different encodings for this character (decimal values 65 and 193, respectively).

Except where stated otherwise, this document uses characters to convey meaning and not to imply a specific character code (the exceptions are certain operations that specifically convert between characters and their representations). At no time is NetRexx concerned with the glyph (actual appearance) of a character.

Character Sets

Programming in the NetRexx language can be considered to involve the use of two character sets. The first is used for expressing the NetRexx program itself, and is the relatively small set of characters described in the next section. The second character set is the set of characters that can be used as character data by a particular implementation of a NetRexx language processor. This character set may be limited in size (sometimes to a limit of 256 different characters, which have a convenient 8-bit representation), or it may be much larger. The Unicode character set, for example, allows for 65536 characters, each encoded in 16 bits.

Usually, most or all of the characters in the second (data) character set are also allowed within a NetRexx program, but only within commentary or immediate (literal) data.

The NetRexx language explicitly defines the first character set, in order that programs will be portable and understandable; at the same time it avoids restrictions due to the language itself on the character set used for data. However, where the language itself manipulates or inspects the data (as when carrying out arithmetic operations), there may be requirements on the data character set (for example, numbers can only be expressed if there are digit characters in the set).

12 These terms have the meanings as defined by the International Organization for Standardization, in ISO 2382 Data processing – Vocabulary.
13 American Standard Code for Information Interchange.
14 Extended Binary Coded Decimal Interchange Code.
Structure and General Syntax

A NetRexx program is built up out of a series of clauses that are composed of: zero or more blanks (which are ignored); a sequence of tokens (described in this section); zero or more blanks (again ignored); and the delimiter “;” (semicolon) which may be implied by line-ends or certain keywords. Conceptually, each clause is scanned from left to right before execution and the tokens composing it are resolved.

Identifiers (known as symbols) and numbers are recognized at this stage, comments (described below) are removed, and multiple blanks (except within literal strings) are reduced to single blanks. Blanks adjacent to operator characters (see page 43) and special characters (see page 43) are also removed.

Blanks and White Space

Blanks (spaces) may be freely used in a program to improve appearance and layout, and most are ignored. Blanks, however, are usually significant

- within literal strings (see below)
- between two tokens that are not special characters (for example, between two symbols or keywords)
- between the two characters forming a comment delimiter
- immediately outside parentheses (“(” and “)” ) or brackets (“[” and “] ”).

For implementations that support tabulation (tab) and form feed characters, these characters outside of literal strings are treated as if they were a single blank; similarly, if the last character in a NetRexx program is the End-of-file character (EOF, encoded in ASCII as decimal 26), that character is ignored.

Comments

Commentary is included in a NetRexx program by means of comments. Two forms of comment notation are provided: line comments are ended by the end of the line on which they start, and block comments are typically used for more extensive commentary.

Line comments

A line comment is started by a sequence of two adjacent hyphens (“--”); all characters following that sequence up to the end of the line are then ignored by the NetRexx language processor.

Example:

i=j+7  -- this line comment follows an assignment

Block comments

A block comment is started by the sequence of characters “/*”, and is ended by the same sequence reversed, “*/”. Within these delimiters any characters are allowed (including quotes, which need not be paired). Block comments may be nested, which is to say that “/*” and “*/” must pair correctly. Block comments may be anywhere, and may be of any length. When a block comment is found, it is treated as though it were a blank (which may then be removed, if adjacent to a special character).

Example:

/* This is a valid block comment */
The two characters forming a comment delimiter ("/ *" or "* /") must be adjacent (that is, they may not be separated by blanks or a line-end).

**Note:** It is recommended that NetRexx programs start with a block comment that describes the program. Not only is this good programming practice, but some implementations may use this to distinguish NetRexx programs from other languages.

**Implementation minimum:** Implementations should support nested block comments to a depth of at least 10. The length of a comment should not be restricted, in that it should be possible to “comment out” an entire program.

**Tokens**

The essential components of clauses are called *tokens*. These may be of any length, unless limited by implementation restrictions,\(^{16}\) and are separated by blanks, comments, ends of lines, or by the nature of the tokens themselves.

The tokens are:

| **Literal strings** | A sequence including any characters that can safely be represented in an implementation\(^{17}\) and delimited by the single quote character (’) or the double-quote ("). Use "*" to include a " in a literal string delimited by *, and similarly use two single quotes to include a single quote in a literal string delimited by single quotes. A literal string is a constant and its contents will never be modified by NetRexx. Literal strings must be complete on a single line (this means that unmatched quotes may be detected on the line that they occur).

Any string with no characters (i.e., a string of length 0) is called a *null string*.

**Examples:**

'Fred'
'Aý'
"Don't Panic!"
".x"
'You shouldn''t' /* Same as "You shouldn't" */
' ' /* A null string */

Within literal strings, characters that cannot safely or easily be represented (for example “control characters”) may be introduced using an *escape sequence*. An escape sequence starts with a backslash ("\"), which must then be followed immediately by one of the following (letters may be in either uppercase or lowercase):

| \( t \) | the escape sequence represents a tabulation (tab) character |
| \( n \) | the escape sequence represents a new-line (line feed) character |
| \( r \) | the escape sequence represents a return (carriage return) character |
| \( f \) | the escape sequence represents a form-feed character |
| \( " \) | the escape sequence represents a double-quote character |

\(^{16}\) Wherever arbitrary implementation restrictions are applied, the size of the restriction should be a number that is readily memorable in the decimal system; that is, one of 1, 25, or 5 multiplied by a power of ten. 500 is preferred to 512, the number 250 is more “natural” than 256, and so on. Limits expressed in digits should be a multiple of three.

\(^{17}\) Some implementations may not allow certain “control characters” in literal strings. These characters may be included in literal strings by using one of the escape sequences provided.
the escape sequence represents a single-quote character
the escape sequence represents a backslash character
the escape sequence represents a “null” character (the character whose encoding equals zero), used to indicate continuation in a say instruction
(zero) the escape sequence represents a “null” character (the character whose encoding equals zero); an alternative to \-
the escape sequence represents a character whose encoding is given by the two hexadecimal digits (“hh”) following the “x”. If the character encoding for the implementation requires more than two hexadecimal digits, they are padded with zero digits on the left.
the escape sequence represents a character whose encoding is given by the four hexadecimal digits (“hhhh”) following the “u”. It is an error to use this escape if the character encoding for the implementation requires fewer than four hexadecimal digits.
Hexadecimal digits for use in the escape sequences above may be any decimal digit (0–9) or any of the first six alphabetic characters (a–f), in either lowercase or uppercase.

Examples:
'You shouldn\'t' /* Same as "You shouldn't" */
'\x6d\u0066\x63' /* In Unicode: 'mfc' */
'\\\u005C' /* In Unicode, two backslashes */

Implementation minimum: Implementations should support literal strings of at least 100 characters. (But note that the length of string expression results, etc., should have a much larger minimum, normally only limited by the amount of storage available.)

Symbols
Symbols are groups of characters selected from the Roman alphabet in uppercase or lowercase (A–Z, a–z), the Arabic numerals (0–9), or the characters underscore, dollar, and euro (“$€”). Implementations may also allow other alphabetic and numeric characters in symbols to improve the readability of programs in languages other than English. These additional characters are known as extra letters and extra digits.19

Examples:
DanYrOgof
minx
Élan
$Virtual3D

A symbol may include other characters only when the first character of the symbol is a digit (0–9 or an extra digit). In this case, it is a numeric symbol – it may include a period (“.”) and it must have the syntax of a number. This may be simple number, which is a sequence of digits with at most one period (which may not be the final character of the sequence), or it may be a hexadecimal or binary numeric symbol (see page 45), or it may be a number expressed in exponential notation.

A number in exponential notation is a simple number followed immediately by the

18 Note that only UTF8-encoded source files can currently use the euro character.
19 It is expected that implementations of NetRexx will be based on Unicode or a similarly rich character set. However, portability to implementations with smaller character sets may be compromised when extra letters or extra digits are used in a program.
sequence “E” (or “e”), followed immediately by a sign (“+” or “−”), followed immediately by one or more digits (which may not be followed by any other symbol characters).

**Examples:**

```
1
1.3
12.007
17.3E-12
3e+12
0.03E+9
```

When extra digits are used in numeric symbols, they must represent values in the range zero through nine. When numeric symbols are used as numbers, any extra digits are first converted to the corresponding character in the range 0-9, and then the symbol follows the usual rules for numbers in NetRexx (that is, the most significant digit is on the left, etc.).

In the reference implementation, the extra letters are those characters (excluding A-Z, a-z, and underscore) that result in 1 when tested with [java.lang.Character.isLetter](https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html#isLetter%28%29). Similarly, the extra digits are those characters (excluding 0-9) that result in 1 when tested with [java.lang.Character.isDigit](https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html#isDigit%28%29).

The meaning of a symbol depends on the context in which it is used. For example, a symbol may be a constant (if a number), a keyword, the name of a variable, or identify some algorithm.

It is recommended that the dollar and euro only be used in symbols in mechanically generated programs or where otherwise essential.

**Implementation minimum:** Implementations should support symbols of at least 50 characters. (But note that the length of its value, if it is a string variable, should have a much larger limit.)

**Operator characters**

The characters `+ - * / % | & = \ > <` are used (sometimes in combination) to indicate operations (see page 61) in expressions. A few of these are also used in parsing templates, and the equals sign is also used to indicate assignment. Blanks adjacent to operator characters are removed, so, for example, the sequences:

```
345>=123
345 >=123
345 >= 123
345 > = 123
```

are identical in meaning.

Some of these characters may not be available in all character sets, and if this is the case appropriate translations may be used.

**Note:** The sequences “−−”, “/∗”, and “∗/” are comment delimiters, as described earlier. The sequences “++” and “\"” are not valid in NetRexx programs.

**Special characters**

The characters `. , ; ) ( [ ]` together with the operator characters have special significance when found outside of literal strings, and constitute the set of special characters. They all act as token delimiters, and blanks adjacent to any of these (except the period) are removed, except that a blank adjacent to the outside of a parenthesis or parentheses is removed.

---

20 The sign in this context is part of the symbol; it is not an operator.
bracket is only deleted if it is also adjacent to another special character (unless this is a parenthesis or bracket and the blank is outside it, too).

Some of these characters may not be available in all character sets, and if this is the case appropriate translations may be used.

To illustrate how a clause is composed out of tokens, consider this example:

```plaintext
'REPEAT' B + 3;
```

This is composed of six tokens: a literal string, a blank operator (described later), a symbol (which is probably the name of a variable), an operator, a second symbol (a number), and a semicolon. The blanks between the “B” and the “+” and between the “+” and the “3” are removed. However one of the blanks between the 'REPEAT' and the “B” remains as an operator. Thus the clause is treated as though written:

```plaintext
'REPEAT' B+3;
```

### Implied semicolons and continuations

A semicolon (clause end) is implied at the end of each line, except if:

1. The line ends in the middle of a block comment, in which case the clause continues at the end of the block comment.
2. The last token was a hyphen. In this case the hyphen is functionally replaced by a blank, and hence acts as a continuation character.

This means that semicolons need only be included to separate multiple clauses on a single line.

**Notes:**

1. A comment is not a token, so therefore a comment may follow the continuation character on a line.
2. Semicolons are added automatically by NetRexx after certain instruction keywords when in the correct context. The keywords that may have this effect are `else`, `finally`, `otherwise`, `then`; they become complete clauses in their own right when this occurs. These special cases reduce program entry errors significantly.

### The case of names and symbols

In general, NetRexx is a case-insensitive language. That is, the names of keywords, variables, and so on, will be recognized independently of the case used for each letter in a name; the name “Swildon” would match the name “swilDon”.

NetRexx, however, uses names that may be visible outside the NetRexx program, and these may well be referenced by case-sensitive languages. Therefore, any name that has an external use (such as the name of a property, method, constructor, or class) has a defined spelling, in which each letter of the name has the case used for that letter when the name was first defined or used.

Similarly, the lookup of external names is both case-preserving and case-insensitive. If a class, method, or property is referenced by the name “Foo”, for example, an exact-case match will first be tried at each point that a search is made. If this succeeds, the search for a matching name is complete. If it is does not succeed, a case-insensitive search in the same context is carried out, and if one item is found, then the search is complete. If more than one item matches then the reference is ambiguous,
and an error is reported.

Implementations are encouraged to offer an option that requires that all name matches are exact (case-sensitive), for programmers or house-styles that prefer that approach to name matching.

**Hexadecimal and binary numeric symbols**

A **hexadecimal numeric symbol** describes a whole number, and is of the form $n\times string$. Here, $n$ is a simple number with no decimal part (and optional leading insignificant zeros) which describes the effective length of the hexadecimal string, the $\times$ (which may be in lowercase) indicates that the notation is hexadecimal, and $string$ is a string of one or more hexadecimal characters (characters from the ranges “a–f”, “A–F”, and the digits “0–9”).

The $string$ is taken as a signed number expressed in $n$ hexadecimal characters. If necessary, $string$ is padded on the left with “0” characters (note, not “sign-extended”) to length $n$ characters.

If the most significant (left-most) bit of the resulting string is zero then the number is positive; otherwise it is a negative number in twos-complement form. In both cases it is converted to a NetRexx number which may, therefore, be negative. The result of the conversion is a number comprised of the Arabic digits 0–9, with no insignificant leading zeros but possibly with a leading “−”.

The value $n$ may not be less than the number of characters in $string$, with the single exception that it may be zero, which indicates that the number is always positive (as though $n$ were greater than the the length of $string$).

**Examples:**

- $1\times 8 == -8$
- $2\times 8 == 8$
- $2\times 08 == 8$
- $0\times 08 == 8$
- $0\times 10 == 16$
- $0\times 81 == 129$
- $2\times 81 == -127$
- $3\times 81 == 129$
- $4\times 81 == 129$
- $04\times 81 == 129$
- $16\times 81 == 129$
- $4x\times 81 == -3967$
- $8x\times 81 == 61569$
- $0x\times 81 == 61569$

A **binary numeric symbol** describes a whole number using the same rules, except that the identifying character is $\mathbf{B}$ or $\mathbf{b}$, and the digits of $string$ must be either 0 or 1, each representing a single bit.

**Examples:**

- $1\mathbf{b}0 == 0$
- $1\mathbf{b}1 == -1$
- $0\mathbf{b}10 == 2$
- $0\mathbf{b}100 == 4$
- $4\mathbf{b}1000 == -8$
- $8\mathbf{b}1000 == 8$

**Note:** Hexadecimal and binary numeric symbols are a purely syntactic device for representing decimal whole numbers. That is, they are recognized only within the source of a NetRexx program, and are not equivalent to a literal string with the same characters within quotes.
Types and Classes

Programs written in the NetRexx language manipulate values, such as names, numbers, and other representations of data. All such values have an associated type (also known as a signature).

The type of a value is a descriptor which identifies the nature of the value and the operations that may be carried out on that value.

A type is normally defined by a class, which is a named collection of values (called properties) and procedures (called methods) for carrying out operations on the properties.

For example, a character string in NetRexx is usually of type Rexx, which will be implemented by the class called Rexx. The class Rexx defines the properties of the string (a sequence of characters) and the methods that work on strings. This type of string may be the subject of arithmetic operations as well as more conventional string operations such as concatenation, and so the methods implement string arithmetic as well as other string operations.

The names of types can further be qualified by the name of a package where the class is held. See the package instruction for details of packages; in summary, a package name is a sequence of one or more non-numeric symbols, separated by periods. Thus, if the Rexx class was part of the netrexx.lang package, then its qualified type would be netrexx.lang.Rexx.

In general, only the class name need be specified to refer to a type. However, if a class of the same name exists in more than one known (imported) package, then the name should be qualified by the package name. That is, if the use of just the class name does not uniquely identify the class then the reference is ambiguous and an error is reported.

Primitive types

Implementations may optionally provide primitive types for efficiency. Primitive types are “built-in” types that are not necessarily implemented as classes. They typically represent concepts native to the underlying environment (such as 32-bit binary integer numbers) and may exhibit semantics that are different from other types. NetRexx, however, makes no syntax distinction in the names of primitive types, and assumes binary constructors (see page 150) exist for primitive values.

Primitive types are necessary when performance is an overriding consideration, and so this definition will assume that primitive types corresponding to the common binary number formats are available and will describe how they differ from other types, where appropriate.

In the reference implementation, the names of the primitive types are:

    boolean char byte short int long float double

where the first two describe a single-bit value and Unicode character respectively, and the remainder describe signed numbers of various formats. The main difference between these and other types is that the primitive types are not a subclass of Object, so they cannot be assigned to a variable of type Object or passed to methods “by reference”. To use them in this way, an object must be created to “wrap” them; Java provides classes for this (for example, the class Long).

Dimensioned types

Another feature that is provided for efficiency is the concept of dimensioned types, which are types (normal or primitive) that have an associated dimension (in the sense of the dimensions of an array).
Dimensioned values are described in detail in the section on *Arrays* (see page 74).

The dimension of a dimensioned type is represented in NetRexx programs by square brackets enclosing zero or more commas, where the dimension is given by the number of commas, plus one. A dimensioned type is distinct from the type of the same name but with no dimensions.

**Examples:**

```netrexx
Rexx
int
Rexx[]
int[,] [,]
```

The examples show a normal type, a primitive type, and a dimensioned version of each (of dimension 1 and 3 respectively). The latter type would result from constructing an array thus:

```netrexx
myarray=int[10,10,10]
```

That is, the dimension of the type matches the count of indexes defined for the array.

**Minor and Dependent classes**

A *minor class* in NetRexx is a class whose name is qualified by the name of another class, called its *parent*. This qualification is indicated by the form of the name of the class: the short name of the minor class is prefixed by the name of its parent class (separated by a period). For example, if the parent is called *Foo* then the full name of a minor class *Bar* would be written *Foo.Bar*.

A *dependent class* is a minor class that has a link to its parent class that allows a child object simplified access to its parent object and its properties.

These refinements of classes and are described in the section *Minor and Dependent classes* (see page 127).
Terms

A term in NetRexx is a syntactic unit which describes some value (such as a literal string, a variable, or the result of some computation) that can be manipulated in a NetRexx program. Terms may be either simple (consisting of a single element) or compound (consisting of more than one element, with a period and no other characters between each element).

Simple terms

A simple term may be:

- A literal string (see page 41) – a character string delimited by quotes, which is a constant.
- A symbol (see page 42). A symbol that does not begin with a digit identifies a variable, a value, or a type. One that does begin with a digit is a numeric symbol, which is a constant.
- A method call (see page 53), which is of the form
  \[ \text{symbol}([\text{expression}[, \text{expression}]...]) \]
- An indexed reference (see page 73), which is of the form\(^{22}\)
  \[ \text{symbol}[\text{expression}[, \text{expression}]...]\]
- An array initializer (see page 75), which is of the form
  \[ [\text{expression}[, \text{expression}]...]\]
- A sub-expression (see page 66), which consists of any expression enclosed within a left and a right parenthesis.

Blanks are not permitted between the symbol in a method call and the “(”, or between the symbol in an indexed reference and the “[”.

Within simple terms, method calls with no arguments (that is, with no expressions between the parentheses) may be expressed without the parentheses provided that they refer to a method in the current class (or to a static method in a class used by the current class) and do not refer to a constructor (see page 56). An implementation may optionally provide a mechanism that disallows this parenthesis omission.

Compound terms

Compound terms may start with any simple term, and, in addition, may start with a qualified class name (see page 110) or a qualified constructor (see page 53). These last two both start with a package name (a sequence of non-numeric symbols separated by periods and ending in a period).

This first part of a compound term is known as the stub of the term.

\(^{22}\) The notations \[ and ] indicate square brackets appearing in the NetRexx program.
Example stubs:

"A string"
Arca
12.10
paint(g)
indexedVar[i+1]
("A" "string")
java.lang.Math        -- qualified class name
netrexx.lang.Rexx(1)  -- qualified constructor

All stubs are syntactically valid terms (either simple or compound) and may optionally be followed by a *continuation*, which is one or more additional non-numeric symbols, method calls, or indexed references, separated from each other and from the stub by *connectors* which are periods.

Example compound terms:

"A rabbit".word(2).pos('b')
Fluffy.left(3)
12.10.max(j)
paint(g).picture
indexedVar[i+1].length
("A" "string").word(1)
java.lang.Math.PI
java.lang.Math.log(10)

Within compound terms, method calls with no arguments (that is, with no expressions between the parentheses) may be expressed without the parentheses provided that they do not refer to a constructor (see page 56). For example, the term:

Thread.currentThread().suspend()

could be written:

Thread.currentThread.suspend

An implementation may optionally provide a mechanism that disallows this parenthesis omission.

**Evaluation of terms**

Simple terms are evaluated as a whole, as described below. Compound terms are evaluated from left to right. First the stub is evaluated according to the rules detailed below. The type of the value of the stub then qualifies the next element of the term (if any) which is then evaluated (again, the exact rules are detailed below). This process is then repeated for each element in the term.

For instance, for the example above:

"A rabbit".word(2).pos('b')

the evaluation proceeds as follows:

1. The stub ("A rabbit") is evaluated, resulting in a string of type Rexx.
2. Because that string is of type Rexx, the Rexx class is then searched for the word method. This is then invoked on the string, with argument 2. In other words, the word method is invoked with the string “A rabbit” as its current context (the properties of the Rexx class when the method is invoked reflect that value).

   This returns a new string of type Rexx, “rabbit” (the second word in the original string).
3. In the same way as before, the pos method of the Rexx class is then invoked on the new string,
with argument “b”.

This returns a new string of type \texttt{Rexx}, “3” (the position of the first “b” in the previous result). This value, “3”, is the final value of the term.

The remainder of this section gives the details of term evaluation; it is best skipped on first reading.

\textbf{Simple term evaluation}

All simple terms may also be used as stubs, and are evaluated in precisely the same way as stubs, as described below. For example, numeric symbols are evaluated as though they were enclosed in quotes; their value is a string of type \texttt{Rexx}.

In binary classes (see page 80), however, simple terms that are strings or numeric symbols are given an implementation-defined string or primitive type respectively, as described in the section on \textit{Binary values and operations} (see page 149)

\textbf{Stub evaluation}

A term’s stub is evaluated according to the following rules:

- If the stub is a literal string, its value is the string, of type \texttt{Rexx}, constructed from that literal.
- If the stub is a numeric symbol, its value is the string, of type \texttt{Rexx}, constructed from that literal (as though the literal were enclosed in quotes).
- If the stub is an unqualified method or constructor call, or a qualified constructor call, then its value and type is the result of invoking the method identified by the stub, as described in \textit{Methods and Constructors} (see page 53).
- If the stub is a (non-numeric) symbol, then its value and type will be determined by whichever of the following is first found:
  1. A local variable or method argument within the current method, or a property in the current class.
  2. A method whose name matches the symbol, and takes no arguments, and that is not a constructor, in the current class.\textsuperscript{23} If the stub is part of a compound symbol, then it may also be in a superclass, searching upwards from the current class.
  3. A static or constant property, or a static method,\textsuperscript{24} whose name matches the symbol (and that takes no arguments, if a method) in a class listed in the \texttt{uses} phrase of the \texttt{class} instruction. Each class from the list is searched for a matching property or method, and then its superclasses are searched upwards from the class in the same way; this process is repeated for each of the classes, in the order specified in the list.
  4. One of the allowed special words described in \textit{Special words and methods} (see page 130), such as \texttt{this} or \texttt{version}.
  5. The short name of a known class or primitive type (in which case the stub has no value, just a type).
- If the stub is an indexed reference, then its value and type will be determined by whichever of

\textsuperscript{23} Unless parenthesis omission is disallowed by an implementation option, such as \texttt{options strictargs}.

\textsuperscript{24} Unless parenthesis omission is disallowed by an implementation option, such as \texttt{options strictargs}. 
1. The symbol naming the reference is an undimensioned local variable or method argument within the current method, or a property in the current class, which has type Rexx. In this case, the reference is to an indexed string (see page 73); the expressions within the brackets must be convertible to type Rexx, and the type of the result will be Rexx.

2. The symbol naming the reference is a dimensioned local variable or method argument within the current method, or a property in the current class. In this case, the reference is to an array (see page 74), and the expressions within the brackets must be convertible to whole numbers allowed for array indexes. The type of the result will have the type of the array, with dimensions reduced by the number of dimensions specified in the stub.

   For example, if the array foo was of type Baa[,] and the stub referred to foo[1,2], then the result would be of type Baa[ ]. It would have been an error for the stub to have specified more than four dimensions.

3. The symbol naming the reference is the name of a static or constant property in a class listed in the uses phrase of the class instruction. Each class from the list is searched in the same way as for symbols, above. The reference may be to an indexed string or an array, as for properties in the current class.

4. The symbol naming the reference is the name of a primitive type or the short name of a known class, and there are no expressions within the brackets (just optional commas). In this case, the stub describes a dimensioned type (see page 46).

5. The symbol naming the reference is the name of a primitive type or is the short name of a known class, and there are one or more expressions within the brackets. In this case, the reference is to an array constructor (see page 74); the expressions within the brackets must be convertible to non-negative whole numbers allowed for array indexes, and the value is an array of the specified type, dimensions, and size.

   • If the stub is a sub-expression, then its value and type will be determined by evaluating the expression (see page 61) within the parentheses.

   • If the stub starts with the name of a package, then it will either describe a qualified type (see page 46) or a qualified constructor (see page 56). Its type will be same in either case, and if a constructor then its value will be the value returned by the constructor.

If the stub is not followed by further segments, the final value and type of the term is the value and type of the stub.

**Continuation evaluation**

Each segment of a term’s continuation is evaluated from left to right, according to the type of the evaluation of the term so far (the continuation type) and the syntax of the new segment:

• If the segment is a method call, then its value and type is the result of invoking the matching method in the class defining the continuation type (or a superclass or subclass of that class), as described in Methods and Constructors (see page 53). Note that method calls in term continuations cannot be constructors.

• If the stub is an indexed reference, then it will refer to a property in the class defining the continuation type (or a superclass of that class). That property will either be an undimensioned NetRexx string (type Rexx) or a dimensioned array. In either case, it is evaluated in the same
way as an indexed reference found in the stub, except that it is not necessarily in the current class, cannot be an array constructor, and cannot result in a dimensioned type.

• If the segment is a symbol, then it refers to either a property or a method in the class defining the continuation type (or a superclass of that class).\textsuperscript{25}

The search for the property or method starts with the class defining the continuation type. If a property name matches, it is used; if not, a method of the same name and having no arguments (or only optional arguments) will match. If neither property nor method is found, then the same search is applied to each of the continuation class’s superclasses in turn, starting with the class that it extends.

As a convenient special case, if the property or method is not found, then if the segment is the final segment of the term and is the simple symbol \texttt{length} and the continuation type is a single-dimensioned array, then the segment evaluates to the size of the array. This will be a non-negative whole number of an appropriate primitive type (or of type \texttt{Rexx} if there is no appropriate type).

The final value and type of the term is the value and type determined by the evaluation of the last segment of the continuation.

\textbf{Arrays in terms}

If a partially-evaluated term results in a dimensioned array (see page \textsuperscript{74}), its type is treated as type \texttt{Object} so that evaluation of the term can continue. For example, in

```rexx
c=\texttt{char[]} \texttt{"tosh"}
say \texttt{c}.\texttt{toString()}
```

the variable \texttt{ca} is an array of characters; in the expression on the second line the method \texttt{toString()} of the class \texttt{Object} will be found.\textsuperscript{26}

\textsuperscript{25} Unless parenthesis omission is disallowed by an implementation option, such as \texttt{options strictargs}, in which case it can only be a property.

\textsuperscript{26} \textit{In the reference implementation, this would return an identifier for the object.}
Methods and Constructors

Instructions in NetRexx are grouped into methods, which are named routines that always belong to (are part of) a class.

Methods are invoked by being referenced in a term (see page 48), which may be part of an expression or be a clause in its own right (a method call instruction). In either case, the syntax used for a method invocation is:

```
symbol([expression[,expression]...])
```

The symbol, which must be non-numeric, is called the name of the method. It is important to note that the name of the method must be followed immediately by the “(”, with no blank in between, or the construct will not be recognized as a method call (a blank operator would be assumed at that point instead).

The expressions (separated by commas) between the parentheses are called the arguments to the method. Each argument expression may include further method calls.

The argument expressions are evaluated in turn from left to right and the resulting values are then passed to the method (the procedure for locating the method is described below). The method then executes some algorithm (usually dependent on any arguments passed, though arguments are not mandatory) and will eventually return a value. This value is then included in the original expression just as though the entire method reference had been replaced by the name of a variable whose value is that returned data.

For example, the substr method is provided for strings of type Rexx and could be used as:

```netrexx
c='abcdefghijk'
a=c.substr(3,7)
/* would set A to "cdefghi" */
```

Here, the value of the variable c is a string (of type Rexx). The substr (substring) method of the Rexx class is then invoked, with arguments 3 and 7, on the value referred to by c. That is, the properties available to (the context of) the substr method are the properties constructed from the literal string 'abcdefghijk'. The method returns the substring of the value, starting at the third character and of length seven characters.

A method may have a variable number of arguments: only those required need be specified. For example, 'ABCDEF'.substr(4) would return the string 'DEF', as the substr method will assume that the remainder of the string is to be returned if no length is provided.

Method invocations that take no arguments may omit the (empty) parentheses in circumstances where this would not be ambiguous. See the section on Terms (see page 48) for details.

Implementation minimum: At least 10 argument expressions should be allowed in a method call.

Method call instructions

When a clause in a method consists of just a term, and the final part of the term is a method invocation, the clause is a method call instruction:

```
symbol([expression[,expression]...]);
```

The method is being called as a subroutine of the current method, and any returned value is discarded. In this case (and in this case only), the method invoked need not return a value (that is, the return
A method call instruction that is the first instruction in a constructor (see below) can only invoke the special constructors `this` and `super`.

### Method resolution (search order)

Method resolution in NetRexx proceeds as follows:

- If the method invocation is the first part (stub) of a term, then:
  1. The current class is searched for the method (see below for details of searching).
  2. If not found in the current class, then the superclasses of the current class are searched, starting with the class that the current class extends.
  3. If still not found, then the classes listed in the `uses` phrase of the `class` instruction are searched for the method, which in this case must be a static method (see page 100). Each class from the list is searched for the method, and then its superclasses are searched upwards from the class; this process is repeated for each of the classes, in the order specified in the list.
  4. If still not found, the method invocation must be a constructor (see below) and so the method name, which may be qualified by a package name, should match the name of a primitive type or a known class (type). The specified class is then searched for a constructor that matches the method invocation.

- If the method invocation is not the first part of the term, then the evaluation of the parts of the term to the left of the method invocation will have resulted in a value (or just a type), which will have a known type (the continuation type). Then:
  1. The class that defines the continuation type is searched for the method (see below for details of searching).
  2. If not found in that class, then the superclasses of that class are searched, starting with the class that that class extends.

If the search did not find a method, an error is reported.

If the search did find a method, that is the method which is invoked, except in one case:

- If the evaluation so far has resulted in a value (an object), then that value may have a type which is a subclass of the continuation type. If, within that subclass, there is a method that exactly overrides (see page 55) the method that was found in the search, then the method in the subclass is invoked.

This case occurs when an object is earlier assigned to a variable of a type which is a superclass of the type of the object. This type simplification hides the real type of the object from the language processor, though it can be determined when the program is executed.

Searching for a method in a class proceeds as follows:

1. Candidate methods in the class are selected. To be a candidate method:
   - the method must have the same name as the method invocation (independent of the case (see page 44) of the letters of the name)

27 A method call instruction is equivalent to the `call` instruction of other languages, except that no keyword is required.
the method must have the same number of arguments as the method invocation (or more arguments, provided that the remainder are shown as optional in the method definition)

it must be possible to assign the result of each argument expression to the type of the corresponding argument in the method definition (if strict type checking is in effect, the types must match exactly).

2. If there are no candidate methods then the search is complete; the method was not found.

3. If there is just one candidate method, that method is used; the search is complete.

4. If there is more than one candidate method, the sum of the costs of the conversions (see page 60) from the type of each argument expression to the type of the corresponding argument defined for the method is computed for each candidate method.

5. The costs of those candidates (if any) whose names match the method invocation exactly, including in case, are compared; if one has a lower cost than all others, that method is used and the search is complete.

6. The costs of all the candidates are compared; if one has a lower cost than all others, that method is used and the search is complete.

7. If there remain two or more candidates with the same minimum cost, the method invocation is ambiguous, and an error is reported.

Note: When a method is found in a class, superclasses of that class are not searched for methods, even though a lower-cost method may exist in a superclass.

Method overriding

A method is said to exactly override a method in another class if

1. the method in the other class has the same name as the current method

2. the method in the other class is not private

3. the other class is a superclass of the current class, or is a class that the current class implements (or is a superclass of one of those classes).

4. the number and type of the arguments of the method in the other class exactly match the number and type of the arguments of the current method (where subsets are also checked, if either method has optional arguments).

For example, the Rexx class includes a substr (see page 166) method, which takes from one to three strings of type Rexx. In the class:

```plaintext
class mystring extends Rexx
  method substr(n=Rexx, length=Rexx)
    return this.reverse.substr(n, length)
  method substr(n=int, length=int)
    return this.reverse.substr(Rexx n, Rexx length)
```

the first method exactly overrides the substr method in the Rexx class, but the second does not, because the types of the arguments do not match.

A method that exactly overrides a method is assumed to be an extension of the overridden method, to be used in the same way. For such a method, the following rules apply:
• It must return a value of the same type as the overridden method (or none, if the overridden method returns none).

• It must be at least as visible as the overridden routine. For example, if the overridden routine is `public` then it must also be `public`.

• If the overridden method is `static` then it must also be `static`.

• If the overridden method is not `static` then it must not be `static`.

• If the underlying implementation checks exceptions (see page 152), only those checked exceptions that are signalled by the overridden method may be left uncaught in the current method.

### Constructor methods

As described above, methods are usually invoked in the context of an existing value or type. A special kind of method, called a constructor method, is used to actually create a value of a given type (an object).

Constructor methods always have the same short name as the class in which they are found, and construct and return a value of the type defined by that class (sometimes known as an `instance` of that class). If the class is part of a package, then the constructor call may be qualified by the package name.

#### Example constructors:

```
File('Dan.yr.Ogof')
java.io.File('Speleogroup.letter')
Rexx('some words')
netrexx.lang.Rexx(1)
```

There will always be at least one constructor if values can be created for a class. NetRexx will add a default public constructor that takes no arguments if no constructors are provided, unless the components of the class are all static or constant, or the class is an interface class.

All constructors follow the same rules as other methods, and in addition:

1. Constructor calls always include parentheses in the syntax, even if no arguments are supplied. This distinguishes them from a reference to the type of the same name.

2. Constructors must call a constructor of their superclass (the class they extend) before they carry out any initialization of their own. This is so any initialization carried out by the superclass takes place, and at the appropriate moment. Only after this call is complete can they make any reference to the special words `this` or `super` (see page 130).

   Therefore, the first instruction in a constructor must be either a call to the superclass, using the special constructor `super()` (with optional arguments), or a call to another constructor in the same class, using the special constructor `this()` (with optional arguments). In the latter case, eventually a constructor that explicitly calls `super()` will be invoked and the chain of local constructor calls ends.

   As a convenience, NetRexx will add a default call to `super()`, with no arguments, if the first instruction in a constructor is not a call to `this()` or `super()`.

3. The properties of a constructed value are initialized, in the order given in the program, after the call to `super()` (whether implicit or explicit).
4. By definition, constructors create a value (object) whose type is defined by the current class, and then return that value for use. Therefore, the `returns` keyword on the `method` instruction (see page 98) that introduces the constructor is optional (if given, the type specified must be that of the class). Similarly, the only possible forms of the `return` instruction used in a constructor are either “return this;”, which returns the value that has just been constructed, or just “return;”, in which case, the “this” is assumed (this form will be assumed at the end of a method, as usual, if necessary).

Here is an example of a class with two constructors, showing the use of `this()` and `super()`, and taking advantage of some of the assumptions:

```rxml
class MyChars extends SomeClass

    properties private
    /* the data 'in' the object */
    value=char[]

    /* construct the object from a char array */
    method MyChars(array=char[])
        /* initialize superclass */
        super()
        value=array -- save the value

    /* construct the object from a String */
    method MyChars(s=String)
        /* convert to char[] and use the above */
        this(s.toCharArray())

Objects of type MyChars could then be created thus:

    myvar=MyChars("From a string")

or by using an argument that has type char[].
```
Type conversions

As described in the section on Types and classes (see page 46), all values that are manipulated in NetRexx have an associated type. On occasion, a value involved in some operation may have a different type than is needed, and in this case conversion of a value from one type to another is necessary.

NetRexx considerably simplifies the task of programming, without losing robustness, by making many such conversions automatic. It will automatically convert values providing that there is no loss of information caused by the automatic conversion (or if there is, an exception would be raised).

Conversions can also be made explicit by concatenating a type (see page 64) to a value and in this case less robust conversions (sometimes known as casts) may be effected. See below for details of the permitted automatic and explicit conversions.

Almost all conversions carry some risk of failure, or have a performance cost, and so it is expected that implementations will provide options to either report costly conversions or require that programmers make all conversions explicit. Such options might be recommended for certain critical programming tasks, but are not necessary for general programming.

Permitted automatic conversions

In general, the semantics of a type is unknown, and so conversion (from a source type to a target type) is only possible in the following cases:

- The target type and the source type are identical (the trivial case).
- The target type is a superclass of the source type, or is an interface class implemented by the source type. This is called type simplification, and in this case the value is not altered, no information is lost, and an explicit conversion may be used later to revert the value to its original type.
- The source type has a dimension, and the target type is Object.
- The source type is null and the target type is not primitive.
- The target and source types have known semantics (that is, they are “well-known” to the implementation) and the conversion can be effected without loss of information (other than knowledge of the original type). These are called well-known conversions.

Necessarily, the well-known conversions are implementation-dependent, in that they depend on the standard classes for the implementation and on the primitive types supported (if any). Equally, it is this automatic conversion between strings and the primitive types of an implementation that offer the greatest simplifications of NetRexx programming.

For example, if the implementation supported an int binary type (perhaps a 32-bit integer) then this can safely be converted to a NetRexx string (of type Rexx). Hence a value of type int can be added to a number which is a NetRexx string (resulting in a NetRexx string) without concern over the difference in the types of the two terms used in the operation.

Conversely, converting a long integer to a shorter one without checking for truncation of significant digits could cause a loss of information and would not be permitted.

---

28 In the reference implementation, options strictassign may be used to disallow automatic conversions.
In the reference implementation, the semantics of each of the following types is known to the language processor (the first four are all string types, and the remainder are known as binary numbers):

- `netrexx.lang.Rexx` – the NetRexx string class
- `java.lang.String` – the Java string class
- `char` – the Java primitive which represents a single character
- `char[]` – an array of chars
- `boolean` – a single-bit primitive
- `byte, short, int, long` – signed integer primitives (8, 16, 32, or 64 bits)
- `float, double` – floating-point primitives (32 or 64 bits)

Under the rules described above, the following well-known conversions are permitted:

- Rexx to binary number, char[], String, or char
- String to binary number, char[], Rexx, or char
- char to binary number, char[], String, or Rexx
- char[] to binary number, Rexx, String, or char
- binary number to Rexx, String, char[], or char
- binary number to binary number (if no loss of information can take place – no sign, high order digits, decimal part, or exponent information would be lost)

**Notes:**

1. Some of the conversions can cause a run-time error (exception), as when a string represents a number that is too large for an int and a conversion to int is attempted, or when a string that does not contain exactly one character is converted to a char.

2. The boolean primitive is treated as a binary number that may only take the values 0 or 1. A boolean may therefore be converted to any binary number type, as well as any of the string (or char) types, as the character “0” or “1”. Similarly, any binary number or string can be converted to boolean (and must have a value of 0 or 1 for the conversion to succeed).

3. The char type is a single-character string (it is not a number that represents the encoding of the character).

**Permitted explicit conversions**

Explicit conversions are permitted for all permitted automatic conversions and, in addition, when:

- The target type is a subclass of the source type, or implements the source type. This conversion will fail if the value being converted was not originally of the target type (or a subclass of the target type).

- Both the source and target types are primitive and (depending on the implementation) the conversion may fail or truncate information.

- The target type is Rexx or a well-known string type (all values have an explicit string representation).
 Costs of conversions

All conversions are considered to have a cost, and, for permitted automatic conversions, these costs are used to select a method for execution when several possibilities arise, using the algorithm described in Methods and Constructors (see page 54).

For permitted automatic conversions, the cost of a conversion from a source type to a target type will range from zero through some arbitrary positive value, constrained by the following rules:

• The cost is zero only if the source and target types are the same, or if the source type is null and the target type is not primitive.

• Conversions from a given primitive source type to different primitive target types should have different costs. For example, conversion of an 8-bit number to a 64-bit number might cost more than conversion of a 8-bit number to a 32-bit number.

• Conversions that may require the creation of a new object cost more than those that can never require the creation of a new object.

• Conversions that may fail (raise an exception) cost more than those that may require the creation of an object but can never fail.

Within these constraints, exact costs are arbitrary, and (because they mostly involve implementation-dependent primitive types) are necessarily implementation-dependent. The intent is that the “best performance” method be selected when there is a possibility of more than one.
Expressions and Operators

Many clauses can include expressions. Expressions in NetRexx are a general mechanism for combining one or more data items in various ways to produce a result, usually different from the original data.

Expressions consist of one or more terms (see page 48), such as literal strings, symbols, method calls, or sub-expressions, and zero or more operators that denote operations to be carried out on terms. Most operators act on two terms, and there will be at least one of these dyadic operators between every pair of terms. There are also prefix (monadic) operators, that act on the term that is immediately to the right of the operator. There may be one or more prefix operators to the left of any term, provided that adjacent prefix operators are different.

Evaluation of an expression is left to right, modified by parentheses and by operator precedence (see page 66) in the usual “algebraic” manner. Expressions are wholly evaluated, except when an error occurs during evaluation.

As each term is used in an expression, it is evaluated as appropriate and its value (and the type of that value) are determined.

The result of any operation is also a value, which may be a character string, a data object of some other type, or (in special circumstances) a binary representation of a character or number. The type of the result is well-defined, and depends on the types of any terms involved in an operation and the operation carried out. Consequently, the result of evaluating any expression is a value which has a known type.

Note that the NetRexx language imposes no restriction on the maximum size of results, but there will usually be some practical limitation dependent upon the amount of storage and other resources available during execution.

Operators

The operators in NetRexx are constructed from one or more operator characters (see page 43). Blanks (and comments) adjacent to operator characters have no effect on the operator, and so the operators constructed from more than one character may have embedded blanks and comments. In addition, blank characters, where they occur between tokens within expressions but are not adjacent to another operator, also act as an operator.

The operators may be subdivided into five groups: concatenation, arithmetic, comparative, logical, and type operators. The first four groups work with terms whose type is “well-known” (that is, strings, or known types that can be be converted to strings without information loss). The operations in these groups are defined in terms of their operations on strings.

29 One operator, direct concatenation, is implied if two terms abut (see page 62).
**Concatenation** The concatenation operators are used to combine two strings to form one string by appending the second string to the right-hand end of the first string. The concatenation may occur with or without an intervening blank:

(\(\text{blank}\)) Concatenate terms with one blank in between.

|| Concatenate without an intervening blank.

(\(\text{abuttal}\)) Concatenate without an intervening blank.

Concatenation without a blank may be forced by using the \(\text{||}\) operator, but it is useful to remember that when two terms are adjacent and are not separated by an operator,\(^{30}\) they will be concatenated in the same way. This is the abuttal operation. For example, if the variable Total had the value '37.4', then Total \(\%\) would evaluate to '37.4\%'.

Values that are not strings are first converted to strings before concatenation.

**Arithmetic** Character strings that are numbers (see page 65) may be combined using the arithmetic operators:

+ Add.

- Subtract.

* Multiply.

/ Divide.

\% Integer divide.

Divide and return the integer part of the result.

// Remainder.

Divide and return the remainder (this is not modulo, as the result may be negative).

** Power.

Raise a number to a whole number power.

Prefix - Same as the subtraction: “0–number”.

Prefix + Same as the addition: “0+number”.

The section on Numbers and Arithmetic (see page 140) describes numeric precision, the format of valid numbers, and the operation rules for arithmetic. Note that if an arithmetic result is shown in exponential notation, then it is likely that rounding has occurred.

In binary classes (see page 80), the arithmetic operators will use binary arithmetic if both terms involved have values which are binary numbers. The section on Binary values and operations (see page 149) describes binary arithmetic.

**Comparative** The comparative operators compare two terms and return the value '1' if the result of the comparison is true, or '0' otherwise. Two sets of operators are defined: the strict comparisons and the normal comparisons.

The strict comparative operators all have one of the characters defining the operator doubled. The “\(\text{==}\)”, and “\(\text{\%=}\)” operators test for strict equality or inequality between

---

30 This can occur when the terms are syntactically distinct (such as a literal string and a symbol).
two strings. Two strings must be identical to be considered strictly equal. Similarly, the other strict comparative operators (such as “>>” or “<<”) carry out a simple left-to-right character-by-character comparison, with no padding of either of the strings being compared. If one string is shorter than, and is a leading sub-string of, another then it is smaller (less than) the other. Strict comparison operations are case sensitive, and the exact collating order may depend on the character set used for the implementation.\footnote{For example, in an ASCII or Unicode environment, the digits 0-9 are lower than the alphabetics, and lowercase alphabetics are higher than uppercase alphabetics. In an EBCDIC environment, lowercase alphabetics precede uppercase, but the digits are higher than all the alphabetics.}

For all the other comparative operators, if both the terms involved are numeric,\footnote{That is, if they can be compared numerically without error.} a numeric comparison (in which leading zeros are ignored, etc.) is effected; otherwise, both terms are treated as character strings. For this character string comparison, leading and trailing blanks are ignored, and then the shorter string is padded with blanks on the right. The character comparison operation takes place from left to right, and is not case sensitive (that is, “Yes” compares equal to “yes”). As for strict comparisons, the exact collating order may depend on the character set used for the implementation.

The comparative operators return true (‘1’) if the terms are:

**Normal** comparative operators:

- `=` Equal (numerically or when padded, etc.).
- `\=` Not equal (inverse of `=`).
- `>` Greater than.
- `<` Less than.
- `>\`, `<>` Greater than or less than (same as “Not equal”).
- `>=`, `\<` Greater than or equal to, not less than.
- `<=`, `\>` Less than or equal to, not greater than.

**Strict** comparative operators:

- `==` Strictly equal (identical).
- `\==` Strictly not equal (inverse of `==`).
- `>>` Strictly greater than.
- `<<` Strictly less than.
- `>>=`, `\<<` Strictly greater than or equal to, strictly not less than.
- `<==`, `\>>` Strictly less than or equal to, strictly not greater than.

The equal and not equal operators (”=”,”==”, “\=”,” and “\==”) may be used to compare two objects which are not strings for equality, if the implementation allows them to be compared (usually they will need to be of the same type). The strict operators test whether the two objects are in fact the same object,\footnote{Note that two distinct objects compared in this way may contain values (properties) that are identical, yet they will not compare equal as they are not the same object.} and the normal
operators may provide a more relaxed comparison, if available to the implementation.\[34\]

In binary classes (see page 80), all the comparative operators will use binary arithmetic to effect the comparison if both terms involved have values which are binary numbers. In this case, there is no distinction between the strict and the normal comparative operators. The section on Binary values and operations (see page 149) describes the binary arithmetic used for comparisons.

**Logical (Boolean)**

A character string is taken to have the value “false” if it is '0', and “true” if it is '1'. The logical operators take one or two such values (values other than '0' or '1' are not allowed) and return '0' or '1' as appropriate:

- & And. Returns 1 if both terms are true.
- | Inclusive or. Returns 1 if either term is true.
- && Exclusive or. Returns 1 if either (but not both) is true.
- Prefix \ Logical not. Negates; 1 becomes 0 and vice versa.

In binary classes (see page 80), the logical operators will act on all bits in the values if both terms involved have values which are boolean or integers. The section on Binary values and operations (see page 149) describes this in more detail.

**Type**

Several of the operators already described can be used to carry out operations related to types. Specifically:

- Any of the concatenation operators may be used for type concatenation, which concatenates a type to a value. All three operators (blank, “||”, and abuttal) have the same effect, which is to convert (see page 58)\[35\] the value to the type specified (if the conversion is not possible, an error is reported or an exception is signalled). The type must be on the left-hand side of the operator.

**Examples:**

```
String "abc"
int (a+1)
long 1
Exception e
InputStream myfile
```

- A type on the left hand side of an operator that could be a prefix operator (+, -, or \) is assumed to imply type concatenation after the prefix operator is applied to the right-hand operand, as though an explicit concatenation operator were placed before the prefix operator.

For example:

---

34 In the reference implementation, the `equals` method is used for normal comparisons. Where not provided by a type, this is implemented by the `Object` class as a strict comparison.

35 This is sometimes known as casting.
x=int -y
means that \(-y\) is evaluated, and then the result is converted to \(\text{int}\) before being assigned to \(x\).\(^{36}\)

- The “less than or equal” and the “greater than or equal” operators (“\(<=\)” and “\(>=\)”)
  may be used with a type on either side of the operator, or on both sides. In this case, they test whether a value or
type is a subclass of, or is the same as, a type, or vice versa.

  **Examples:**

  ```
  if i<=Object then say 'I is an Object'
  if String>=i then say 'I is a String'
  if String<=Object then say 'String is an Object'
  ```

  The precedence of these operators is not affected by their being used with types as operands.

**Numbers**

The arithmetic operators above require that both terms involved be numbers; similarly some of the
comparative operators carry out a numeric comparison if both terms are numbers.

Numbers are introduced and defined in detail in the section on Numbers and arithmetic (see page 140).
In summary, numbers are character strings consisting of one or more decimal digits optionally
prefixed by a plus or minus sign, and optionally including a single period (“.”) which then represents
a decimal point. A number may also have a power of ten suffixed in conventional exponential
notation: an “\(E\)” (uppercase or lowercase) followed by a plus or minus sign then followed by one or
more decimal digits defining the power of ten.

Numbers may have leading blanks (before and/or after the sign, if any) and may have trailing blanks.
Blanks may not be embedded among the digits of a number or in the exponential part.

**Examples:**

`'12'`
`'-17.9'`
`'127.0650'`
`'73e+128'`
`' + 7.9E-5 '`
`'00E+000'`

Note that the sequence \(-17.9\) (without quotes) in an expression is not simply a number. It is a minus
operator (which may be prefix minus if there is no term to the left of it) followed by a positive number.
The result of the operation will be a number.

A whole number (see page 147) in NetRexx is a number that has a zero (or no) decimal part.

**Implementation minimum:** All implementations must support 9-digit arithmetic. In unavoidable
cases this may be limited to integers only, and in this case the divide operator (“\(/\)”)
must not be supported. If exponents are supported in an implementation, then they must be supported for
exponents whose absolute value is at least as large as the largest number that can be expressed as an
exact integer in default precision, \(i.e.,\ 999999999\).

\(^{36}\) This could also have been written \(x=\text{int} (-y)\).
Parentheses and operator precedence

Expression evaluation is from left to right; this is modified by parentheses and by operator precedence:

- When parentheses are encountered, other than those that identify method calls (see page 53), the entire sub-expression delimited by the parentheses is evaluated immediately when the term is required.

- When the sequence
  \[ \text{term}_1 \text{ operator}_1 \text{ term}_2 \text{ operator}_2 \text{ term}_3 \]
  is encountered, and \text{operator}_2 has a higher precedence than \text{operator}_1, then the operation \((\text{term}_2 \text{ operator}_2 \text{ term}_3)\) is evaluated first. The same rule is applied repeatedly as necessary.

Note, however, that individual terms are evaluated from left to right in the expression (that is, as soon as they are encountered). It is only the order of operations that is affected by the precedence rules.

For example, “\*” (multiply) has a higher precedence than “+” (add), so 3+2*5 will evaluate to 13 (rather than the 25 that would result if strict left to right evaluation occurred). To force the addition to be performed before the multiplication the expression would be written \((3+2)\times5\), where the first three tokens have been formed into a sub-expression by the addition of parentheses.

The order of precedence of the operators is (highest at the top):

- **Prefix operators**: +  -  \
- **Power operator**: **
- **Multiplication and division**: *  /  %  //
- **Addition and subtraction**: +  -
- **Concatenation**: (blank)  ||  (abuttal)
- **Comparative operators**: =  ==  >  <  <= >=  <<  >> etc.
- **And**: &
- **Or, exclusive or**: |  &&

If, for example, the symbol \text{a} is a variable whose value is ‘3’, and \text{day} is a variable with the value ‘Monday’, then:

\begin{verbatim}
  a+5  == '8'
a-4+2 == '-5'
a/2   == '1.5'
a%2  == '1'
0.5**2 == '0.25'
(a+1)>7 == '0'   /* that is, False */
' '=1' == '1'   /* that is, True */
' !===' == '0'  /* that is, False */
' \='1' == '1'  /* that is, True */
(a+1)*3=12 == '1' /* that is, True */
'077'>='11'   == '1' /* that is, True */
'077'>>>='11' == '0' /* that is, False */
'abc'>>>'ab' == '1' /* that is, True */
'If it is' day == 'If it is Monday'
day.substr(2,3) == 'ond'
'!'day!!' == '!'Monday!'
\end{verbatim}
Note: The NetRexx order of precedence usually causes no difficulty, as it is the same as in conventional algebra and other computer languages. There are two differences from some common notations; the prefix minus operator always has a higher priority than the power operator, and power operators (like other operators) are evaluated left-to-right. Thus

\[\begin{align*}
-3\cdot2 & = 9 \quad /* \text{not} \ -9 \ */ \\
-(2+1)\cdot2 & = 9 \quad /* \text{not} \ -9 \ */ \\
2\cdot2\cdot3 & = 64 \quad /* \text{not} \ 256 \ */
\end{align*}\]

These rules were found to match the expectations of the majority of users when the Rexx language was first designed, and NetRexx follows the same rules.
Clauses and Instructions

Clauses (see page 40) are recognized, and can usefully be classified, in the following order:

**Null clauses**
A clause that is empty or comprises only blanks, comments, and continuations is a *null clause* and is completely ignored by NetRexx (except that if it includes a comment it will be traced, if reached during execution).

**Note:** A null clause is not an instruction, so (for example) putting an extra semicolon after the *then* or *else* in an *if* instruction is not equivalent to putting a dummy instruction (as it would be in C or PL/I). The *nop* instruction is provided for this purpose.

**Assignments**
Single clauses within a class and of the form `term=expression;` are instructions known as *assignments* (see page 69). An assignment gives a variable, identified by the `term`, a type or a new value.

In just one context, where property assignments are expected (before the first method in a class), the “=” and the expression may be omitted; in this case, the term (and hence the entire clause) will always be a simple non-numeric symbol which names the property.

**Method call instructions**
A method call instruction (see page 53) is a clause within a method that comprises a single term that is, or ends in, a method invocation.

**Keyword instructions**
A *keyword instruction* consists of one or more clauses, the first of which starts with a non-numeric symbol which is not the name of a variable or property in the current class (if any) and is immediately followed by a blank, a semicolon (which may be implied by the end of a line), a literal string, or an operator (other than “=”, which would imply an assignment). This symbol, the *keyword*, identifies the instruction.

Keyword instructions control the external interfaces, the flow of control, and so on. Some keyword instructions (see page 77) *(do, if, loop, or select)* can include nested instructions.
Assignments and Variables

A variable is a named item whose value may be changed during the course of execution of a NetRexx program. The process of changing the value of a variable is called assigning a new value to it.

Each variable has an associated type, which cannot change during the execution of a program; therefore, the values assigned to a given variable must always have a type that can safely be assigned to that variable.

Variables may be assigned a new value by the method or parse instructions, but the most common way of changing the value of a variable is by using an assignment instruction. Any clause within a class and of the form:

```
assignment;
where assignment is:
   term=expression
```

is taken to be an assignment instruction. The result of the expression becomes the new value of the variable named by the term to the left of the equals sign. When the term is simply a symbol, this is called the name of the variable.

Example:

```
/* Next line gives FRED the value 'Frederic' */
fred='Frederic'
```

The symbol naming the variable cannot begin with a digit (0-9).

Within a NetRexx program, variable names are not case-sensitive (for example, the names fred, Fred, and FRED refer to the same variable). Where public names are exposed (for example, the names of properties, classes, and methods, and in cross-reference listings) the case used for the name will be that used when the name was first introduced (“first” is determined statically by position in a program rather than dynamically).

Similarly, the type of a NetRexx variable is determined by the type of the value of the expression that is first assigned to it. For subsequent assignments, it is an error to assign a value to a variable with a type mismatch unless the language processor can determine that the value can be assigned safely to the type of the variable.

In practice, this means that the types must match exactly, be a simplification, or both be “well-known” types such as Rexx, String, int, etc., for which safe conversions are defined. The possibilities are described in the section on Conversions (see page 58).
For example, if there are types (classes) called `ibm.util.hex`, `RunKnown`, and `Window`, then:

```
hexy=ibm.util.hex(3) -- 'hexy' has type 'ibm.util.hex'
rk=RunKnown()        -- 'rk' has type 'RunKnown'
fred=Window(10, 20)  -- 'fred' has type 'Window'
s="Los Lagos"        -- 's' has type 'Rexx'
j=5                  -- 'j' has type 'Rexx'
```

The first three examples invoke the constructor method for the type to construct a value (an object). A constructor method always has the same name as the class to which it belongs, and returns a new value of that type. Constructor methods are described in detail in Methods and Constructors (see page 53).

The last two examples above illustrate that, by default, the types of literal strings and numbers are NetRexx strings (type `Rexx`) and so variables tend to be of type `Rexx`. This simplifies the language and makes it easy to learn, as many useful programs can be written solely using the powerful `Rexx` type. Potentially more efficient (though less human-oriented) primitive or built-in types for literals will be used in binary classes (see page 80).

*If the examples above were in a binary class, then, in the reference implementation, the types of `s` and `j` would have been `java.lang.String` and `int` respectively.*

A variable may be introduced (“declared”) without giving it an initial value by simply assigning a type to it:

```
i=int
r=Rexx
f=java.io.File
```

Here, the expression to the right of the “=” simply evaluates to a type with no value.

### The use and scope of variables

NetRexx variables all follow the same rules of assignment, but are used in different contexts. These are:

- **Properties**: Variables which name the values (the data) owned by an object of the type defined by the class are called properties. When an object is constructed by the class, its properties are created and are initialized to either a default value (`null` or, for variables of primitive type, an implementation-defined value, typically 0) or to a value provided by the programmer.

  The attributes of properties can be changed by the properties instruction (see page 112). For example, properties may also be constant, which means that they are initialized when the class is first loaded and do not change thereafter.

- **Method arguments**: When a method is invoked, arguments may be passed to it. These method arguments are assigned to the variables named on the method instruction (see page 98) that introduces the method.

- **Local variables**: Variables that are known only within a method are called local variables; each time a method is invoked a distinct set of local variables is available. Local variables are normally given an initial value by the programmer. If they are not, they are initialized to a default value (`null` or, for variables of primitive type, an implementation-defined value, typically 0).

In order for types to be determined and type-checking to be possible at “compile-time”, and easily determined by inspection, the use and type of every variable is determined by its position in the
program, not by the order in which assignments are executed. That is, variable typing is static.

The visibility of a variable depends on its use. Properties are visible to all methods in a class; method arguments and local variables are only visible within the method in which they appear. In particular:

- Within a class, properties have unique names (they cannot be overridden by method arguments or by local variables within methods); this avoids error-prone ambiguity.
- Within a method, a method argument acts like a local variable (that is, it is in the same name-
  space as local variables, and can be assigned new values); it can be considered to be a local
  variable that is assigned a value just before the body of the method is executed. There cannot be
  both a method argument and a local variable in a method with the same name.
- Within methods, variables can take only one type, the type assigned to them when first
  encountered in the method (in a strict “physical” sense, that is, as parsed from top to bottom of
  the program and from left to right on each line). Since methods tend to be small, there is no
  local scoping of variables inside the constructs within a method.\(^{40}\)

Thus, in this example:

```
method iszero(x)
  if x=0 then qualifier='is zero'
  else qualifier='is not zero'
  say 'The argument' qualifier'.
```

the variable `qualifier` is known throughout the method and hence has a known type and
value when the `say` instruction is executed.

To summarize: a symbol that names a variable in the current class either refers to a property (and in
any use of it within the class refers to that property), or it refers to a variable that is unique within a
method (and any use of the name within that method refers to the same variable).

Note: A variable is just a name, or “handle” for a value. It is possible for more than one variable to
refer to the same value, as in the program:

```
first='A string'
second=first
```

Here, both variables refer to the same value. If that value is changeable then a change to the value
referred to by one of the variable names would also be seen if the value is referred to by the other. For
example, sub-values of a NetRexx string can be changed, using Indexed references (see page 73), so a
change to a sub-value of `first` would also be seen in an identical indexed reference to `second`.

**Terms on the left of assignments**

In an assignment instruction, the term to the left of the equals sign is most commonly a simple non-
numeric symbol, which always names a variable in the current class. The other possibilities, as seen in
the example below, are:

1. The term is an indexed reference (see page 73), to an existing variable that refers to a string of
type Rexx or an array (see page 74). The variable may be in the current class, or be a property
in a class named in the uses phrase of the class instruction for the current class.

2. The term is a compound term (see page 48) that ultimately refers to a property (see above) in
some class (which may be the current class). This property cannot be a constant.

\(^{40}\) Unlike the block scoping of PL/I, C, or Java.
Examples:

```plaintext
r=Rexx ''
r['foo']='?'          -- indexed string assignment
s=String[3]
s[0]='test'          -- array assignment
Sample.value=1       -- property assignment
this.value=1         -- property assignment
super.value=1        -- property assignment
```

The last two examples show assignments to a property in the current class or in a superclass of the current class, respectively. Note that references to properties in other classes must always be qualified in some way (for example, by the prefix `super.`). The use of the prefix `this.` for properties in the current class is optional.
Indexed strings and Arrays

Any NetRexx string (that is, a value of type Rexx), has the ability to have sub-values, values (also of type Rexx) which are associated with the original string and are indexed by an index string which identifies the sub-value. Any string with such sub-values is known as an indexed string.

The sub-values of a NetRexx string are accessed using indexed references, where the name of a variable of type Rexx is followed immediately by square brackets enclosing one or more expressions separated by commas:

```
symbol"[expression[, expression]...]]'
```

It is important to note that the symbol that names the variable must be followed immediately by the “[”, with no blank in between, or the construct will not be recognized as an indexed reference.

The expressions (separated by commas) between the brackets are called the indexes to the string. These index expressions are evaluated in turn from left to right, and each must evaluate to a value is of type Rexx or that can be converted to type Rexx.

The resulting index strings are taken “as-is” – that is, they must match exactly in content, case, and length for a reference to find a previously-set item. They may have any length (including the null string) and value (they are not constrained to be just those strings which are numbers, for example).

If a reference does not find a sub-value, then a copy of the non-indexed value of the variable is used.

Example:

```r
surname='Unknown'         -- default value
surname['Fred']='Bloggs'
surname['Davy']='Jones'
try='Fred'
say surname[try] surname['Bert']
```

would say “Bloggs Unknown”.

When multiple indexes are used, they indicate accessing a hierarchy of strings. A single NetRexx string has a single set of indexes and sub-values associated with it. The sub-values, however, are also NetRexx strings, and so may in turn have indexes and sub-values. When more than one index is specified in an indexed reference, the indexes are applied in turn from left to right to each retrieved sub-value.

For example, in the sequence:

```r
x='?'
x['foo', 'bar']='OK'
say x['foo', 'bar']
y=x['foo']
say y['bar']
```

both `say` instructions would display the string “OK”.

Indexed strings may be used to set up “associative arrays”, or dictionaries, in which the subscript is not necessarily numeric, and thus offer great scope for the creative programmer. A useful application is to set up a variable in which the subscripts are taken from the value of one or more variables, so effecting a form of associative (content addressable) memory.

---

41 The notations '[' and ']' indicate square brackets appearing in the NetRexx program.
Notes:

1. A variable of type Rexx must have been assigned a value before indexing is used on it. This is the value that is used as the default value whenever an indexed reference finds no sub-value.

2. The indexes, and hence the sub-values, of a Rexx object can be retrieved in turn using the over (see page 94) keyword of the loop instruction.

3. The exists method (see page 161) of the Rexx class may be used to test whether an indexed reference has an explicitly-set value.

4. Assigning null to an indexed reference (for example, the assignment switch[7]=null;) drops the sub-value; until set to a new value, any reference to the sub-value (including use of the exists method) will return the same result as when it had never been set.

Arrays

In addition to indexed strings, NetRexx also includes the concept of fixed-size arrays, which may be used for indexing values of any type (including strings).

Arrays are used with the same syntax and in the same manner as indexed strings, but with important differences that allow for compact implementations and access to equivalent data structures constructed using other programming languages:

1. The indexes for arrays must be whole numbers that are zero or positive. There will usually be an implementation restriction on the maximum value of the index (typically 999999999 or higher).

2. The elements of an array are considered to be ordered; the first element has index 0, the second 1, and so on.

3. An array is of fixed size; it must be constructed before use.

4. Variables that are assigned arrays can only be assigned arrays (of the same dimension, see below) in the future. That is, being an array changes the type of a value; it becomes a dimensioned type (see page 46).

Array references use the NetRexx indexed reference syntax defined above. The same syntax is used for constructing arrays, except that the symbol before the left bracket describes a type (and hence may be qualified by a package name). The expression or expressions between the brackets indicate the size of the array in each dimension, and must be a positive whole number or zero:

```
arg=String[4]       -- makes an array for four Strings
i=int[3]            -- makes an array for three 'int's
```

(Another way of describing this is that array constructors look just like other object constructors, except that brackets are used instead of parentheses.)

Once an array has been constructed, its elements can be referred to using brackets and expressions, as before:

```
i[2]=3       -- sets the '2'-indexed value of 'i'
j=i[2]       -- sets 'j' to the '2'-indexed value of 'i'
```

Regular multiple-dimensioned arrays may be constructed and referenced by using multiple expressions within the brackets:
i=int[2,3] -- makes a 2x3 array of 'int' type objects
i[1,2]=3   -- sets the '1,2'-indexed value of 'i'
j=i[1,2]   -- sets 'j' to the '1,2'-indexed value of 'i'

As with indexed strings, when multiple indexes are used, they indicate accessing a hierarchy of arrays
(the underlying model is therefore of a hierarchy of single-dimensioned arrays). When more than one
index is specified in an indexed reference to an array, the indexes are applied in turn from left to right
to each array.

As described in the section on Types (see page 46), the type of a variable that refers to an array can be
set (declared) by assignment of the type with array notation that indicates the dimension of an array
without any sizes:

k=int[]     -- one-dimensional array of 'int' objects
m=float[,]  -- 3-dimensional array of 'float' objects

The same syntax is also used when describing an array type in the arguments of a method instruction
or when converting types. For example, after:

gg=char[] "Horse"

the variable gg has as its value an array of type char[] containing the five characters H, o, r, s, and e.

**Array initializers**

An **array initializer** is a simple term which is recognized if it does not immediately follow (abut) a
symbol, and has the form:

```
['expression[,expression]...']
```

An array initializer therefore comprises a list of one or more expressions, separated by commas, within
brackets. When an array initializer is evaluated, the expressions are evaluated in turn from left to
right, and all must result in a value. An array is then constructed, with a number of elements equal to
the number of expressions in the list, with each element initialized by being assigned the result of the
corresponding expression.

The type of the array is derived by adding one dimension to the type of the result of the first
expression in the list, where the type of that expression is determined using the same rules as are used
to select the type of a variable when it is first assigned a value (see page 69). All the other expressions
in the list must have types that could be assigned to the chosen type without error.

For example, in

```
var1=['aa', 'bb', 'cc']
var2=[char 'a', 'b', 'c']
var3=[String 'a', 'bb', 'c']
var4=[[1, 2, 3, 4, 5, 6]]
```

the types of the variables would be Rexx[], char[], String[], Rexx[], and Rexx[], respectively.
In a binary class in the reference implementation, the types would be String[], char[], String[],
int[], and int[].

Array initializers are most useful for initializing properties and variables, but like other simple terms,
they may start a compound term.

---

42 The notations '[' and ']' indicate square brackets appearing in the NetRexx program.
So, for example

```
  say [1,1,1,1].length
```

would display 4.

Note that an array of length zero cannot be constructed with an array initializer, as its type would be undefined. An explicitly typed array constructor (for example, `int[0]`) must be used.
**Keyword Instructions**

A *keyword instruction* is one or more clauses, the first of which starts with a keyword that identifies the instruction. Some keyword instructions affect the flow of control; the remainder just provide services to the programmer. Some keyword instructions (do, if, loop, or select) can include nested instructions. Appendix A (see page 171) includes an example of a NetRexx program using many of the instructions available.

As can be deduced from the syntax rules described earlier, a keyword instruction is recognized *only* if its keyword is the first token in a clause, and if the second token is not an “=” character (implying an assignment). It would also not be recognized if the second token started with “(“, “[“, or “.” (implying that the first token starts a term).

Further, if a current local variable, method argument, or property has the same name as a keyword then the keyword will not be recognized. This important rule allows NetRexx to be extended with new keywords in the future without invalidating existing programs.

Thus, for example, this sequence in a program with no say variable:

```plaintext
say 'Hello'
say('1')
say=3
say 'Hello'
```

would be a say instruction, a call to some say method, an assignment to a say variable, and an error. In NetRexx, therefore, keywords are not reserved; they may be used as the names of variables (though this is not recommended, where known in advance).

Certain other keywords, known as *sub-keywords*, may be known within the clauses of individual instructions – for example, the symbols to and while in the loop instruction. Again, these are not reserved; if they had been used as names of variables, they would not be recognized as sub-keywords.

Blanks adjacent to keywords have no effect other than that of separating the keyword from the subsequent token. For example, this applies to the blanks next to the sub-keyword while in

```plaintext
loop while a=3
```

Here at least one blank was required to separate the symbols forming the keywords and the variable name, a. However the blank following the while is not necessary in

```plaintext
loop while 'Me'=a
```

though it does aid readability.
**Class instruction**

```netrexx
class name [visibility] [modifier] [binary] [deprecated]
    [extends classname]
    [uses useslist]
    [implements interfacelist];
```

where `visibility` is one of:

- private
- public
- shared

and `modifier` is one of:

- abstract
- adapter
- final
- interface

and `useslist` and `interfacelist` are lists of one or more `classnames`, separated by commas.

The `class` instruction is used to introduce a class, as described in the sections *Types and Classes* (see page 46) and *Program structure* (see page 124), and define its attributes. The class must be given a `name`, which must be different from the name of any other classes in the program. The `name`, which must be a non-numeric symbol, is known as the `short name` of the class.

A `classname` can be either the short name of a class (if that is unambiguous in the context in which it is used), or the qualified name of the class – the name of the class prefixed by a package name and a period, as described under the `package` instruction (see page 110).

The `body` of the class consists of all clauses following the class instruction (if any) until the next `class` instruction or the end of the program.

The `visibility`, `modifier`, and `binary` keywords, and the `extends`, `uses`, and `implements` phrases, may appear in any order.

**Visibility**

Classes may be `public`, `private`, or `shared`:

- A `public class` is visible to (that is, may be used by) all other classes.
- A `private class` is visible only within same program and to classes in the same package (see page 110).
- A `shared class` is also visible only within same program and to classes in the same package.

A program may have only one public class, and if no class is marked public then the first is assumed to be public (unless it is explicitly marked private).

---

43 The `shared` keyword on the `class` instruction means exactly the same as the keyword `private`, and is accepted for consistency with the other meanings of `shared`.
**Modifier**

Most classes are collections of data (properties) and the procedures that can act on that data (methods); they completely implement a datatype (type), and are permitted to be subclassed. These are called *standard classes*. The *modifier* keywords indicate that the class is not a standard class – it is special in some way. Only one of the following modifier keywords is allowed:

- **abstract**
  
  An *abstract class* does not completely implement a datatype; one or more of the methods that it defines (or which it inherits from classes it extends or implements) is abstract – that is, the name of the method and the types of its arguments are defined, but no instructions to implement the method are provided.

  Since some methods are not provided, an object cannot be constructed from an abstract class. Instead, the class must be extended and any missing methods provided. Such a subclass can then be used to construct an object.

  Abstract classes are useful where many subclasses can share common data or methods, but each will have some unique attribute or attributes (data and/or methods). For example, some set of geometric objects might share dimensions in X and Y, yet need unique methods for calculating the area of the object.

- **adapter**

  An *adapter class* is a class that is guaranteed to implement all unimplemented abstract methods of its superclasses and interface classes that it inherits or lists as implemented on the *class* instruction.

  If any unimplemented methods are found, they will be automatically generated by the language processor. Methods generated in this way will have the same visibility and signature as the abstract method they implement, and if a return value is expected then a default value is returned (as for the initial value of variables of the same type: that is, *null* or, for values of primitive type, an implementation-defined value, typically 0). Other than possibly returning a value, these methods are empty; that is, they have no side-effects.

  An adapter class provides a concrete representation of its superclasses and the interface classes it implements. As such, it is especially useful for implementing event handlers and the like, where only a small number of event-handling methods are needed but many more might be specified in the interface class that describes the event model.44

  An adapter class cannot have any abstract methods.

- **final**

  A *final class* is considered to be complete; it cannot be subclassed (extended), and all its methods are considered complete.45

- **interface**

  An *interface class* is an abstract class that contains only abstract method definitions and/or constants. That is, it defines neither instructions that implement methods nor modifiable properties, and hence cannot be used to construct an object.

  Interface classes are used by classes that claim to *implement* them (see the *implements* keyword, described below). The difference between abstract and interface classes is that the former may have methods which are not abstract, and hence can only be subclassed (extended), whereas the latter are wholly abstract and may only be implemented.

---

44 For example, see the “Scribble” sample in the NetRexx package.

45 This modifier is provided for consistency with other languages, and may allow compilers to improve the performance of classes that refer to the final class. In many cases it will reduce the reusability of the class, and hence should be avoided.
**Binary**

The keyword `binary` indicates that the class is a *binary class*. In binary classes, literal strings and numeric symbols are assigned native string or binary (primitive) types, rather than NetRexx types, and native binary operations are used to implement operators where possible. When `binary` is not in effect (the default), terms in expressions are converted to NetRexx types before use by operators. The section *Binary values and operations* (see page 149) describes the implications of binary classes in detail.

Individual methods in a class which is not binary can be made into *binary methods* using the `binary` keyword on the `method` instruction (see page 98).

**Deprecated**

The keyword `deprecated` indicates that the class is *deprecated*, which implies that a better alternative is available and documented. A compiler can use this information to warn of out-of-date or other use that is not recommended.

**Extends**

Classes form a hierarchy, with all classes (except the top of the tree, the `Object` class) being a *subclass* of some other class. The `extends` keyword identifies the `classname` of the immediate *superclass* of the new class – that is, the class immediately above it in the hierarchy. If no `extends` phrase is given, the superclass is assumed to be `Object` (or `null`, in the case where the current class is `Object`).

**Uses**

The `uses` keyword introduces a list of the names of one or more classes that will be used as a source of constant (or static) properties and/or methods.

When a term (see page 48) starts with a symbol, method call, or indexed reference that is not known in the current context, each class in the `useslist` and its superclasses are searched (in the order specified in the `useslist`) for a constant or static method or property that matches the item. If found, the method or property is used just as though explicitly qualified by the name of the class in which it was found.

The `uses` mechanism affects only the syntax of terms in the current class; it is not inherited by subclasses of the current class.

**Implements**

The `implements` keyword introduces a list of the names of one or more interface classes (see above). These interface classes are then known to (inherited by) the current class, in the order specified in the `interfacelist`. Their methods (which are all abstract) and constant properties act as though part of the current class, unless they are overridden (hidden) by a method or constant of the same name in the current class.

If the current class is not an interface class then it must implement (provide non-abstract methods for) all the methods inherited from the interface classes in the `implements` list.

Interface classes, therefore, can be used to:

1. Define a common set of methods (possibly with associated constants) that will be implemented

46 In the reference implementation, `java.lang.Object`. 
by other classes.

2. Conveniently package collections of constants for use by other classes. The implements list may not include the superclass of the current class.
**Do instruction**

```netrexx
do [label name] [protect term];
  instructionlist
  [catch [vare =] exception;
   instructionlist]...
  [finally[]
   instructionlist]
end [name];
```

where *name* is a non-numeric symbol

and *instructionlist* is zero or more instructions

The `do` instruction is used to group instructions together for execution; these are executed once. The group may optionally be given a label, and may protect an object while the instructions in the group are executed; exceptional conditions can be handled with `catch` and `finally`.

The most common use of `do` is simply for treating a number of instructions as group.

**Example:**

```netrexx
/* The two instructions between DO and END will both */
/* be executed if A has the value 3. */
if a=3 then do
  a=a+2
  say 'Smile!'
end
```

Here, only the first *instructionlist* is used. This forms the body of the group.

The instructions in the *instructionlists* may be any assignment, method call, or keyword instruction, including any of the more complex constructions such as `loop`, `if`, `select`, and the `do` instruction itself.

**Label phrase**

If `label` is used to specify a *name* for the group, then a `leave` which specifies that name may be used to leave the group, and the `end` that ends the group may optionally specify the name of the group for additional checking.

**Example:**

```netrexx
do label sticky
  x=ask
  if x='quit' then leave sticky
  say 'x was' x
end sticky
```

**Protect phrase**

If `protect` is given it must be followed by a *term* that evaluates to a value that is not just a type and is not of a primitive type; while the `do` construct is being executed, the value (object) is protected – that is, all the instructions in the `do` construct have exclusive access to the object.

Both `label` and `protect` may be specified, in any order, if required.
**Exceptions in do groups**

Exceptions that are raised by the instructions within a do group may be caught using one or more **catch** clauses that name the *exception* that they will catch. When an exception is caught, the exception object that holds the details of the exception may optionally be assigned to a variable, `vare`.

Similarly, a **finally** clause may be used to introduce instructions that will always be executed at the end of the group, even if an exception is raised (whether caught or not).

The *Exceptions* section (see page 152) has details and examples of **catch** and **finally**.
Exit instruction

```
exit [expression];
```

`exit` is used to unconditionally leave a program, and optionally return a result to the caller. The entire program is terminated immediately.

If an `expression` is given, it is evaluated and the result of the evaluation is then passed back to the caller in an implementation-dependent manner when the program terminates. Typically this value is expected to be a small whole number; most implementations will accept values in the range 0 through 250. If no expression is given, a default result (which depends on the implementation, and is typically zero) is passed back to the caller.

**Example:**

```netrexx
j=3
exit j*4
/* Would exit with the value '12' */
```

“Running off the end” of a program is equivalent to the instruction `return;` . In the case where the program is simply a stand-alone application with no `class` or `method` instructions, this has the same effect as `exit;`, in that it terminates the whole program and returns a default result.
### If instruction

```plaintext
if expression[;]
  then[;] instruction
[else[;] instruction]
```

The `if` construct is used to conditionally execute an instruction or group of instructions. It can also be used to select between two alternatives.

The expression is evaluated and must result in either 0 or 1. If the result was 1 (true) then the instruction after the `then` is executed. If the result was 0 (false) and an `else` was given then the instruction after the `else` is executed.

**Example:**

```plaintext
if answer='Yes' then say 'OK!'
else say 'Why not?'
```

Remember that if the `else` clause is on the same line as the last clause of the `then` part, then you need a semicolon to terminate that clause.

**Example:**

```plaintext
if answer='Yes' then say 'OK!'; else say 'Why not?'
```

The `else` binds to the nearest `then` at the same level. This means that any `if` that is used as the instruction following the `then` in an `if` construct that has an `else` clause, must itself have an `else` clause (which may be followed by the dummy instruction, `nop`).

**Example:**

```plaintext
if answer='Yes' then if name='Fred' then say 'OK, Fred.'
else say 'OK.'
else say 'Why not?'
```

To include more than one instruction following `then` or `else`, use a grouping instruction (`do, loop, or select`).

**Example:**

```plaintext
if answer='Yes' then do
  say 'Line one of two'
say 'Line two of two'
end
```

In this instance, both `say` instructions are executed when the result of the `if` expression is 1.

Multiple expressions, separated by commas, can be given on the `if` clause, which then has the syntax:

```plaintext
if expression[, expression]... [:]
```

In this case, the expressions are evaluated in turn from left to right, and if the result of any evaluation is 1 then the test has succeeded and the instruction following the associated `then` clause is executed. If all the expressions evaluate to 0 and an `else` was given then the instruction after the `else` is executed.

Note that once an expression evaluation has resulted in 1, no further expressions in the clause are evaluated. So, for example, in:

```plaintext
-- assume 'name' is a string
if name=null, name='' then say 'Empty'
```
then if name does not refer to an object it will compare equal to null and the say instruction will be executed without evaluating the second expression in the if clause.

Notes:

1. An instruction may be any assignment, method call, or keyword instruction, including any of the more complex constructions such as do, loop, select, and the if instruction itself. A null clause is not an instruction, however, so putting an extra semicolon after the then or else is not equivalent to putting a dummy instruction. The nop instruction is provided for this purpose.

2. The keyword then is treated specially, in that it need not start a clause. This allows the expression on the if clause to be terminated by the then, without a “;” being required – were this not so, people used to other computer languages would be inconvenienced. Hence the symbol then cannot be used as a variable name within the expression.47

---

47 Strictly speaking, then should only be recognized if not the name of a variable. In this special case, however, NetRexx language processors are permitted to treat then as reserved in the context of an if clause, to provide better performance and more useful error reporting.
**Import instruction**

```import name;```

where `name` is one or more non-numeric symbols separated by periods, with an optional trailing period.

The `import` instruction is used to simplify the use of classes from other packages. If a class is identified by an `import` instruction, it can then be referred to by its short name, as given on the `class` instruction (see page 78), as well as by its fully qualified name.

There may be zero or more `import` instructions in a program. They must precede any `class` instruction (or any instruction that would start the default class).

In the following description, a `package name` names a package as described under the `package` instruction (see page 110). The `import name` must be one of:

- A qualified class name, which is a package name immediately followed by a period which is immediately followed by a short class name – in this case, the individual class identified is imported.
- A package name – in this case, all the classes in the specified package are imported. The name may have a trailing period.
- A partial package name (a package name with one or more parts omitted from the right, indicated by a trailing period after the parts that are present) – in this case, all classes in the package hierarchy below the specified point are imported.

Examples:

```import java.lang.String```
```import java.lang```
```import java.```

The first example above imports a single class (which could then be referred to simply as “String”). The second example imports all classes in the “java.lang” package. The third example imports all classes in all the packages whose name starts with “java.”.

When a class is imported explicitly, for example, using

```import java.awt.List```

this indicates that the short name of the class (List, in this example) may be used to refer to the class unambiguously. That is, using this short name will not report an ambiguous reference warning (as it would without the `import` instruction, because a `java.util.List` class was added in Java 1.2).

It follows that:

- Two classes imported explicitly cannot have the same short name.
- No class in a program being compiled can have the same short name as a class that is imported explicitly.

because in either of these situations a use of the short name would be ambiguous.

Note also that an explicit import does not import the minor or dependent classes associated with a name; they each require their own explicit import (unless the entire package is imported).
In the reference implementation, the fundamental NetRexx and Java package hierarchies are automatically imported by default, as though the instructions:

```plaintext
import netrexx.lang.
import java.lang.
import java.io.
import java.util.
import java.net.
import java.awt.
import java.applet.
```

had been executed before the program begins. In addition, classes in the current (working) directory are imported if no `package` instruction is specified. If a `package` instruction is specified then all classes in that package are imported.
**Iterate instruction**

```
iterate [name];
```

where *name* is a non-numeric symbol.

**iterate** alters the flow of control within a **loop** construct. It may only be used in the body (the first *instructionlist*) of the construct.

Execution of the instruction list stops, and control is passed directly back up to the **loop** clause just as though the last clause in the body of the construct had just been executed. The control variable (if any) is then stepped (iterated) and termination conditions tested as normal and the instruction list is executed again, unless the loop is terminated by the **loop** clause.

If no *name* is specified, then **iterate** will step the innermost active loop.

If a *name* is specified, then it must be the name of the label, or control variable if there is no label, of a currently active loop (which may be the innermost), and this is the loop that is iterated. Any active **do**, **loop**, or **select** constructs inside the loop selected for iteration are terminated (as though by a **leave** instruction).

**Example:**

```
loop i=1 to 4
  if i=2 then iterate i
  say i
end
/* Would display the numbers:  1, 3, 4  */
```

**Notes:**

1. A loop is active if it is currently being executed. If a method (even in the same class) is called during execution of a loop, then the loop becomes inactive until the method has returned. **iterate** cannot be used to step an inactive loop.

2. The *name* symbol, if specified, must exactly match the label (or the name of the control variable, if there is no label) in the **loop** clause in all respects except case.
**Leave instruction**

```
leave [name];
```

where `name` is a non-numeric symbol.

**leave** causes immediate exit from one or more **do**, **loop**, or **select** constructs. It may only be used in the body (the first instructionlist) of the construct.

Execution of the instruction list is terminated, and control is passed to the **end** clause of the construct, just as though the last clause in the body of the construct had just been executed or (if a loop) the termination condition had been met normally, except that on exit the control variable (if any) will contain the value it had when the **leave** instruction was executed.

If no `name` is specified, then **leave** must be within an active loop and will terminate the innermost active loop.

If a `name` is specified, then it must be the name of the label (or control variable for a loop with no label), of a currently active **do**, **loop**, or **select** construct (which may be the innermost). That construct (and any active constructs inside it) is then terminated. Control then passes to the clause following the **end** clause that matches the **do**, **loop**, or **select** clause identified by the `name`.

**Example:**

```
loop i=1 to 5
  say i
  if i=3 then leave
end i
/* Would display the numbers:  1, 2, 3 */
```

**Notes:**

1. If any construct being left includes a **finally** clause, the instructionlist following the **finally** will be executed before the construct is left.

2. A **do**, **loop**, or **select** construct is active if it is currently being executed. If a method (even in the same class) is called during execution of an active construct, then the construct becomes inactive until the method has returned. **leave** cannot be used to leave an inactive construct.

3. The `name` symbol, if specified, must exactly match the label (or the name of the control variable, for a loop with no label) in the **do**, **loop**, or **select** clause in all respects except case.
# Loop instruction

```netrexx
loop [label name] [protect term] [repetitor] [conditional];
    instructionlist
    [catch [vare =] exception;
        instructionlist]...
    [finally[];
        instructionlist]
end [name];
```

where `repetitor` is one of:

- `varc = expr [to exprt] [by exprb] [for exprf]`
- `varo over termo`
- `forever`

and `conditional` is either of:

- `while exprw`
- `until expru`

and `name` is a non-numeric symbol

and `instructionlist` is zero or more instructions

and `expri`, `exprt`, `exprb`, `exprf`, `exprr`, `exprw`, and `expru` are expressions.

The `loop` instruction is used to group instructions together and execute them repetitively. The loop may optionally be given a label, and may protect an object while the instructions in the loop are executed; exceptional conditions can be handled with `catch` and `finally`.

`loop` is the most complicated of the NetRexx keyword instructions. It can be used as a simple indefinite loop, a predetermined repetitive loop, as a loop with a bounding condition that is recalculated on each iteration, or as a loop that steps over the contents of a collection of values.

**Syntax notes:**

- The `label` and `protect` phrases may be in any order. They must precede any `repetitor` or `conditional`.
- The first `instructionlist` is known as the `body` of the loop.
- The `to`, `by`, and `for` phrases in the first form of `repetitor` may be in any order, if used, and will be evaluated in the order they are written.
- Any instruction allowed in a method is allowed in an `instructionlist`, including assignments, method call instructions, and keyword instructions (including any of the more complex constructions such as `if`, `do`, `select`, or the `loop` instruction itself).
- If `for` or `forever` start the `repetitor` and are followed by an “=” character, they are taken as
control variable names, not keywords (as for assignment instructions).

- The expressions expri, exprt, exprb, or exprf will be ended by any of the keywords to, by, for, while, or until (unless the word is the name of a variable).
- The expressions exprw or expru will be ended by either of the keywords while or until (unless the word is the name of a variable).

**Indefinite loops**

If neither repetitor nor conditional are present, or the repetitor is the keyword forever, then the loop is an indefinite loop. It will be ended only when some instruction in the first instructionlist causes control to leave the loop.

**Example:**

``` NetRexx
/* This displays "Go caving!" at least once */
loop forever
  say 'Go caving!'
  if ask='' then leave
end
```

**Bounded loops**

If a repetitor (other than forever) or conditional is given, the first instructionlist forms a bounded loop, and the instruction list is executed according to any repetitor phrase, optionally modified by a conditional phrase.

**Simple bounded loops**

When the repetitor starts with the keyword for, the expression exprr is evaluated immediately (with 0 added, to effect any rounding) to give a repetition count, which must be a whole number that is zero or positive. The loop is then executed that many times, unless it is terminated by some other condition.

**Example:**

``` NetRexx
/* This displays "Hello" five times */
loop for 5
  say 'Hello'
end
```

**Controlled bounded loops**

A controlled loop begins with an assignment, which can be identified by the “=” that follows the name of a control variable, varc. The control variable is assigned an initial value (the result of expri, formatted as though 0 had been added) before the first execution of the instruction list. The control variable is then stepped (by adding the result of exprb) before the second and subsequent times that the instruction list is executed.

The name of the control variable, varc, must be a non-numeric symbol that names an existing or new variable in the current method or a property in the current class (that is, it cannot be element of an array, the property of a superclass, or a more complex term). It is further restricted in that it must not already be used as the name of a control variable or label in a loop (or do or select construct) that encloses the new loop.

The instruction list in the body of the loop is executed repeatedly while the end condition (determined by the result of exprt) is not met. If exprb is positive or
zero, then the loop will be terminated when \textit{varc} is greater than the result of \textit{exprt}. If negative, then the loop will be terminated when \textit{varc} is less than the result of \textit{exprt}.

The expressions \textit{exprt} and \textit{exprb} must result in numbers. They are evaluated once only (with 0 added, to effect any rounding), in the order they appear in the instruction, and before the loop begins and before \textit{expri} (which must also result in a number) is evaluated and the control variable is set to its initial value.

The default value for \textit{exprb} is 1. If no \textit{exprt} is given then the loop will execute indefinitely unless it is terminated by some other condition.

\textbf{Example:}

\begin{verbatim}
loop i=3 to -2 by -1
  say i
end
/* Would display: 3, 2, 1, 0, -1, -2 */
\end{verbatim}

Note that the numbers do not have to be whole numbers:

\textbf{Example:}

\begin{verbatim}
x=0.3
loop y=x to x+4 by 0.7
  say y
end
/* Would display: 0.3, 1.0, 1.7, 2.4, 3.1, 3.8 */
\end{verbatim}

The control variable may be altered within the loop, and this may affect the iteration of the loop. Altering the value of the control variable in this way is normally considered to be suspect programming practice, though it may be appropriate in certain circumstances.

Note that the end condition is tested at the start of each iteration (and after the control variable is stepped, on the second and subsequent iterations). It is therefore possible for the body of the loop to be skipped entirely if the end condition is met immediately.

The execution of a controlled loop may further be bounded by a \texttt{for} phrase. In this case, \textit{exprf} must be given and must evaluate to a non-negative whole number. This acts just like the repetition count in a simple bounded loop, and sets a limit to the number of iterations around the loop if it is not terminated by some other condition.

\textit{exprf} is evaluated along with the expressions \textit{exprt} and \textit{exprb}. That is, it is evaluated once only (with 0 added), when the \texttt{loop} instruction is first executed and before the control variable is given its initial value; the three expressions are evaluated in the order in which they appear. Like the \texttt{to} condition, the \texttt{for} count is checked at the start of each iteration, as shown in the programmer’s model (see page 96).

\textbf{Example:}

\begin{verbatim}
loop y=0.3 to 4.3 by 0.7 for 3
  say y
\end{verbatim}
In a controlled loop, the symbol that describes the control variable may be specified on the end clause (unless a label is specified, see below). NetRexx will then check that this symbol exactly matches the varc of the control variable in the loop clause (in all respects except case). If the symbol does not match, then the program is in error – this enables the nesting of loops to be checked automatically.

Example:

```rarex
loop k=1 to 10
  ...
  ...
end k /* Checks this is the END for K loop */
```

Note: The values taken by the control variable may be affected by the numeric settings, since normal NetRexx arithmetic rules apply to the computation of stepping the control variable.

**Over loops**

When the second token of the repetitor is the keyword over, the control variable, varo, is used to work through the sub-values in the collection of indexed strings identified by termo. In this case, the loop instruction takes a “snapshot” of the indexes that exist in the collection at the start of the loop, and then for each iteration of the loop the control variable is set to the next available index from the snapshot.

The number of iterations of the loop will be the number of indexes in the collection, unless the loop is terminated by some other condition.

Example:

```rarex
mycoll=''
mycoll['Tom']=1
mycoll['Dick']=2
mycoll['Harry']=3
loop name over mycoll
  say mycoll[name]
end
/* might display: 3, 1, 2 */
```

Notes:

1. The order in which the values are returned is undefined; all that is known is that all indexes available when the loop started will be recorded and assigned to varo in turn as the loop iterates.
2. The same restrictions apply to varo as apply to varc, the control variable for controlled loops (see above).
3. Similarly, the symbol varo may be used as a name for the loop and be specified on the end clause (unless a label is specified, see below).

In the reference implementation, the over form of repetitor may also be used to step though the contents of any object that is of a type that is a subclass of java.util.Dictionary, such as an object of type java.util.Hashtable. In this case, termo specifies the dictionary, and a snapshot (enumeration) of the keys to the Dictionary is taken at the start of the loop. Each iteration of the loop then assigns a new key to the control variable varo which must be (or will be
Conditional phrases

Any of the forms of loop syntax can be followed by a *conditional* phrase which may cause termination of the loop.

If **while** is specified, *exprw* is evaluated, using the latest values of all variables in the expression, before the instruction list is executed on every iteration, and after the control variable (if any) is stepped. The expression must evaluate to either 0 or 1, and the instruction list will be repeatedly executed while the result is 1 (that is, the loop ends if the expression evaluates to 0).

**Example:**
```
loop i=1 to 10 by 2 while i<6
    say i
end
/* Would display: 1, 3, 5 */
```

If **until** is specified, *expru* is evaluated, using the latest values of all variables in the expression, on the second and subsequent iterations, and before the control variable (if any) is stepped. The expression must evaluate to either 0 or 1, and the instruction list will be repeatedly executed until the result is 1 (that is, the loop ends if the expression evaluates to 1).

**Example:**
```
loop i=1 to 10 by 2 until i>6
    say i
end
/* Would display: 1, 3, 5, 7 */
```

Note that the execution of loops may also be modified by using the **iterate** or **leave** instructions.

Label phrase

The **label** phrase may used to specify a *name* for the loop. The name can then optionally be used on

- a **leave** instruction, to specify the name of the loop to leave
- an **iterate** instruction, to specify the name of the loop to be iterated
- the **end** clause of the loop, to confirm the identity of the loop that is being ended, for additional checking.

**Example:**
```
loop label pooks i=1 to 10
    loop label hill while j<3
        ...
        if a=b then leave pooks
        ...
    end hill
end pooks
```

In this example, the **leave** instruction leaves both loops.

If a label is specified using the **label** keyword, it overrides any name derived from the control variable name (if any). That is, the variable name cannot be used to refer to the loop if a label is specified.

---

48 Thus, it appears that the **until** condition is tested after the instruction list is executed on each iteration. However, it is the **loop** clause that carries out the evaluation.
**Protect phrase**

The `protect` phrase may be used to specify a term, `termp`, that evaluates to a value that is not just a type and is not of a primitive type; while the `loop` construct is being executed, the value (object) is protected – that is, all the instructions in the `loop` construct have exclusive access to the object.

**Example:**

```
loop protect myobject while a<b
...%
end
```

Both `label` and `protect` may be specified, in any order, if required.

**Exceptions in loops**

Exceptions that are raised by the instructions within a `loop` construct may be caught using one or more `catch` clauses that name the `exception` that they will catch. When an exception is caught, the exception object that holds the details of the exception may optionally be assigned to a variable, `vare`.

Similarly, a `finally` clause may be used to introduce instructions that will always be executed when the loop ends, even if an exception is raised (whether caught or not).

The Exceptions section (see page 152) has details and examples of `catch` and `finally`.

**Programmer's model – how a typical loop is executed**

This model forms part of the definition of the `loop` instruction.

For the following loop:

```
loop varc = expri to exprt by exprb while exprw
...%
instruction list
...%
end
```

NetRexx will execute the following:

```
$tempt=exprt+0 /* ($variables are internal and */
$tempb=exprb+0 /* are not accessible.) */
varc=expri+0
Transfer control to the point identified as $start:

$loop:
/* An UNTIL expression would be tested here, with: */
/* if expru then leave */
varc=varc + $tempb
$start:
if varc > $tempt then leave /* leave quits a loop */
/* A FOR count would be checked here */
if \exprw then leave
...%
instruction list
...%
Transfer control to the point identified as $loop:
```

**Notes:**

1. This example is for `exprb >= 0`. For a negative `exprb`, the test at the start point of the loop
would use “<” rather than “>”.

2. The upwards transfer of control takes place at the end of the body of the loop, immediately before the end clause (or any catch or finally clause). The end clause is only reached when the loop is finally completed.
Method instruction

```
method name[[arglist]]
  [visibility] [modifier] [protect] [binary] [deprecated]
  [returns termr]
  [signals signallist];
```

where `arglist` is a list of one or more `assignments`, separated by commas

and `visibility` is one of:

- inheritable
- private
- public
- shared

and `modifier` is one of:

- abstract
- constant
- final
- native
- static

and `signallist` is a list of one or more `terms`, separated by commas.

The `method` instruction is used to introduce a method within a class, as described in `Program structure` (see page 124), and define its attributes. The method must be given a `name`, which must be a non-numeric symbol. This is its `short name`.

If the short name of a method matches the short name of the class in which it appears, it is a `constructor method`. Constructor methods are used for constructing values (objects), and are described in detail in `Methods and Constructors` (see page 53).

The `body` of the method consists of all clauses following the method instruction (if any) until the next `method` or `class` instruction, or the end of the program.

The `visibility`, `modifier`, and `protect` keywords, and the `returns` and `signals` phrases, may appear in any order.

**Arguments**

The `arglist` on a `method` instruction, immediately following the method name, is optional and defines a list of the arguments for the method. An `argument` is a value that was provided by the caller when the method was invoked.

If there are no arguments, this may optionally be indicated by an “empty” pair of parentheses.

In the `arglist`, each argument has the syntax of an `assignment` (see page 69), where the “=” and the following `expression` may be omitted. The name in the assignment provides the name for the argument (which must not be the same as the name of any property in the class). Each argument is also
optionally assigned a type, or type and default value, following the usual rules of assignment. If there is no assignment, the argument is assigned the NetRexx string type, Rexx.

If there is no assignment (that is, there is no “=”) or the expression to the right of the “=” returns just a type, the argument is required (that is, it must always be specified by the caller when the method is invoked).

If an explicit value is given by the expression then the argument is optional; when the caller does not provide an argument in that position, then the expression is evaluated when the method is invoked and the result is provided to the method as the argument.

Optional arguments may be omitted “from the right” only. That is, arguments may not be omitted to the left of arguments that are not omitted.

Examples:

```netrexx
method fred
method fred()
method fred(width, height)
method fred(width=int, height=int 10)
```

In these examples, the first two `method` instructions are equivalent, and take no arguments. The third example takes two arguments, which are both strings of type Rexx. The final example takes two arguments, both of type int; the second argument is optional, and if not supplied will default to the value 10 (note that any valid expression could be used for the default value).

Visibility

Methods may be public, inheritable, private, or shared:

- A **public method** is visible to (that is, may be used by) all other classes to which the current class is visible.
- An **inheritable method** is visible to (that is, may be used by) all classes in the same package and also those classes that extend (that is, are subclasses of) the current class.
- A **private method** is visible only within the current class.
- A **shared method** is visible within the current package but is not visible outside the package. Shared methods cannot be inherited by classes outside the package.

By default (i.e., if no visibility keyword is specified), methods are public.

Modifier

Most methods consist of instructions that follow the `method` instruction and implement the method; the method is associated with an object constructed by the class. These are called standard methods. The modifier keywords define that the method is not a standard method – it is special in some way. Only one of the following modifier keywords is allowed:

- **abstract**  An abstract method has the name of the method and the types (but not values) of its arguments defined, but no instructions to implement the method are provided (or permitted).

  If a class contains any abstract methods, an object cannot be constructed from it, and so the class itself must be abstract; the abstract keyword must be present on the class.
instruction (see page 78).

Within an interface class, the abstract keyword is optional on the methods of the class, as all methods must be abstract. No other modifier is allowed on the methods of an interface class.

**constant**

A constant method is a static method that cannot be overridden by a method in a subclass of the current class. That is, it is both final and static (see below).

**final**

A final method is considered to be complete; it cannot be overridden by a subclass of the current class. private methods are implicitly final.49

**native**

A native method is a method that is implemented by the environment, not by instructions in the current class. Such methods have no NetRexx instructions to implement the method (and none are permitted), and they cannot be overridden by a method in a subclass of the current class.

Native methods are used for accessing primitive operations provided by the underlying operating system or by implementation-dependent packages.

**static**

A static method is a method that is not a constructor and is associated with the class, rather than with an object constructed by the class. It cannot use properties directly, except those that are also static (or constant).

Static methods may be invoked in the following ways:

1. Within the initialization expression of a static or constant property (such methods are invoked when the class is first loaded).

2. By qualifying the name of the method with the name of its class (qualified by the package name if necessary), for example, using “Math.Sin(1.3)” or “java.lang.Math.Sin(1.3)”. Methods called in this way are in effect functions.

3. By implicitly qualifying the name by including the name of its class in the uses phrase in the class instruction for the current class. Static methods in classes listed in this way can be used directly, without qualification, for example, as “Sin(1.3)”. They may be still be qualified, if preferred.

   In the reference implementation, stand-alone applications are started by the java command invoking a static method called main which takes a single argument (of type java.lang.String[]) and returns no result.

**Protect**

The keyword protect indicates that the method protects the current object (or class, for a static method) while the instructions in the method are executed. That is, the instructions in the method have exclusive access to the object; if some other method (or construct) is executing in parallel with the invocation of the method and is protecting the same object then the method will not start execution until the object is no longer protected.

Note that if a method or construct protecting an object invokes a method (or starts a new construct) that protects the same object then execution continues normally. The inner method or construct is not

---

49 This modifier may allow compilers to improve the performance of methods that are final, but may also reduce the reusability of the class.
prevented from executing, because it is not executing in parallel.

**Binary**

The keyword `binary` indicates that the method is a binary method.

In binary methods, literal strings and numeric symbols are assigned native string or binary (primitive) types, rather than NetRexx types, and native binary operations are used to implement operators where possible. When `binary` is not in effect (the default), terms in expressions are converted to NetRexx types before use by operators. The section *Binary values and operations* (see page 149) describes the implications of binary methods and classes in detail.

**Notes:**

1. Only the instructions inside the body of the method are affected by the `binary` keyword; any arguments and expressions on the method instruction itself are not affected (this ensures that a single rule applies to all the method signatures in a class).

2. All methods in a binary class are binary methods; the `binary` keyword on methods is provided for classes in which only the occasional method needs to be binary (perhaps for performance reasons). It is not an error to specify `binary` on a method in a binary class.

**Deprecated**

The keyword `deprecated` indicates that the method is deprecated, which implies that a better alternative is available and documented. A compiler can use this information to warn of out-of-date or other use that is not recommended.

Note that individual methods in interface classes cannot be deprecated; the whole class should be deprecated in this case.

**Returns**

The `returns` keyword is followed by a term, `termr`, that must evaluate to a type. This type is used to define the type of values returned by `return` instructions within the method.

The `returns` phrase is only required if the method is to return values of a type that is not NetRexx strings (class `Rexx`). If `returns` is specified, all `return` instructions (see page 114) within the method must specify an expression.

**Example:**

```
method filer(path, name) returns File
  return File(path||name)
```

This method always returns a value which is a `File` object.

**Signals**

The `signals` keyword introduces a list of terms that evaluate to types that are Exceptions (see page 152). This list enumerates and documents the exceptions that are signalled within the method (or by a method which is called from the current method) but are not caught by a `catch` clause in a control construct.
Example:

```netrexx
method soup(cat) signals IOException, DivideByZero
```

It is considered good programming practice to use this list to document “unusual” exceptions signalled by a method. Implementations that support the concept of checked exceptions (see page 153) must report as an error any checked exception that is incorrectly included in the list (that is, if the exception is never signalled or would always be caught). Such implementations may also offer an option that enforces the listing of all or some checked exceptions.

**Duplicate methods**

Methods may not duplicate properties or other methods in the same class. Specifically:

- The short name of a method must not be the same as the name of any property in the same class.
- The number (zero or more) and types of the arguments of a method (or any subset permitted by omitting optional arguments) must not be the same as those of any other method of the same name in the class (also checking any subset permitted by omitting optional arguments).

Note that the second rule does allow multiple methods with the same name in a class, provided that the number of arguments differ or at least one argument differs in type.
**Nop instruction**

**nop**: 

**nop** is a dummy instruction that has no effect. It can be useful as an explicit “do nothing” instruction following a **then** or **else** clause.

**Example:**

```rxml
select
  when a=b then nop           -- Do nothing
  when a>b then say 'A > B'
  otherwise     say 'A < B'
end
```

**Note:** Putting an extra semicolon instead of the **nop** would merely insert a null clause, which would just be ignored by NetRexx. The second **when** clause would then immediately follow the **then**, and hence would be reported as an error. **nop** is a true instruction, however, and is therefore a valid target for the **then** clause.
**Numeric instruction**

```plaintext
numeric digits [exprd];
form [scientific];
[engineering];
```

The `numeric` instruction is used to change the way in which arithmetic operations are carried out by a program. The effects of this instruction are described in detail in the section on *Numbers and Arithmetic* (see page 140).

**numeric digits** controls the precision under which arithmetic operations will be evaluated. If no expression `exprd` is given then the default value of 9 is used. Otherwise the result of the expression is rounded, if necessary, according to the current setting of `numeric digits` before it is used. The value used must be a positive whole number.

There is normally no limit to the value for `numeric digits` (except the constraints imposed by the amount of storage and other resources available) but note that high precisions are likely to be expensive in processing time. It is recommended that the default value be used wherever possible.

Note that small values of `numeric digits` (for example, values less than 6) are generally only useful for very specialized applications. The setting of `numeric digits` affects all computations, so even the operation of loops may be affected by rounding if small values are used.

If an implementation does not support a requested value for `numeric digits` then the instruction will fail with an exception (which may, as usual, be caught with the `catch` clause of a control construct).

The current setting of `numeric digits` may be retrieved with the `digits` special word (see page 130).

**numeric form** controls which form of exponential notation (see page 147) is to be used for the results of operations. This may be either `scientific` (in which case only one, non-zero, digit will appear before the decimal point), or `engineering` (in which case the power of ten will always be a multiple of three, and the part before the decimal point will be in the range 1 through 999). The default notation is scientific.

The form is set directly by the sub-keywords `scientific` or `engineering`; if neither sub-keyword is given, `scientific` is assumed. The current setting of `numeric form` may be retrieved with the `form` special word (see page 130).

If an implementation does not support a requested value for `numeric form` then the instruction will fail with an exception (which may, as usual, be caught with the `catch` clause of a control construct).

The `numeric` instruction may be used where needed as a dynamically executed instruction in a method.

It may also appear, more than once if necessary, before the first method in a class, in which case it forms the default setting for the initialization of subsequent properties in the class and for all methods in the class. In this case, any exception due to the `numeric` instruction is raised when the class is first loaded.
Further, one or more `numeric` instructions may be placed before the first `class` instruction in a program; they do not imply the start of a class. The numeric settings then apply to all classes in the program (except interface classes), as though the `numeric` instructions were placed immediately following the `class` instruction in each class (except that they will not be traced).
Options instruction

```
options wordlist;
```

where `wordlist` is one or more symbols separated by blanks.

The `options` instruction is used to pass special requests to the language processor (for example, an interpreter or compiler).

Individual words, known as `option words`, in the `wordlist` which are meaningful to the language processor will be obeyed (these might control optimizations, enforce standards, enable implementation-dependent features, etc.); those which are not recognized will be ignored (they are assumed to be instructions to a different language processor). Option words in the list that are known will be recognized independently of case.

There may be zero or more `options` instructions in a program. They apply to the whole program, and must come before the first `class` instruction (or any instruction that starts a class).

In the reference implementation, the known option words are:

- **binary**
  All classes in this program will be binary classes (see page 80). In binary classes, literals are assigned binary (primitive) or native string types, rather than NetRexx types, and native binary operations are used to implement operators where appropriate, as described in “Binary values and operations” (see page 149). In classes that are not binary, terms in expressions are converted to the NetRexx string type, Rexx, before use by operators.

- **comments**
  Comments from the NetRexx source program will be passed through to the the Java output file (which may be saved with a `.java.keep` extension by using the `-keep` command option).

  Line comments become Java line comments (introduced by `//`). Block comments become Java block comments (delimited by `/*` and `*/`), with nested block comments having their delimiters changed to `(-` and `-)`.

- **compact**
  Requests that warnings and error messages be displayed in compact form. This format is more easily parsed than the default format, and is intended for use by editing environments.

  Each error message is presented as a single line, prefixed with the error token identification enclosed in square brackets. The error token identification comprises three words, with one blank separating the words. The words are: the source file specification, the line number of the error token, the column in which it starts, and its length. For example (all on one line):

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

  Any blanks in the file specification are replaced by a null (`'\0'`) character. Additional words could be added to the error token identification later.
console  Requests that compiler messages be written to console (the default). Use -noconsole to prevent messages being written to the console.

This option only has an effect as a compiler option, and applies to all programs being compiled.

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

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

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

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

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

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

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

replace  Requests that replacement of the translator output (.java) file be allowed. The default, noreplace, prevents an existing .java file being accidentally overwritten.

savelog  Requests that compiler messages be written to the file NetRexxC.log in the current directory. The messages are also displayed on the console, unless -noconsole is specified.

This option only has an effect as a compiler option, and applies to all programs being compiled.

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

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

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

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

**strictimport** Requires that all imported packages and classes be imported explicitly using `import` instructions. That is, if in effect, there will be no automatic imports (see page 87), except those related to the `package` instruction.

This option only has an effect as a compiler option, and applies to all programs being compiled.

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

**strictsignal** Requires that all checked exceptions (see page 153) signalled within a method but not caught by a `catch` clause be listed in the `signals` phrase of the method instruction.

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

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

If `notrace` is given, then trace instructions are ignored. The latter can be useful to prevent tracing overheads while leaving `trace` instructions in a program.

**utf8** If given, clauses following the `options` instruction are expected to be encoded using UTF-8, so all Unicode characters may be used in the source of the program.

In UTF-8 encoding, Unicode characters less than `'\u0080'` are represented using one byte (whose most-significant bit is 0), characters in the range `'\u0080' through `'\u07FF'` are encoded as two bytes, in the sequence of bits:

```
 110xxxxx  10xxxxxx
```

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

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

where the sixteen digits shown as x are the sixteen bits of the character.

If `noutf8` is given, following clauses are assumed to comprise only Unicode characters in the range `'\x00' through `'\xFF'`, with the more significant byte of the encoding of each character being 0.

**Note:** this option only has an effect as a compiler option, and applies to all programs being compiled. If present on an `options` instruction, it is checked and must match the compiler option (this allows processing with or without `utf8` to be enforced).

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

Prefixing any of the above with “no” turns the selected option off.

Example:

```plaintext
options binary nocrossref nostrictassign strictargs
```

The default settings of the various options are:

```plaintext
nobinary nocomments nocompact console crossref decimal nodiag noexplicit
noformat java logo noreplace nosavelog nosourcedir nostrictargs
nostrictassign nostrictcase nostrictimport nostrictprops nostrictsignal
nosymbols trace2 noutf8 verbose3
```

When an option word is repeated (in the same options instruction or not), or conflicting option words are specified, then the last use determines the state of the option.

All option words may also be set as command line options when invoking the processor, by prefixing them with “-”:

Example:

```plaintext
java COM.ibm.netrexx.process.NetRexxC -format foo.nrx
```

In this case, any options may come before, after, or between file specifications.

With the except of the utf8 option (see above), options set with the options instruction override command-line settings, following the “last use” rule.

For more information, see the installation and user documentation for your implementation.
**Package instruction**

```plaintext
package name;

where `name` is one or more non-numeric *symbols* separated by periods.
```

The `package` instruction is used to define the package to which the class or classes in the current program belong.

Classes that belong to the same package have privileged access to other classes in the same package, in that each class is visible to all other classes in the same package, even if not declared public. Packages also conveniently group classes for use by the `import` instruction (see page 87).

The `name` must specify a `package name`, which is one or more non-numeric symbols, separated by periods, with no blanks.

There must be at most one `package` instruction in a program. It must precede any `class` instruction (or any instruction that would start the default class).

If a program contains no `package` instruction then its package is implementation-defined. Typically it is grouped with other programs in some implementation-defined logical collection, such as a directory in a file system.

**Examples:**

```plaintext
    package testpackage
    package com.ibm.venta
```

When a class is identified as belonging to a package, it has a *qualified class name*, which is its short name, as given on the `class` instruction (see page 78), prefixed with the package name and a period. For example, if the short name of a class is “RxLanguage” and the package name is “com.ibm.venta” then the qualified name of the class would be “com.ibm.venta.RxLanguage”.

*In the reference implementation, packages are kept in a hierarchy derived from the Java classpath, where the segments of a package name correspond to a path in the hierarchy. The hierarchy is typically the directories in a file system, or some equivalent (such as a “Zip” archive file), and so package names should be considered case-sensitive (as some Java implementations use case-sensitive file systems).*
Parse instruction

```
parse term template;
```

where `template` is one or more non-numeric `symbols` separated by blanks and/or `patterns`, and a `pattern` is one of:

```
literalstring
  [indicator] number
  [indicator] (symbol)
```

and `indicator` is one of `+`, `-`, or `=`. The `parse` instruction is used to assign characters (from a string) to one or more variables according to the rules and templates described in the section Parsing templates (see page 133).

The value of the `term` is expected to be a string; if it is not a string, it will be converted to a string.

Any variables used in the `template` are named by non-numeric `symbols` (that is, they cannot be an array reference or other term); they refer to a variable or property in the current class. Any values that are used in patterns during the parse are converted to strings before use.

Any variables set by the `parse` instruction must have a known string type, or are given the NetRexx string type, `Rexx`, if they are new.

The term itself is not changed unless it is a variable which also appears in the template and whose value is changed by being in the template.

Example:

```
parse wordlist word1 wordlist
```

In this idiomatic example, the first word is removed from `wordlist` and is assigned to the variable `word1`, and the remainder is assigned back to `wordlist`.

Notes:

1. The special words `ask`, `source`, and `version`, as described in the section Special names and methods (see page 130), allow:

```
parse ask x     -- parses a line from input stream
parse source x -- parses 'Java method filename'
parse version x -- parses 'NetRexx version date'
```

These special words may also be used within expressions.

2. Similarly, it is recommended that the initial (main) method in a stand-alone application place the command string passed to it in a variable called `arg`.\(^\text{50}\)

If this is done, the instruction:

```
parse arg template
```

will work, in a stand-alone application, in the same way as in Rexx (even though `arg` is not a keyword in this case).\(^\text{51}\)

\(^{50}\) In the reference implementation, this is automatic if the `main` method is generated by the NetRexx language processor.

\(^{51}\) Note, though, that the command string may have been edited by the environment; certain characters may not be allowed, multiple blanks may have been reduced to single blanks, etc.
### Properties instruction

```netrexx
properties [visibility] [modifier] [deprecated] [unused];
```

where `visibility` is one of:

- inheritable
- private
- public
- shared

and `modifier` is one of:

- constant
- static
- transient
- volatile

and there must be at least one `visibility` or `modifier` keyword.

The `properties` instruction is used to define the attributes of following `property` variables, and therefore must precede the first `method` instruction in a class. A `properties` instruction replaces any previous `properties` instruction (that is, the attributes specified on `properties` instructions are not cumulative).

The `visibility`, `modifier`, `deprecated`, and `unused` keywords may be in any order.

An example of the use of `properties` instructions may be found in the `Program Structure` section (see page 124).

### Visibility

Properties may be `public`, `inheritable`, `private`, or `shared`.\(^{52}\)

- A `public property` is visible to (that is, may be used by) all other classes to which the current class is visible.
- An `inheritable property` is visible to (that is, may be used by) all classes in the same package and also those classes that extend (that is, are subclasses of) the current class, and which qualify the property using an object of the subclass, or either `this` or `super`.
- A `private property` is visible only within the current class.
- A `shared property` is visible within the current package but is not visible outside the package. Shared properties cannot be inherited by classes outside the package.

By default, if no `properties` instruction is used, or `visibility` is not specified, properties are inheritable (but not public).\(^{53}\)

---

\(^{52}\) An experimental option for `visibility`, `indirect`, is described in Appendix B (see page 175).

\(^{53}\) The default, here, was chosen to encourage the “encapsulation” of data within classes.
**Modifier**

Properties may also be **constant, static, transient, or volatile:**

- A *constant property* is associated with the class, rather than with an instance of the class (an object). It is initialized when the class is loaded and may not be changed thereafter.
- A *static property* is associated with the class, rather than with an instance of the class (an object). It is initialized when the class is loaded, and may be changed thereafter.
- A *transient property* is a property which should not be saved when an instance of the class is saved (made persistent).
- A *volatile property* may change asynchronously, outside the control of the class, even when no method in the class is being executed. If an implementation does not allow asynchronous modification of properties, it should ignore this keyword.

Constant and static properties exist from when the class is first loaded (used), even if no object is constructed by the class, and there will only be one copy of each property. Other properties are constructed and initialized only when an object is constructed by the class; each object then has its own copy of such properties.

By default, if no **properties** instruction is used, or **modifier** is not specified, properties are associated with an object constructed by the class.

**Deprecated**

The keyword **deprecated** indicates that any property introduced by this instruction is deprecated, which implies that a better alternative is available and documented. A compiler can use this information to warn of out-of-date or other use that is not recommended.

**Unused**

The keyword **unused** indicates that the private properties which follow are not referenced explicitly in the code for the class, and so a language processor should not warn that they exist but have not been used. If a **visibility** keyword is specified it must be **private**.

For example:

```netrexx
properties private constant unused
-- Serialization version
serialVersionUID=long 8245355804974198832
```

**Properties in interface classes**

In interface classes (see page 79), properties must be both **public** and **constant**. In such classes, these attributes for properties are the default and the **properties** instruction must not be used.
**Return instruction**

```
return [expression];
```

`return` is used to return control (and possibly a result) from a NetRexx program or method to the point of its invocation.

The expression (if any) is evaluated, active control constructs are terminated (as though by a `leave` instruction), and the value of the expression is passed back to the caller.

The result passed back to the caller is a string of type `Rexx`, unless a different type was specified using the `returns` keyword on the `method` instruction (see page 98) for the current method. In this case, the type of the value of the expression must match (or be convertible to, as by the rules for assignment) the type specified by the `returns` phrase.

Within a method, the use of expressions on `return` must be consistent. That is, either all `return` instructions must specify a expression, or none may. If a `returns` phrase is given on the `method` instruction for the current method then all `return` instructions must specify an expression.
Say instruction

say [expression];

say writes a string to the default output character stream. This typically causes it to be displayed (or spoken, or typed, etc.) to the user.

Example:

data=100
say data 'divided by 4 =>' data/4
/* would display: "100 divided by 4 => 25" */

The result of evaluating the expression is expected to be a string; if it is not a string, it will be converted to a string. This result string is written from the program via an implementation-defined output stream.

By default, the result string is treated as a “line” (an implementation-dependent mechanism for indicating line termination is effected after the string is written). If, however, the string ends in the NUL character ('\- ' or '\0 ') then that character is removed and line termination is not indicated.

The result string may be of any length. If no expression is specified, or the expression result is null, then an empty line is written (that is, as though the expression resulted in a null string).
**Select instruction**

```
select [label name] [protect term] [case expression];
  whenlist
    [otherwise:] instructionlist]
  [catch [vare =] exception;
    instructionlist]...
  [finally[:]
    instructionlist]
end [name];
```

where `name` is a non-numeric symbol

and `whenlist` is one or more `whenconstructs`

and `whenconstruct` is:

```
  when expression[, expression]... [: then[: instruction
```

and `instructionlist` is zero or more `instructions`.

**select** is used to conditionally execute one of several alternatives. The construct may optionally be given a label, and may protect an object while the instructions in the construct are executed; exceptional conditions can be handled with **catch** and **finally**, which follow the body of the construct.

Starting with the first **when** clause, each expression in the clause is evaluated in turn from left to right, and if the result of any evaluation is 1 (or equals the **case** expression, see below) then the test has succeeded and the instruction following the associated **then** (which may be a complex instruction such as **if**, **do**, **loop**, or **select**) is executed and control will then pass directly to the **end**.

If the result of all the expressions in a **when** clause is 0, control will pass to the next **when** clause.

Note that once an expression evaluation in a **when** clause has resulted in a successful test, no further expressions in the clause are evaluated.

If none of the **when** expressions result in 1, then control will pass to the instruction list (if any) following **otherwise**. In this situation, the absence of an **otherwise** is a run-time error. 54

**Notes:**

1. An **instruction** may be any assignment, method call, or keyword instruction, including any of the more complex constructions such as **do**, **loop**, **if**, and the **select** instruction itself. A null clause is not an instruction, however, so putting an extra semicolon after the **then** is not equivalent to putting a dummy instruction (as it would be in C or PL/I). The **nop** instruction is provided for this purpose.

2. The keyword **then** is treated specially, in that it need not start a clause. This allows the expression on the **when** clause to be terminated by the **then**, without a “;” being required – were this not so, people used to other computer languages would be inconvenienced. Hence the symbol **then** cannot be used as a variable name within the expression. 55

54 In the reference implementation, a **NonQueryException** is raised.
55 Strictly speaking, **then** should only be recognized if not the name of a variable. In this special case, however, NetRexx
**Label phrase**

If `label` is used to specify a `name` for the select group, then a `leave` instruction (see page 90) which specifies that name may be used to leave the group, and the `end` that ends the group may optionally specify the name of the group for additional checking.

**Example:**

```netrexx
select label roman
  when a=b then say 'same'
  when a<b then say 'lo'
  otherwise
    say 'hi'
    if a=0 then leave roman
    say 'a non-0'
end roman
```

In this example, if the variable `a` has the value 0 and `b` is negative then just “hi” is displayed.

**Protect phrase**

If `protect` is given it must be followed by a `term` that evaluates to a value that is not just a type and is not of a primitive type; while the `select` construct is being executed, the value (object) is protected – that is, all the instructions in the `select` construct have exclusive access to the object.

Both `label` and `protect` may be specified, in any order, if required.

**Case phrase**

If `case` is given it must follow any `label` or `protect` phrase, and must be followed by an `expression`.

When `case` is used, the expression following it is evaluated at the start of the `select` construct. The result of the expression is then compared, using the strict equality operator (==), to the result of evaluating the expression or expressions in each of the `when` clauses in turn until a match is found. As usual, if no match is found then control will pass to the instruction list (if any) following `otherwise`, and in this situation the absence of an `otherwise` is a run-time error.

For example, in:

```netrexx
select case i+1
  when 1 then say 'one'
  when 1+1 then say 'two'
  when 3, 4, 5 then say 'many'
end
```

then if `i` had the value 1 then the message displayed would be “two”.

The third `when` clause in the example demonstrates the use of the multiple expressions in a `when` clause in this context. Similar to a `select` without `case`, each expression is evaluated in turn from left to right and is then compared to the result of the `case` expression. As soon as one matches that result, execution of the `when` clause stops (any further expressions are not evaluated) and the instruction following the associated `then` clause is executed.

**Notes:**

1. When `case` is used, the result of evaluating the expression following each `when` no longer has language processors are permitted to treat `then` as reserved in the context of a `when` clause, to provide better performance and more useful error reporting.

**Version 2.00**

NetRexx Language Definition 117
to be 0 or 1. Instead, it must be possible to compare each result to the result of the **case** expression.

2. The **case** expression is evaluated only on entry to the **select** construct; it is not re-evaluated for each **when** clause.

3. An exception raised during evaluation of the **case** expression will be caught by a suitable **catch** clause in the construct, if one is present. Similarly, evaluation of the **case** expression is protected by the **protect** phrase, if one is present.

4. **In the reference implementation, a select case construct will be translated into a Java switch construct provided that it meets the following criteria:**

   - The type of the **case** expression is **byte**, **char**, **int**, or **short**.
   - The value of all the expressions on the **when** clauses are primitive constants (that is, they consist of only constants of primitive types and operators valid for them and so may be evaluated at compile time).
   - No two expressions on the **when** clauses evaluate to the same value.
   - It is not subject to tracing.

Under these conditions the semantics of the switch construct match those defined for select. The example shown above would be translated to a switch construct if `i` had type **int** and **options binary** were in effect.

**Exceptions in select constructs**

Exceptions that are raised by the instructions within the body of the group, or during evaluation of the **case** expression, may be caught using one or more **catch** clauses that name the exception that they will catch. When an exception is caught, the exception object that holds the details of the exception may optionally be assigned to a variable, **vare**.

Similarly, a **finally** clause may be used to introduce instructions that will always be executed at the end of the select group, even if an exception is raised (whether caught or not).

The Exceptions section (see page 152) has details and examples of **catch** and **finally**.
**Signal instruction**

```plaintext
signal term;
```

The `signal` instruction causes an “abnormal” change in the flow of control, by raising an `exception`. The exception `term` may be a term that constructs or evaluates to an exception object, or it may be expressed as the name of an exception type (in which case the default constructor, with no arguments, for that type is used to construct an exception object). The exception object then represents the exception and is available, if required, when the exception is handled.

The handling of exceptions is detailed in the *Exceptions* section (see page 152). In summary, when an exception is signalled, all active pending `do` groups, `loop` loops, `if` constructs, and `select` constructs may be ended. For each one in turn, from the innermost:

1. No further clauses within the body of the construct will be executed (in this respect, `signal` acts like a `leave` for the construct).
2. The `instructionlist` following the first `catch` clause that matches the exception, if any, is executed.
3. The `instructionlist` following the `finally` clause for the construct, if any, is executed.

If a `catch` matched the exception the exception is deemed handled, and execution resumes as though the construct ended normally (unless a new exception was signalled in the `catch` or `finally` instruction lists, in which case it is processed). Otherwise, any enclosing construct is ended in the same manner. If there is no enclosing construct, then the current method is ended and the exception is signalled in the caller.

**Examples:**

```plaintext
signal RxErrorTrace
signal DivideException('Divide by zero')
```

*In the reference implementation, the term must either*

- *evaluate to an object that is assignable to the type Throwable (for example, a subclass of Exception or RuntimeException).*

- *be a type that is a subclass of Throwable, in which case the default constructor (with no arguments) for the given type is used to construct the exception object.*
### Trace instruction

```
trace traceoption;
```

where `traceoption` is one of:

```
   tracesetting  var [varlist]
```

where `tracesetting` is one of:

```
   all
   methods
   off
   results
```

and `varlist` is one or more variable `names`, optionally prefixed with a `+` or `-`

---

The `trace` instruction is used to control the tracing of the execution of NetRexx methods, and is primarily used for debugging. It may change either the general trace `setting` or may select or deselect the tracing of individual variables.

Within methods, the `trace` instruction changes the trace setting or variables tracing when it is executed, and affects the tracing of all clauses in the method which are then executed (until changed by a later `trace` instruction).

One or more `trace` instructions may appear before the first method in a class, one of which may set the initial trace setting for all methods in the class (the default is `off`) and others may set up variables tracing that applies to all the methods in the class. These act as though the `trace` instructions were placed immediately following the `method` instruction in each method (except that they will not be traced).

Similarly, one or more `trace` instructions may be placed before the first `class` instruction in a program; they do not imply the start of a class. One of these may set the initial trace setting and others may set up variables tracing for all classes in the program (except interface classes) and act as though the `trace` instructions were placed immediately following the `class` instruction in each class.

### Tracing clauses

The trace `setting` controls the tracing of clauses in a program, and may be one of the following:

- **all**: All clauses (except null clauses without commentary) which are in methods and which are executed after the `trace` instruction will be traced. If `trace all` is placed before the first method in the current class, the `method` instructions in the class, together with the values of the arguments passed to each method, will be traced when the method is invoked (that is, `trace all` implies `trace methods`).

- **methods**: All `method` clauses in the class will be traced when the method they introduce is invoked, together with the values of the arguments passed to each method; no other clauses, or results, will be traced. The `trace methods` instruction must be placed before the first method in the current class (as otherwise it would have no effect).

- **off**: Turns tracing off; no following clauses, variables, or results will be traced.
All clauses (except null clauses without commentary) which are in methods and which are executed after the `trace` instruction will be traced, as though `trace all` had been requested. In addition, the results of all `expression` evaluations and any results assigned to a variable by an assignment, `loop`, or `parse` instruction are also traced.

If `trace results` is placed before the first method in the current class, the `method` instructions in the class will be traced when the method is invoked, together with the values of the arguments passed to each method.

Notes:

1. Tracing of clauses shows the data from the source of the program, starting at the first character of the first token of the clause and including any commentary from that point until the end of the clause.

2. When a loop is being traced, the `loop` clause itself will be traced on every iteration of the loop, as indicated by the programmer’s model (see page 96); the `end` clause is only traced once, when the loop completes normally.

3. With `trace results`, an expression is not traced if it is immediately used for an assignment (in an assignment instruction, or when the control variable is initialized in a `loop` instruction). The assignment will trace the result of the expression.

**Tracing variables**

The `var` option adds names to a list of monitored variables; it can also remove names from the list. 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` clause that assigns a new value to the named variable.

Variable names are specified by listing them after the `var` keyword. Each name may be optionally prefixed by a `+` or a `-` sign. A `+` sign indicates that the variable is to be added to the list of monitored variables (the default), and a `-` sign indicates that the variable is to be removed from the list. Blanks may be added before and after variable names and signs to separate the tokens and to improve readability.

For example:

```plaintext
trace var a b c
  -- now variables a, b, and c will be traced
trace var -b -c d
  -- now variables a and d will be traced
```

Notes:

1. Names in the list following the `var` keyword are simple symbols that name variables in the current class or current method. The variables may be properties, method arguments, or local variables, and may be of any type, including arrays. The names are not case-sensitive; any variables whose names match, independent of case, will be monitored.

2. No variable name can appear more than once in the list on one `trace var` instruction. However, it is not an error to add the name of a variable which does not exist or is not then assigned a value. Similarly, it is not an error to remove a name which is not currently being monitored.

3. One or more `trace var` instructions (along with one other `trace` instruction) are allowed before the first method in a class. They all modify an initial list of monitored variables which is then...
used for all methods in the class. Similarly, **trace var** instructions are allowed before the first class in a program, in which case they apply to all classes (except interface classes).

4. Other **trace** instructions do not affect the list of monitored variables. The **trace off** instruction may be used to turn off tracing completely; in this case **trace var** (with or without any variable names) will then turn the tracing of variables back on, using the current (or modified) variable list.

5. For a **parse** instruction, only monitored variables have their assignments traced (unless **trace results** is already in effect).

**The format of trace output**

Trace output is either clauses from the program being traced, or results (such as the results from expressions).

The first clause or result traced on any line will be preceded by its line number in the program; this is right-justified in a space which allows for the largest line number in the program, plus one blank. Following clauses or results from the same line are preceded by white space of the same width; however, any change of line number causes the line number to be included.

Clauses that are traced will be displayed with the formatting (indentation) and layout used in the original source stream for the program, starting with the first character of the first token of the clause.

Results (if requested) are converted to a string for tracing if necessary, are not indented, and have a double quote prefixed and suffixed so that leading and trailing blanks are apparent; if, however, the result being traced is null (see page 131) then the string “null” is shown (without quotes). For results with an associated name (the values assigned to local variables, method arguments, or properties in the current class), the name of the result precedes the data, separated by a single blank.

For clarity, implementations may replace “control codes” in the encoding of results (for example, EBCDIC values less than ‘\x40’, or Unicode values less than ‘\x20’) by a question mark (“?”).

All lines displayed during tracing have a three character tag to identify the type of data being traced. This tag follows the line number (or the space for a line number), and is separated from the line number by a single blank. The traced clause or result follows the tag, after another blank. The identifier tags may be:

- **=* identifies the first line of the source of a single clause, *i.e.*, the data actually in the program.
- **=* identifies a continuation line from the source of a single clause. Continuations may be due to the use of a continuation character (see page 44) or to the use of a block comment (see page 40) which spans more than one line.
- **a>** Identifies a value assigned to a method argument of the current method. The name of the argument is included in the trace.
- **p>** Identifies a value assigned to a property. The name of the property is included in the trace if the property is in the current class.
- **v>** Identifies a value assigned to a local variable in the current method. The name of the variable is included in the trace.
- **>>** Identifies the result of an expression evaluation that is not used for an assignment (for example, an argument expression in a method call).
- **+++** Reserved for error messages that are not supplied by the environment underlying the
implementation.

If a trace line is produced in a different context (program or thread) from the preceding trace line (if any) then a trace context line is shown. This shows the name of the program that produced the trace line, and also the name of the thread (and thread group) of the context.

The thread group name is not shown if it is main, and in this case the thread name is then also suppressed if its name is main.

Examples:

If the following instructions, starting on line 53 of a 120-line program, were executed:

```plaintext
trace all
if i=1 then say 'Hello'
   else say 'i<>1'
say -
   'A continued line'
```

the trace output (if i were 1) would be:

```
54 *=* if i=1
55 *=*    then
56 *=*      say 'Hello'
57 *=*    say -
58 *=*      'A continued line'
```

Similarly, for the 3-line program:

```plaintext
trace results
number=1/7
parse number before '.' after
```

the trace output would be:

```
2 *=* number=1/7
2 >v> number "0.142857143"
3 *=* parse number before '.' after
3 >v> before "0"
3 >v> after "142857143"
```

Notes:

1. Trace output is written to an implementation-defined output stream (typically the “standard error” output stream, which lets it be redirected to a destination separate from the default destination for output which is used by the say instruction).

2. In some implementations, the use of trace instructions may substantially increase the size of classes and the execution time of methods affected by tracing.\(^{56}\)

3. With some implementations it may be possible to switch tracing on externally, without requiring modification to the program.

---

\(^{56}\) In the reference implementation, options notrace may be used to disable all trace instructions and hence ensure that tracing overhead is not accidentally incurred.
Program structure

A NetRexx program is a collection of clauses (see page 40) derived from a single implementation-defined source stream (such as a file). When a program is processed by a language processor\textsuperscript{57} it defines one or more classes. Classes are usually introduced by the class instruction (see page 78), but if the first is a standard class, intended to be run as a stand-alone application, then the class instruction can be omitted. In this case, NetRexx defines an implied class and initialization method that will be used.

The implied class and method permits the writing of “low boilerplate” programs, with a minimum of syntax. The simplest, documented, NetRexx program that has an effect might therefore be:

Example:

```netrexx
/* This is a very simple NetRexx program */
say 'Hello World!'
```

In more detail, a NetRexx program consists of:

1. An optional prolog (package, import, and options instructions). Only one package instruction is permitted per program.

2. One or more class definitions, each introduced by a class instruction.

A class definition comprises:

1. The class instruction which introduces the class (which may be inferred, see below).

2. Zero or more property variable assignments, along with optional properties instructions that can alter their attributes, and optional numeric and trace instructions. Property variable assignments take the form of an assignment (see page 69), with an optional “=” and expression, which may:
   \begin{itemize}
   \item just name a property (by omitting the “=” and expression of the assignment), in which case it refers to a string of type Rexx
   \item assign a type to the property (when the expression evaluates to just a type)
   \item assign a type and initial value to the property (when the expression returns a value).
   \end{itemize}

3. Zero or more method definitions, each introduced by a method instruction (which may be inferred if the class instruction is inferred, see below).

A method definition comprises:

- Any NetRexx instructions, except the class, method, and properties instructions and those allowed in the prolog (the package, import, and options instructions).

\textsuperscript{57} Such as a compiler or interpreter.
/* A program with two classes */
import java.applet.   -- for example

class testclass extends Applet
properties public
    state             -- property of type 'Rexx'
    i=int             -- property of type 'int'
properties constant
    j=int 3           -- property initialized to '3'

method start
    say 'I started'
    state='start'

method stop
    say 'I stopped'
    state='stop'

class anotherclass
method testing
    loop i=1 to 10
        say '1, 2, 3, 4...'
        if i=7 then return
    end
    return

method anothertest
    say '1, 2, 3, 4'

This example shows a prolog (with just an import instruction) followed by two classes. The first class includes two public properties, one constant property, and two methods. The second class includes no properties, but also has two methods.

Note that a return instruction implies no static scoping; the content of a method is ended by a method (or class) instruction, or by the end of the source stream. The return instruction at the end of the testing method is, therefore, unnecessary.

Program defaults
The following defaults are provided for NetRexx programs:

1. If, while parsing prolog instructions, some instruction that is not valid for the prolog and is not a class instruction is encountered, then a default class instruction (with an implementation-provided short name, typically derived from the name of the source stream) is inserted. If the instruction was not a method instruction, then a default method instruction (with a name and attributes appropriate for the environment, such as main) is also inserted.

In this latter case, it is assumed that execution of the program will begin by invocation of the default method. In other words, a “stand-alone” application can be written without explicitly providing the class and method instructions for the first method to be executed. An example of such a program is given in Appendix A (see page 171).

In the reference implementation, the main method in a stand-alone application is passed the words forming the command string as an array of strings of type java.lang.String (one word to each element of the array). When the NetRexx reference implementation provides the main method instruction by default, it also constructs a NetRexx string of type Rexx from this
array of words, with a blank added between words, and assigns the string to the variable arg.

The command string may also have been edited by the underlying operating system environment; certain characters may not be allowed, multiple blanks or whitespace may have been reduced to single blanks, etc.

2. If a method ends and the last instruction at the outer level of the method scope is not return then a return instruction is added if it could be reached. In this case, if a value is expected to be returned by the method (due to other return instructions returning values, or there being a returns keyword on the method instruction), an error is reported.

Language processors may provide options to prevent, or warn of, these defaults being applied, as desired.
**Minor and Dependent classes**

A *minor class* in NetRexx is a class whose name is qualified by the name of another class, called its *parent*, and a *dependent class* is a minor class that has a link to its parent class that allows a child object simplified access to its parent object and its properties.

**Minor classes**

A *minor class* in NetRexx is a class whose name is qualified by the name of another class, called its *parent*. This qualification is indicated by the form of the name of the class: the short name of the minor class is prefixed by the name of its parent class (separated by a period). For example, if the parent is called *Foo* then the full name of a minor class *Bar* would be written *Foo.Bar*. The short name, *Bar*, is used for the name of any constructor method for the class; outside the class it can only be used to identify the class in the context of the parent class (or from children of the minor class, see below).

The names of minor classes may be used in exactly the same way as other class names (types) in programs. For example, a property might be declared and initialized thus:

```netrexx
abar=Foo.Bar null  -- this has type Foo.Bar
```

or, if the class has a constructor, perhaps:

```netrexx
abar=Foo.Bar()    -- constructs a Foo.Bar object
```

Minor classes must be in the same program (and hence in the same package) as their parent. They are introduced by a *class* instruction that specifies their full name, for example:

```netrexx
class Foo.Bar extends SomeClass
```

Minor classes must immediately follow their parent class. Minor classes may have a parent which is itself a minor class, to any depth; the name and the positioning rules are extended as necessary. For example, the following classes might exist in a program:

```netrexx
class Foo
    class Foo.Bar
        class Foo.Bar.Nod
        class Foo.Bar.Pod
    class Foo.Car
```

As before, the children of *Foo.Bar* immediately follow their parent. The list of children of *Foo* can be continued after the children of *Foo.Bar* have all been specified.

Note that the short name (last part of the name) of a minor class may not be the same as the short name of any of its parents (a class *Foo.Bar.Foo* or a class *Foo.Bar.Bar* would be in error, for example). This allows minor classes to refer to their parent classes by their short name without ambiguity.

**Constructing objects in minor classes**

A parent class can construct an object of a child class in the usual manner, by simply specifying its constructor (identified by its short name, full name, or qualified name). For example, a method in the *Foo.Bar* class above could construct an object of type *Foo.Bar.Nod* using:

```netrexx
anod=Nod()
```

---

58 This allows compilers that generate Java source code to preserve line numbering.
(assuming the Foo.Bar.Nod class has a constructor that takes no arguments).

Similarly, minor classes can refer to the types and constructors of any of its parents by simply using their short names. Hence, the Foo.Bar.Nod class could construct objects of its parents' types thus:

    aBar=Bar()
    aFoo=Foo()

(again assuming the parent classes have constructors that take no arguments).

Classes other than the parent or an immediate child must use the full name (if necessary, qualified by the package name) to refer to a minor class or its constructor.

**Dependent classes**

As described in the last section, minor classes provide an enhanced packaging (naming) mechanism for classes, allowing classes to be structured within packages. A stronger link between a child class and its parent is indicated by the modifier keyword `dependent` on the child class, which indicates that the child is a dependent class. For example:

```plaintext
class Foo.Dep dependent extends SomeClass
  method Dep -- this is the constructor
```

An object constructed from a dependent class (a dependent object) is linked to the context of an object of its parent type (its parent object). The linkage thus provided allows the child object simplified access to the parent object and its properties.

In the example, an object of type `Foo.Dep` can only be constructed in the context of a parent object, which must be of type `Foo`.

**Constructing dependent objects**

A parent class can construct a dependent object in the same way as when constructing objects of other child types; that is, by simply specifying its constructor. In this case, however, the current object (this) becomes the parent object of the newly constructed object. For example, a method in the `Foo` class above could construct a dependent object of type `Foo.Dep` using:

    aDep=Dep()

(assuming the `Dep` class has a constructor that takes no arguments).

In general, for a class to construct an object from a dependent class, it must have a reference to an object of the parent class (which will become the parent of the new object), and the constructor must be called (by its short name) in the context of that parent object. For example:

```plaintext
parentObject=Foo()
adep=parentObject.Dep()
```

(In the same way, the first example could have been written:

    adep=this.Dep()

within the parent class the `this` is implied.)

In order to subclass a dependent class, the constructor of the dependent class must be invoked by the subclass constructor in a similar manner. In this case, a qualified call to the usual special constructor `super` is used, for example:
class ASub extends Foo.Dep
method Asub(afoo=Foo)
   afoo.super()

The qualifier (afoo in the example) must be either the name of an argument to the constructor, or the special word parent (if the classes share a common parent class), or the short name of a parent class followed by .this (see below). The call to super must be the first instruction in the method, as usual, and it must be present (it will not be generated automatically by the compiler).

Access to parent objects and their properties
Dependent classes have simplified access to their parent objects and their properties. In particular:

• The special word parent may be used to refer to the parent object of the current object. It may appear alone in a term, or at the start of a compound term. It can only be used in non-static contexts in a dependent class.

• In general, any of the objects in the chain of parents of a dependent object may be referred to by qualifying the special word this with the short name of the parent class. For example, extending the previous example, if the class Foo.Dep.Ent was a dependent class it could contain references to Foo.this (the parent of its parent) or Dep.this (the latter being the same as specifying parent). If preferred, the full name or the fully qualified name of the parent class may be used instead of the short name.

Like parent, this construct can only be used at the start of a term in non-static contexts in a dependent class.

• As usual, properties external to the current class must always be qualified in some way (for example, the prefix parent. can be used in a term such as parent.aprop).

Restrictions
Minor classes may have any of the attributes (public, interface, etc.) of other classes, and behave in every way like other classes, with the following restrictions:

• If a class is a static class (that is, it contains only static or constant properties and methods) then any children cannot be dependent classes (because no object of the parent class can be constructed). Similarly, interface classes and abstract classes cannot have dependent classes.

• Dependent classes may not be interfaces.

• Dependent classes may not contain static or constant properties (or methods). These must be placed in a parent which is not a dependent class.

• Minor classes may be public only if their parent is also public. (Note that this is the only case where more than one public class is permitted in a program.) In general: a minor class cannot be more visible than its parent.

59 This restriction allows compilation for the Java platform.
**Special names and methods**

For convenience, NetRexx provides some *special names* for naming commonly-used concepts within terms. These are only recognized if there is no variable of the same name previously seen in the current scope, as described in the section on Terms (see page 48). This allows the set of special words to be expanded in the future, if necessary, without invalidating existing variables. Therefore, these names are not reserved; they may be used as variable names instead, if desired.

There are also two “special methods” that are used when constructing objects.

**Special names**

The following special names are allowed in NetRexx programs, and are recognized independently of case. With the exception of `length` and `class`, these may only be used alone as a term or at the start of a compound term.

- `ask` Returns a string of type Rexx, read as a line from the implementation-defined default input stream (often the user’s “console”).

  **Example:**
  ```rexx
  if ask='yes' then say 'OK'
  ```

  `ask` can only appear alone, or at the start of a compound term.

- `class` The object of type Class that describes a specific type. This word is only recognized as the second part of a compound term, where the evaluation of the first part of the term resulted in a type or qualified type.

  **Example:**
  ```rexx
  obj=String.class
  say obj.isInterface /* would say '0' */
  ```

- `digits` The current setting of numeric digits (see page 104), returned as a string of type Rexx. This will be one or more Arabic numerals, with no leading blanks, zeros, or sign, and no trailing blanks or exponent.

  `digits` can only appear alone, or at the start of a compound term.

- `form` The current setting of numeric form (see page 104), returned as a string of type Rexx. This will have either the value “scientific” or the value “engineering”.

  `form` can only appear alone, or at the start of a compound term.

- `length` The length of an array (see page 74), returned as an implementation-dependent binary type or string. This word is only recognized as the last part of a compound term, where the evaluation of the rest of the term resulted in an array of dimension 1.

  **Example:**
  ```rexx
  foo=char[7]
  say foo.length /* would say '7' */
  ```

  Note that you can get the length of a NetRexx string with the same syntax. In that

---

60 Unless options strictcase is in effect.

61 In the reference implementation, `ask` is simply a shorthand for RexxIO.Ask().

62 Unless options strictargs is in effect.
null

The empty reference. This is a special value that represents “no value” and may be assigned to variables (or returned from methods) except those whose type is both primitive and undimensioned. It may also be used in a comparison for equality (or inequality) with values of suitable type, and may be given a type.

Examples:

```
blob=int[3]   -- 'blob' refers to array of 3 ints
blob=null     -- 'blob' is still of type int[],
               -- but refers to no real object
mob=Mark null -- 'mob' is type 'Mark'
```

The null value may be considered to represent the state of being uninitialized. It can only appear as simple symbol, not as a part of a compound term.

source

Returns a string of type Rexx identifying the source of the current class. The string consists of the following words, with a single blank between the words and no trailing or leading blanks:

1. the name of the underlying environment (e.g., Java)
2. either method (if the term is being used within a method) or class (if the term is being used within a property assignment, before the first method in a class)
3. an implementation-dependent representation of the name of the source stream for the class (e.g., Fred.nrx).

source can only appear alone, or at the start of a compound term.

sourceline

The line number of the first token of the current clause in the NetRexx program, returned as a string of type Rexx. This will be one or more Arabic numerals, with no leading blanks, zeros, or sign, and no trailing blanks or exponent.

sourceline can only appear alone, or at the start of a compound term.

super

Returns a reference to the current object, with a type that is the type of the class that the current object’s class extends. This means that a search for methods or properties which super qualifies will start from the superclass rather than in the current class. This is used for invoking a method or property (in the superclass or one of its superclasses) that has been overridden in the current class.

Example:

```
method printit(x)
  say 'it'          -- modification
  super.printit(x)  -- now the usual processing
```

If a property being referenced is in fact defined by a superclass of the current class, then the prefix “super.” is perhaps the clearest way to indicate that name refers to a property of a superclass rather than to a local variable. (You could also qualify it by the name of the superclass.)

super can only appear alone, or at the start of a compound term.

this

Returns a reference to the current object. When a method is invoked, for example in:
word=Rexx "hello" -- 'word' refers to "hello"
say word.substr(3) -- invokes substr on "hello"

then the method substr in the class Rexx is invoked, with argument '3', and with the properties of the value (object) "hello" available to it. These properties may be accessed simply by name, or (more explicitly) by prefixing the name with “this.”. Using “this.” can make a method more readable, especially when several objects of the same type are being manipulated in the method.

this can only appear alone, or at the start of a compound term.

trace

The current trace (see page 120) setting, returned as a NetRexx string. This will be one of the words:

    off var methods all results

(var is returned when clause tracing is off but variable tracing has then been turned on using a trace var instruction.)

trace can only appear alone, or at the start of a compound term.

version

Returns a string of type Rexx identifying the version of the NetRexx language in effect when the current class was processed. The string consists of the following words, with a single blank between the words and no trailing or leading blanks:

1. A word describing the language. The first seven letters will be the characters NetRexx, and the remainder may be used to identify a particular implementation or language processor. This word may not include any periods.

2. The language level description, which must be a number with no sign or exponential part. For example, “2.00” is the language level of this definition.

3. Three words describing the language processor release date in the same format as the default for the Rexx “date()” function.63 For example, “22 May 2009”.

version can only appear alone, or at the start of a compound term.

Special methods

Constructors (methods used for constructing objects) in NetRexx must invoke a constructor of their superclass before making any modifications to the current object (or invoke another constructor in the current class).

This is simplified and made explicit by the provision of the special method names super and this, which refer to constructors of the superclass and current class respectively. These special methods are only recognized when used as the first, method call, instruction in a constructor, as described in Methods and constructors (see page 53). Their names will be recognized independently of case.64

In addition, NetRexx provides special constructor methods for the primitive types that allow binary construction of primitives. These are described in Binary values and arithmetic (see page 150).

64 Unless options strictcase is in effect.
**Parsing templates**

The `parse` instruction allows a selected string to be parsed (split up) and assigned to variables, under the control of a *template*.

The various mechanisms in the template allow a string to be split up by explicit matching of strings (called *patterns*), or by specifying numeric positions (*positional patterns*) – for example, to extract data from particular columns of a line read from a character stream. Once split into parts, each segment of the string can then be assigned to variables as a whole or by words (delimited by blanks).

This section first gives some informal examples of how the parsing template can be used, and then defines the algorithms in detail.

**Introduction to parsing**

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

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

the term (in this case a literal string) following the instruction keyword is parsed, and then: 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.”.

Leading blanks are removed from each word in the string before it is assigned to a variable, as is the blank that delimits the end of the word. Thus, variables set in this manner (`v1` and `v2` in the example) will never have leading or trailing blanks, though `v3` could have both leading and trailing blanks.

Note that the variables assigned values in a template are always given a new value and so if there are fewer words in the string than variables in the template then the unused variables will be set to the null string.

The second parsing mechanism uses a literal string in a template as a pattern, to split up the string. For example:

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

would cause the string to be scanned for the comma, and then split at that point: the variable `w1` would be set to “To be”, and `w2` is set to “ or not to be?”. Note that the pattern itself (and only the pattern) is removed from the string. Each section of the string is treated in just the same way as the whole string was in the previous example, and so either section could be split up into words.

Thus, in:

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

`w2` and `w3` would be assigned the values “or” and “not”, and `w4` would be assigned the remainder: “to be?”.

If the string in the last example did not contain a comma, then the pattern would effectively “match” the end of the string, so the variable to the left of the pattern would get the entire input string, and the variables to the right would be set to a null string.

The pattern may be specified as a variable, by putting the variable name in parentheses. The following
instructions therefore have the same effect as the last example:

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

The third parsing mechanism is the numeric positional pattern. This works in the same way as the string pattern except that it specifies a column number. So:

```r xl
  parse 'Flying pigs have wings' x1 5 x2
```

would split the string at the fifth column, so \texttt{x1} would be \texttt{"Flyi"} and \texttt{x2} would start at column 5 and so be \texttt{"ng pigs have wings"}.

More than one pattern is allowed, so for example:

```r xl
  parse 'Flying pigs have wings' x1 5 x2 10 x3
```

would split the string at columns 5 and 10, so \texttt{x2} would be \texttt{"ng pi"} and \texttt{x3} would be \texttt{"gs have wings"}.

The numbers can be relative to the last number used, so:

```r xl
  parse 'Flying pigs have wings' x1 5 x2 +5 x3
```

would have exactly the same effect as the last example; here the +5 may be thought of as specifying the length of the string to be assigned to \texttt{x2}.

As with literal string patterns, the positional patterns can be specified as a variable by putting the name of a variable, in parentheses, in place of the number. An absolute column number should then be indicated by using an equals sign ("=") instead of a plus or minus sign. The last example could therefore be written:

```r xl
  start=5
  length=5
  data='Flying pigs have wings'
  parse data x1 =(start) x2 +(length) x3
```

String patterns and positional patterns can be mixed (in effect the beginning of a string pattern just specifies a variable column number) and some very powerful things can be done with templates. The next section describes in more detail how the various mechanisms interact.

**Parsing definition**

This section describes the rules that govern parsing.

In its most general form, a template consists of alternating pattern specifications and variable names. Blanks may be added between patterns and variable names to separate the tokens and to improve readability. The patterns and variable names are used strictly in sequence from left to right, and are used once only. In practice, various simpler forms are used in which either variable names or patterns may be omitted; we can therefore have variable names without patterns in between, and patterns without intervening variable names.

In general, the value assigned to a variable is that sequence of characters in the input string between the point that is matched by the pattern on its left and the point that is matched by the pattern on its right.

If the first item in a template is a variable, then there is an implicit pattern on the left that matches the start of the string, and similarly if the last item in a template is a variable then there is an implicit pattern on the right that matches the end of the string. Hence the simplest template consists of a single
variable name which in this case is assigned the entire input string.

Setting a variable during parsing is identical in effect to setting a variable in an assignment.

The constructs that may appear as patterns fall into two categories; patterns that act by searching for a matching string (literal patterns), and numeric patterns that specify an absolute or relative position in the string (positional patterns). Either of these can be specified explicitly in the template, or alternatively by a reference to a variable whose value is to be used as the pattern.

For the following examples, assume that the following sample string is being parsed; note that all blanks are significant – there are two blanks after the first word “is” and also after the second comma:

'This is the text which, I think, is scanned.'

**Parsing with literal patterns**

Literal patterns cause scanning of the data string to find a sequence that matches the value of the literal. Literals are expressed as a quoted string. The null string matches the end of the data.

The template:

```
  w1 ', ' w2 ', ' w3
```

when parsing the sample string, results in:

```
  w1 has the value "This is the text which"
  w2 has the value " I think"
  w3 has the value " is scanned."
```

Here the string is parsed using a template that asks that each of the variables receive a value corresponding to a portion of the original string between commas; the commas are given as quoted strings. Note that the patterns themselves are removed from the data being parsed.

A different parse would result with the template:

```
  w1 ', ' w2 ', ' w3 ', ' w4
```

which would result in:

```
  w1 has the value "This is the text which"
  w2 has the value " I think"
  w3 has the value " is scanned."
  w4 has the value "" (null string)
```

This illustrates an important rule. When a match for a pattern cannot be found in the input string, it instead “matches” the end of the string. Thus, no match was found for the third ', ' in the template, and so w3 was assigned the rest of the string. w4 was assigned a null string because the pattern on its left had already reached the end of the string.

Note that all variables that appear in a template in this way are assigned a new value.

**Parsing strings into words**

If a variable is directly followed by one or more other variables, then the string selected by the patterns is assigned to the variables in the following manner. Each blank-delimited word in the string is assigned to each variable in turn, except for the last variable in the group (which is assigned the remainder of the string). The values of the variables which are assigned words will have neither leading nor trailing blanks.
Thus the template:
\[w_1 \ w_2 \ w_3 \ w_4 \ ',',\]
would result in:
\[w_1 \text{ has the value } "This"\]
\[w_2 \text{ has the value } "is"\]
\[w_3 \text{ has the value } "the"\]
\[w_4 \text{ has the value } "text \ which"\]
Note that the final variable (\(w_4\) in this example) could have had both leading blanks and trailing blanks, since only the blank that delimits the previous word is removed from the data.

Also observe that this example is not the same as specifying explicit blanks as patterns, as the template:
\[w_1 \ ' ' \ w_2 \ ' ' \ w_3 \ ' ' \ w_4 \ ',',\]
would in fact result in:
\[w_1 \text{ has the value } "This"\]
\[w_2 \text{ has the value } "is"\]
\[w_3 \text{ has the value } "" (null string)\]
\[w_4 \text{ has the value } "the \ text \ which"\]
since the third pattern would match the third blank in the data.

In general, when a variable is followed by another variable then parsing of the input into individual words is implied. The parsing process may be thought of as first splitting the original string up into other strings using the various kinds of patterns, and then assigning each of these new strings to (zero or more) variables.

**Use of the period as a placeholder**

A period (separated from any symbols by at least one blank) acts as a placeholder in a template. It has exactly the same effect as a variable name, except that no variable is set. It is especially useful as a “dummy variable” in a list of variables, or to collect (ignore) unwanted information at the end of a string. Thus the template:
\[... \ \text{word}_4 \ .\]
would extract the fourth word (“text”) from the sample string and place it in the variable \text{word}_4. Blanks between successive periods in templates may be omitted, so the template:
\[... \ \text{word}_4 \ .\]
would have the same result as the last template.

**Parsing with positional patterns**

Positional patterns may be used to cause the parsing to occur on the basis of position within the string, rather than on its contents. They take the form of whole numbers, optionally preceded by a plus, minus, or equals sign which indicate relative or absolute positioning. These may cause the matching operation to “back up” to an earlier position in the data string, which can only occur when positional patterns are used.
**Absolute positional patterns:** A number in a template that is not preceded by a sign refers to a particular (absolute) character column in the input, with 1 referring to the first column. For example, the template:

```
   s1 10 s2 20 s3
```

results in:

```
  s1 has the value "This is 
  s2 has the value "the text w"
  s3 has the value "hich, I think, is scanned."
```

Here \( s1 \) is assigned characters from the first through the ninth character, and \( s2 \) receives input characters 10 through 19. As usual the final variable, \( s3 \), is assigned the remainder of the input.

An equals sign ("=") may be placed before the number to indicate explicitly that it is to be used as an absolute column position; the last template could have been written:

```
   s1 =10 s2 =20 s3
```

A positional pattern that has no sign or is preceded by the equals sign is known as an absolute positional pattern.

**Relative positional patterns:** A number in a template that is preceded by a plus or minus sign indicates movement relative to the character position at which the previous pattern match occurred. This is a relative positional pattern.

If a plus or minus is specified, then the position used for the next match is calculated by adding (or subtracting) the number given to the last matched position. The last matched position is the position of the first character of the last match, whether specified numerically or by a string.

For example, the instructions:

```
parse '123456789' 3 w1 +3 w2 3 w3
```

result in

```
  w1 has the value "345"
  w2 has the value "6789"
  w3 has the value "3456789"
```

The +3 in this case is equivalent to the absolute number 6 in the same position, and may also be considered to be specifying the length of the data string to be assigned to the variable \( w1 \).

This example also illustrates the effects of a positional pattern that implies movement to a character position to the left of (or to) the point at which the last match occurred. The variable on the left is assigned characters through the end of the input, and the variable on the right is, as usual, assigned characters starting at the position dictated by the pattern.

A useful effect of this is that multiple assignments can be made:

```
parses x 1 w1 1 w2 1 w3
```

This results in assigning the (entire) value of \( x \) to \( w1, w2, \) and \( w3 \). (The first "1" here could be omitted as it is effectively the same as the implicit starting pattern described at the beginning of this section.)

If a positional pattern specifies a column that is greater than the length of the data, it is equivalent to specifying the end of the data (i.e., no padding takes place). Similarly, if a pattern specifies a column to the left of the first column of the data, this is not an error but instead is taken to specify the first column of the data.
Any pattern match sets the “last position” in a string to which a relative positional pattern can refer. The “last position” set by a literal pattern is the position at which the match occurred, that is, the position in the data of the first character in the pattern. The literal pattern in this case is not removed from the parsed data. Thus the template:

', -1 x +1

will:

1. Find the first comma in the input (or the end of the string if there is no comma).
2. Back up one position.
3. Assign one character (the character immediately preceding the comma or end of string) to the variable $x$.

One possible application of this is looking for abbreviations in a string. Thus the instruction:

/* Ensure options have a leading blank and are 
in uppercase before parsing. */
parse (' 'opts).upper ' PR' +1 prword ' '

will set the variable $prword$ to the first word in $opts$ that starts with “PR” (in any case), or will set it to the null string if no such word exists.

Notes:

1. The positional patterns +0 and –0 are valid, have the same effect, and may be used to include the whole of a previous literal (or variable) pattern within the data string to be parsed into any following variables.
2. As illustrated in the last example, patterns may follow each other in the template without intervening variable names. In this case each pattern is obeyed in turn from left to right, as usual.
3. There may be blanks between the sign in a positional pattern and the number, because NetRexx defines that blanks adjacent to special characters are removed.

Parsing with variable patterns

It is sometimes desirable to be able to specify a pattern by using the value of a variable instead of a fixed string or number. This may be achieved by placing the name of the variable to be used as the pattern in parentheses (blanks are not necessary either inside or outside the parentheses, but may be added if desired). This is called a variable reference; the value of the variable is converted to string before use, if necessary.

If the parenthesis to the left of the variable name is not preceded by an equals, plus, or minus sign (“=”, “+”, or “–”) the value of the variable is then used as though it were a literal (string) pattern. The variable may be one that has been set earlier in the parsing process, so for example:

```
input="L/look for/1 10"
pstyl input verb 2 delim +1 string (delim) rest
```

will set:

```
verb to 'L'
delim to ' /'
string to 'look for'
rest to '1 10'
```
If the left parenthesis is preceded by an equals, plus, or minus sign then the value of the variable is used as an absolute or relative positional pattern (instead of as a literal string pattern). In this case the value of the variable must be a non-negative whole number, and (as before) it may have been set earlier in the parsing process.
Numbers and Arithmetic

NetRexx arithmetic attempts to carry out the usual operations (including addition, subtraction, multiplication, and division) in as “natural” a way as possible. What this really means is that the rules followed are those that are conventionally taught in schools and colleges. However, it was found that unfortunately the rules used vary considerably (indeed much more than generally appreciated) from person to person and from application to application and in ways that are not always predictable. The NetRexx arithmetic described here is therefore a compromise which (although not the simplest) should provide acceptable results in most applications.

Introduction

Numbers can be expressed in NetRexx very flexibly (leading and trailing blanks are permitted, exponential notation may be used) and follow conventional syntax. Some valid numbers are:

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>A whole number</td>
</tr>
<tr>
<td>'-76'</td>
<td>A signed whole number</td>
</tr>
<tr>
<td>12.76</td>
<td>Some decimal places</td>
</tr>
<tr>
<td>' + 0.003 '</td>
<td>Blanks around the sign, etc.</td>
</tr>
<tr>
<td>17.</td>
<td>Equal to 17</td>
</tr>
<tr>
<td>'.5'</td>
<td>Equal to 0.5</td>
</tr>
<tr>
<td>4E+9</td>
<td>Exponential notation</td>
</tr>
<tr>
<td>0.73e-7</td>
<td>Exponential notation</td>
</tr>
</tbody>
</table>

(Exponential notation means that the number includes a sign and a power of ten following an “e” that indicates how the decimal point will be shifted. Thus 4E+9 above is just a short way of writing 4000000000, and 0.73e-7 is short for 0.000000073.)

The arithmetic operators include addition (indicated by a “+”), subtraction (”-”), multiplication (“*”), power (“**”), and division (“/”). There are also two further division operators: integer divide (“%”) which divides and returns the integer part, and remainder (“//”) which divides and returns the remainder. Prefix plus (“+”) and prefix minus (“-”) operators are also provided.

When two numbers are combined by an operation, NetRexx uses a set of rules to define what the result will be (and how the result is to be represented as a character string). These rules are defined in the next section, but in summary:

- Results will be calculated with up to some maximum number of significant digits. That is, if a result required more than 9 digits it would normally be rounded to 9 digits. For instance, the division of 2 by 3 would result in 0.666666667 (it would require an infinite number of digits for perfect accuracy).

You can change the default of 9 significant digits by using the numeric digits instruction. This lets you calculate using as many digits as you need – thousands, if necessary.

- Except for the division and power operators, trailing zeros are preserved (this is in contrast to most electronic calculators, which remove all trailing zeros in the decimal part of results). So, for example:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.40 + 2</td>
<td>4.40</td>
</tr>
<tr>
<td>2.40 - 2</td>
<td>0.40</td>
</tr>
<tr>
<td>2.40 * 2</td>
<td>4.80</td>
</tr>
<tr>
<td>2.40 / 2</td>
<td>1.2</td>
</tr>
</tbody>
</table>

This preservation of trailing zeros is desirable for most calculations (and especially financial calculations).
If necessary, trailing zeros may be easily removed with the `strip` method (see page 166), or by division by 1.

- A zero result is always expressed as the single digit '0'.
- Exponential form is used for a result depending on its value and the setting of `numeric digits` (the default is 9 digits). If the number of places needed before the decimal point exceeds this setting, or the absolute value of the number is less than 0.000001, then the number will be expressed in exponential notation; thus

```
1e+6 * 1e+6
```

results in “1E+12” instead of “1000000000000”, and

```
1 / 3E+10
```

results in “3.33333333E-11” instead of “0.0000000003333333333”.

- Any mixture of Arabic numerals (0-9) and Extra digits (see page 42) can be used for the digits in numbers used in calculations. The results are expressed using Arabic numerals.

**Definition**

This definition describes arithmetic for NetRexx strings (type `Rexx`). The arithmetic operations are identical to those defined in the ANSI standard for Rexx. 65

**Numbers**

A `number` in NetRexx is a character string that includes one or more decimal digits, with an optional decimal point. The decimal point may be embedded in the digits, or may be prefixed or suffixed to them. The group of digits (and optional point) thus constructed may have leading or trailing blanks, and an optional sign (“+” or “-”) which must come before any digits or decimal point. The sign may also have leading or trailing blanks. Thus:

```
sign ::= + | -
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digits ::= digit [digit]...
numeric ::= digits . [digits]
        | [.] digits
number ::= [blank]... [sign [blank]...]
          numeric [blank]...
```

where if the implementation supports extra digits (see page 42) these are also accepted as `digits`, providing that they represent values in the range zero through nine. In this case each extra digit is treated as though it were the corresponding character in the range 0-9.

Note that a single period alone is not a valid number.

**Precision**

The maximum number of significant digits that can result from an arithmetic operation is controlled by the `digits` keyword on the `numeric` instruction (see page 104):

```
numeric digits [expression];
```

The expression is evaluated and must result in a positive whole number. This defines the precision (number of significant digits) to which arithmetic calculations will be carried out; results will be rounded to that precision, if necessary.

If no expression is specified, then the default precision is used. The default precision is 9, that is, all implementations must support at least nine digits of precision. An implementation-dependent maximum (equal to or larger than 9) may apply: an attempt to exceed this will cause execution of the instruction to terminate with an exception. Thus if an algorithm is defined to use more than 9 digits then if the numeric digits instruction succeeds then the computation will proceed and produce identical results to any other implementation.

Note that numeric digits may set values below the default of nine. Small values, however, should be used with care – the loss of precision and rounding thus requested will affect all NetRexx computations, including (for example) the computation of new values for the control variable in loops.

In the remainder of this section, the notation digits refers to the current setting of numeric digits. This setting may also be referred to in expressions in programs by using the digits special word (see page 130).

**Arithmetic operators**

NetRexx arithmetic is effected by the operators “+”, “−”, “×”, “/”, “÷”, “%”, “//”, and “**” (add, subtract, multiply, divide, integer divide, remainder, and power) which all act upon two terms, together with the prefix operators “+” and “−” (plus and minus) which both act on a single term. The result of all these operations is a NetRexx string, of type Rexx. This section describes the way in which these operations are carried out.

Before every arithmetic operation, the term or terms being operated upon have any extra digits converted to the corresponding Arabic numeral (the digits 0-9). They then have leading zeros removed (noting the position of any decimal point, and leaving just one zero if all the digits in the number are zeros) and are then truncated to digits+1 significant digits\(^{66}\) (if necessary) before being used in the computation. The operation is then carried out under up to double that precision, as described under the individual operations below. When the operation is completed, the result is rounded if necessary to the precision specified by the numeric digits instruction.

Rounding is done in the “traditional” manner, in that the extra (guard) digit is inspected and values of 5 through 9 are rounded up, and values of 0 through 4 are rounded down.\(^{67}\)

A conventional zero is supplied preceding a decimal point if otherwise there would be no digit before it. Trailing zeros are retained for addition, subtraction, and multiplication, according to the rules given below, except that a result of zero is always expressed as the single character ‘0’. For division, insignificant trailing zeros are removed after rounding.

The format method (see page 161) is defined to allow a number to be represented in a particular format if the standard result provided by NetRexx does not meet requirements.

---

\(^{66}\) That is, to the precision set by numeric digits, plus one extra “guard” digit.

\(^{67}\) Even/odd rounding would require the ability to calculate to arbitrary precision (that is, to a precision not governed by the setting of numeric digits) at any time and is therefore not the mechanism defined for NetRexx.
**Arithmetic operation rules – basic operators**

The basic operators (addition, subtraction, multiplication, and division) operate on numbers as follows:

**Addition and subtraction**

If either number is zero then the other number, rounded to *digits* digits if necessary, is used as the result (with sign adjustment as appropriate). Otherwise, the two numbers are extended on the right and left as necessary up to a total maximum of *digits*+1 digits.

The number with smaller absolute value may therefore lose some or all of its digits on the right. The numbers are then added or subtracted as appropriate. For example:

\[
\begin{align*}
xxx.xxx & + yy.yyyyy \\
becomes: & \quad xxx.xxx00 \\
& + 00yy.yyyyy \\
& \quad ------------ \\
& \quad zzzz.zzzzz
\end{align*}
\]

The result is then rounded to *digits* digits if necessary, taking into account any extra (carry) digit on the left after an addition, but otherwise counting from the position corresponding to the most significant digit of the terms being added or subtracted. Finally, any insignificant leading zeros are removed.

The **prefix operators** are evaluated using the same rules; the operations ““+number” and “−number” are calculated as “0+number” and “0−number”, respectively.

**Multiplication**

The numbers are multiplied together (“long multiplication”) resulting in a number which may be as long as the sum of the lengths of the two operands. For example:

\[
\begin{align*}
xxx.xxx & \times yy.yyyyy \\
becomes: & \quad zzzzz.zzzzzzzzz
\end{align*}
\]

and the result is then rounded to *digits* digits if necessary, counting from the first significant digit of the result.

**Division**

For the division:

\[
yyy / xxxxx
\]

the following steps are taken: first, the number “yyy” is extended with zeros on the right until it is larger than the number “xxxxx” (with note being taken of the change in the power of ten that this implies). Thus in this example, “yyy” might become “yyy00”. Traditional long division then takes place, which can be written:

\[
\begin{align*}
\quad zzzz \\
\quad xxxxx \quad \overline{----} \\
& \quad yyyy00
\end{align*}
\]

The length of the result (“zzzz”) is such that the rightmost “z” will be at least as far right as the rightmost digit of the (extended) “y” number in the example. During the division, the “y” number will be extended further as necessary, and the “z” number (which will not include any leading zeros) may increase up to *digits*+1 digits, at which point the division stops and the result is rounded. Following completion of the

---

68 In the example, the number *yy.yyyyy* would have three digits truncated if *digits* were 5.
division (and rounding if necessary), insignificant trailing zeros are removed.

Examples:

```plaintext
/* With 'numeric digits 5' */
12+7.00     ==  19.00
1.3-1.07    ==  0.23
1.3-2.07    ==  -0.77
1.20*3      ==  3.60
7*3         ==  21
0.9*0.8     ==  0.72
1/3         ==  0.33333
2/3         ==  0.66667
5/2         ==  2.5
1/10        ==  0.1
12/12       ==  1
8.0/2       ==  4
```

**Note:** With all the basic operators, the position of the decimal point in the terms being operated upon is arbitrary. The operations may be carried out as integer operations with the exponent being calculated and applied afterwards. Therefore the significant digits of a result are not in any way dependent on the position of the decimal point in either of the terms involved in the operation.

### Arithmetic operation rules – additional operators

The operation rules for the power (“**”), integer division (“%”), and remainder (“/”) operators are as follows:

**Power**

The “**” (power) operator raises a number (on the left of the operator) to a power (on the right of the operator). The term on the right is rounded to digits digits (if necessary), and must, after any rounding, be a whole number, which may be positive, negative, or zero. If negative, the absolute value of the power is used, and then the result is inverted (divided into 1).

For calculating the power, the number is effectively multiplied by itself for the number of times expressed by the power, and finally trailing zeros are removed (as though the result were divided by one).

In practice (see note below for the reasons), the power is calculated by the process of left-to-right binary reduction. For “x**n”: “n” is converted to binary, and a temporary accumulator is set to 1. If “n” has the value 0 then the initial calculation is complete. Otherwise each bit (starting at the first non-zero bit) is inspected from left to right. If the current bit is 1 then the accumulator is multiplied by “x”. If all bits have now been inspected then the initial calculation is complete, otherwise the accumulator is squared by multiplication and the next bit is inspected. When the initial calculation is complete, the temporary result is divided into 1 if the power was negative.

The multiplications and division are done under the normal arithmetic operation rules, detailed earlier in this section, using a precision of digits+elength+1 digits. Here, elength is the length in decimal digits of the integer part of the whole number “n” (i.e., excluding any sign, decimal part, decimal point, or insignificant leading zeros, as though the operation n%1 had been carried out and any sign removed). Finally, the result is rounded to digits digits, if necessary, and insignificant trailing zeros are removed.
**Integer division**

The “%” (integer divide) operator divides two numbers and returns the integer part of the result. The result returned is defined to be that which would result from repeatedly subtracting the divisor from the dividend while the dividend is larger than the divisor. During this subtraction, the absolute values of both the dividend and the divisor are used: the sign of the final result is the same as that which would result if normal division were used.

The result returned will have no fractional part (that is, no decimal point or zeros following it). If the result cannot be expressed exactly within digits digits, the operation is in error and will fail – that is, the result cannot have more digits than the current setting of **numeric digits**. For example, `10000000000%3` requires ten digits to express the result exactly (3333333333) and would therefore fail if `digits` were 9 or smaller.

**Remainder**

The “//” (remainder) operator will return the remainder from integer division, and is defined as being the residue of the dividend after the operation of calculating integer division as just described. The sign of the remainder, if non-zero, is the same as that of the original dividend.

This operation will fail under the same conditions as integer division (that is, if integer division on the same two terms would fail, the remainder cannot be calculated).

**Examples:**

```plaintext
/* Again with 'numeric digits 5' */
2**3      ==  8
2**-3     ==  0.125
1.7**8    ==  69.758
2%3       ==  0
2.1//3     ==  2.1
10%3      ==  3
10//3     ==  1
-10//3     ==  -1
10.2//1   ==  0.2
10//0.3   ==  0.1
3.6//1.3  ==  1.0
```

**Notes:**

1. A particular algorithm for calculating powers is described, since it is efficient (though not optimal) and considerably reduces the number of actual multiplications performed. It therefore gives better performance than the simpler definition of repeated multiplication. Since results could possibly differ from those of repeated multiplication, the algorithm must be defined here so that different implementations will give identical results for the same operation on the same values. Other algorithms for this (and other) operations may always be used, so long as they give identical results to those described here.

2. The integer divide and remainder operators are defined so that they may be calculated as a by-product of the standard division operation (described above). The division process is ended as soon as the integer result is available; the residue of the dividend is the remainder.
**Numeric comparisons**

Any of the comparative operators (see page 62) may be used for comparing numeric strings. However, the strict comparisons (for example, “==” and “>>”) are not numeric comparative operators and should not normally be used for comparing numbers, since they compare from left to right and leading and trailing blanks (and leading zeros) are significant for these operators.

Numeric comparison, using the normal comparative operators, is effected by subtracting the two numbers (calculating the difference) and then comparing the result with ‘0’ – that is, the operation:

\[ A \ ? \ B \]

where “?” is any normal comparative operator, is identical to:

\[ (A - B) \ ? \ '0' \]

It is therefore the difference between two numbers, when subtracted under NetRexx subtraction rules, that determines their equality.

**Exponential notation**

The definition of numbers above (see page 141) describes “pure” numbers, in the sense that the character strings that describe numbers can be very long.

**Examples:**

```
say 10000000000 * 10000000000
/* would display: 100000000000000000000 */

say 0.00000000001 * 0.00000000001
/* would display: 0.0000000000000000000001 */
```

For both large and small numbers some form of exponential notation is useful, both to make such long numbers more readable and to make evaluation possible in extreme cases. In addition, exponential notation is used whenever the “pure” form would give misleading information. For example:

```
numeric digits 5
say 54321*54321
```

would display “2950800000” if long form were to be used. This is misleading, as it appears that the result is an exact multiple of 100000, and so NetRexx would express the result in exponential notation, in this case “2.9508E+9”.

The definition of number (see above) is therefore extended by replacing the description of numeric by the following:

```
mantissa ::= digits . [digits]  
| [.] digits
numeric ::= mantissa [E sign digits]
```

In other words, the numeric part of a number may be followed by an “E” (indicating an exponential part), a sign, and an integer following the sign that represents a power of ten that is to be applied. The “E” may be in uppercase or lowercase. Note that no blanks are permitted within this part of a number, but the integer may have leading zeros.

**Examples:**

```
12E+11  =  120000000000
12E-5   =  0.00012
12e+4   =  120000
```
All valid numbers may be used as data for arithmetic. The results of calculations will be returned in exponential form depending on the setting of numeric digits. If the number of places needed before the decimal point exceeds digits, or if the absolute value of the result is less than 0.000001, then exponential form will be used. The exponential form generated by NetRexx always has a sign following the “E”. If the exponent is 0 then the exponential part is omitted – that is, an exponential part of “E+0” will never be generated.

If the default format for a number is not satisfactory for a particular application, then the format method may be used to control its format. Using this, numbers may be explicitly converted to exponential form or even forced to be returned in “pure” form.

Different exponential notations may be selected with the numeric form instruction (see page 104). This instruction allows the selection of either scientific or engineering notation. Scientific notation adjusts the power of ten so there is a single non-zero digit to the left of the decimal point. Engineering notation causes powers of ten to be expressed as a multiple of three – the integer part may therefore range from 1 through 999.

Examples:

```
numeric form scientific
say 123.45 * 1e11 /* would display: 1.2345E+13 */
```

```
numeric form engineering
say 123.45 * 1e11 /* would display: 12.345E+12 */
```

The default exponential notation is scientific.

**Whole numbers**

Within the set of numbers understood by NetRexx it is useful to distinguish the subset defined as whole numbers.

A whole number in NetRexx is a number that has a decimal part which is all zeros (or that has no decimal part).

**Numbers used directly by NetRexx**

As discussed above, the result of any arithmetic operation is rounded (if necessary) according to the setting of numeric digits. Similarly, when a number (which has not necessarily been involved in an arithmetic operation) is used directly by NetRexx then the same rounding is also applied, just as though the operation of adding the number to 0 had been carried out. After this operation, the integer part of the number must have no more digits than the current setting of numeric digits.

In the following cases, the number used must be a whole number and an implementation restriction on the largest number that can be used may apply:

- positional patterns, including variable positional patterns, in parsing templates (see page 133)
- the power value (right hand operand) of the power operator (see page 144)
- the values of expr and exprf (following the for keyword) in the loop instruction (see page 91)
- the value of exprd (following the digits keyword) in the numeric instruction (see page 104).

**Implementation minimum**: A minimum length of 9 digits must be supported for these uses of whole
Implementation independence

The NetRexx arithmetic rules are defined in detail, so that when a given program is run the results of all computations are sufficiently defined that the same answer will result for all correct implementations. Differences due to the underlying machine architecture will not affect computations. This contrasts with most other programming languages, and with binary arithmetic (see page 149) in NetRexx, where the result obtained may depend on the implementation because the precision and algorithms used by the language processor are defined by the implementation rather than by the language.

Exceptions and errors

The following exceptions and errors may be signalled during arithmetic:

• Divide exception
  This exception will be signalled if division by zero was attempted, or if the integer result of an integer divide or remainder operation had too many digits.

• Overflow/Underflow exception
  This exception will be signalled if the exponential part of a result (from an operation that is not an attempt to divide by zero) would exceed the range that can be handled by the language processor, when the result is formatted according to the current settings of numeric digits and numeric form. The language defines a minimum capability for the exponential part, namely exponents whose absolute value is at least as large as the largest number that can be expressed as an exact integer in default precision. Thus, since the default precision is nine, implementations must support exponents in the range $-999999999$ through $999999999$.

• Insufficient storage
  Storage is needed for calculations and intermediate results, and on occasion an arithmetic operation may fail due to lack of storage. This is considered an operating environment error as usual, rather than an arithmetical exception.

In the reference implementation, the exceptions and error types used for these three cases are DivideException, ExponentOverflowException, and OutOfMemoryError, respectively.
Binary values and operations

By default, arithmetic and string operations in NetRexx are carried out using the NetRexx string class, Rexx, which offers the robust set of operators described in Expressions and operators (see page 61). NetRexx implementations, however, may also provide primitive datatypes, as described in Types and Classes (see page 46). These primitive types are used for compact storage of numbers and for fast binary arithmetic, features which are built-in to the hardware of most computers.

To make use of binary arithmetic, a class is declared to be a binary class (see page 80) by using the binary keyword on the class instruction. In such a class, literal strings and numeric symbols are assigned native string or primitive types, rather than NetRexx types, where appropriate, and native binary operations are used to implement operators where possible, as detailed below. Implementations may also provide a keyword on the options (see page 106) instruction that indicates that all classes in a program are binary classes.69

Alternatively, individual methods within a class may be declared to be a binary method (see page 101) by using the binary keyword on the method instruction.

Binary classes and methods should be used with care. Although binary arithmetic can have a considerable performance advantage over arithmetic that is not implemented in hardware, it can give incorrect or unexpected results. In particular, whole numbers (integers) are often held in fixed-sized data areas (of 8, 16, 32, or 64 bits), and overflowing the data area during a calculation can result in a positive number becoming negative and vice versa. Similarly, binary numbers that are not whole numbers (floating-point numbers) cannot exactly represent common numbers in the decimal system (0.1, 0.2, etc.), and hence can give unexpected results.

Operations in binary classes and methods

In a binary class or method, the following (and only the following) rules differ from the usual rules:

Dyadic operations in expressions

If the operands of a dyadic operator both have primitive numeric types70 then binary operations are carried out. The type of the result is implementation defined, and is typically the type of the more precise of the two operands, or of some minimum precision.71 Arithmetic operations follow the usual rules of binary arithmetic, as defined for the underlying environment of the implementation.

Note that NetRexx provides both divide and integer divide operators; in a binary class or method, the divide operator ("/") converts its operands to floating-point types and returns a floating-point result, whereas the integer divide operator ("%") converts its operands to integer types and returns an integer result. The remainder operator must accept both integer and floating-point types.

Logical operations (and, or, and exclusive or) apply to all the bits of the operands, and are not permitted on floating-point types.

Prefix operations in

If the operand of a prefix operator has a primitive numeric type, then the type of the result is the type of the operand, subject to the same minimum as dyadic operations.

69 In the reference implementation, options binary is used.
70 In the reference implementation, boolean is considered to be a numeric type (having the values 0 or 1) but char is not. Characters, and strings or arrays of characters, always use the rules defined for NetRexx strings.
71 In the reference implementation, the minimum precision is 32 bits, so an int is returned for results that would otherwise be byte or short. If both operands are boolean, however, and the operation is a logical operation, then the type of the result is boolean.
Prefix plus and minus follow the rules of dyadic operators (because they are defined as being zero plus or minus the operand) with the additional rule that if acting on a literal number (a constant in the program) then the result is also considered to be a literal constant. Logical not (prefix “\”) does not apply to all the bits of its operand; instead, it changes a 0 to 1 and vice versa.

In assignments where the value being assigned is the result of an expression which comprises a string or number literal constant, the type of the result is defined as follows:

1. Strings are given the native string type, even for a single-character literal.  
2. Numbers are given the smallest possible primitive numeric type that will contain the literal without loss of information (or minimal loss of information for numbers with decimal or exponential parts). If this is smaller than the implementation-defined minimum precision used for the result of adding the literal to 0, then the type of that minimum precision is used.

If the constant is an integer, and no primitive integer binary type has sufficient precision to hold the number without loss of information, then the number is treated as a literal string (that is, as though it were enclosed in quotes). NetRexx arithmetic would then be used if it were involved in an arithmetic operation.

These rules can apply in assignment instructions, the initial assignment to the control variable in the loop instruction, or the assignment of a default value to the argument of a method; the result type may define the type of the variable (if new, or a method argument).

In the loop instruction, if the control variable has a primitive integer type, and the increment (by value) has a primitive integer type, then binary arithmetic will be used for stepping the control variable, following the rules for binary arithmetic in expressions described above.

Similarly, if the control variable has a primitive integer type, and the end (to) value has a primitive integer type, then binary arithmetic will be used for the comparison that tests for loop termination.

The numeric instruction does not affect binary operations. It has the usual effects on operations carried out using NetRexx arithmetic.

Note: At all times (whether in binary classes, binary methods, or anywhere else) implementations may use primitive types and operations, and techniques such as late binding of types, as an optimization providing that the results obtained are identical to those defined in this language definition.

Binary constructors

NetRexx provides special constructors for implementation-defined primitive types that allow bit-wise construction of primitives. These binary constructors are especially useful for manipulating the binary encodings of individual characters.

The binary constructors follow the same syntax as other constructors, with the name being that of a primitive type. All binary constructors take one argument, which must have a primitive type.

72 In the reference implementation, this type is java.lang.String.
The bits of the value of the argument are extended or truncated on the left to the same length as the bits required for the type of the constructor (following the usual binary rules of sign extension if the argument type is a signed numeric type), and a value with the type of the constructor is then constructed directly from those bits and returned.

**Example:**

This example illustrates types from the reference implementation, with 32-bit signed integers of type `int` and 16-bit Unicode characters of type `char`.

```plaintext
i=int 77   -- i is now the integer 77
c=char(i)  -- c is now the character 'M'
j=int(c)   -- j is now the integer 77
```

Note that the conversion

```plaintext
j=int c
```

would have failed, as “M” is not a number.
Exceptions

Exceptional conditions, including errors, in NetRexx are handled by a mechanism called Exceptions. When an exceptional condition occurs, a signal takes place which may optionally be caught by an enclosing control construct, as detailed below.

An exception can be signalled by:

1. the program’s environment, when some processing error occurs (such as running out of memory, or a problem discovered when reading or writing a file)
2. a method called by a NetRexx program (if, for example, it is passed incorrect arguments)
3. the signal instruction (see page 119).

In all cases, the signal is handled in exactly the same way. First, execution of the current clause ceases; no further operations within the clause will be carried out. Next, an object that represents the exception is constructed. The type of the exception object is implementation-dependent, as described for the signal instruction (see page 119), and defines the type of the exception. The object constructed usually contains information about the Exception (such as a descriptive string).

Once the object has been constructed, all active do groups, loop loops, if constructs, and select constructs in the active method are “unwound”, starting with the innermost, until the exception is caught by a control construct that specifies a suitable catch clause (see below) for handling the exception.

This unwinding takes place as follows:

1. No further clauses within the body of the construct will be executed (in this respect, the signal acts like a leave for the construct).
2. If a catch clause specifies a type to which the exception object can be assigned (that is, it matches or is a superclass of the type of exception object), then the instructionlist following that clause is executed, and the exception is considered to be handled (no further control constructs will be unwound). If more than one catch clause specifies a suitable type, the first is used.
3. The instructionlist following the finally clause for the construct, if any, is executed.
4. The end clause is executed, hence completing execution of the construct. (The only effect of this is that it is seen when tracing.)
5. If the exception was handled, then execution resumes as though the construct completed normally. If it was not handled, then the process is repeated for any enclosing constructs.

If the exception is not caught by any of the control constructs enclosing the original point of the exception signal, then the current active method is terminated, without returning any data, and the exception is then signalled at the point where the method was invoked (that is, in the caller).

The process of unwinding control constructs and terminating the method is then repeated in each calling method until the exception is caught or the initial program invocation method (the main method) is terminated, in which case the program ends and the environment receives the signal (it would usually then display diagnostic information).

---

73 This is the only case in which an expression will not be wholly evaluated, for example.
**Syntax and example**

The constructs that may be used to handle (catch) an exception are `do` groups, `loop` loops, and `select` constructs. Specifically, as shown in the syntax diagrams (q.v.), where the `end` clause can appear in these constructs, zero or more `catch` clauses can be used to define exception handlers, followed by zero or one `finally` clauses that describe “clean-up” code for the construct. The whole construct continues to be ended by an `end` clause.

The syntax of a `catch` clause is shown in the syntax diagrams. It always specifies an `exception` type, which may be qualified. It may optionally specify a symbol, `vare`, which is followed by an equals sign. This indicates that when the exception is caught then the object representing the exception will be assigned to the variable `vare`. If new, the type of the variable will be `exception`.

Here is an example of a program that handles some of the exceptions signalled by methods in the `Rexx` class; the `trace results` instruction is included to show the flow of execution:

```plaintext
trace results
do                -- could be LOOP i=1 to 10, etc.
    say 1/arg
    catch DivideException
        say 'Divide exception'
    catch ex=NumberFormatException
        /* 'ex' is assigned the exception object */
        say 'Bad number for division:' ex.getMessage
    finally
        say 'Done!'  
end
```

In this example, if the argument passed to the program (and hence placed in `arg`) is a valid number, then its inverse is displayed. If the argument is 0, then “Divide exception” would be displayed. If the argument were an invalid number, the message describing the bad number would be displayed. For any other exception (such as an `ExponentOverflowException`), the program would end and the environment would normally report the exception.

In all cases, the message “Done!” would be displayed; this would be true even if the body of the `do` construct executed a `return`, `leave`, or `iterate` instruction. Only an `exit` instruction (see page 84) would cause immediate termination of the construct (and the program).

**Note:** The `finally` keyword, like `otherwise` in the `select` construct, implies a semicolon after it, so the last `say` instruction in the example could have appeared on the same line as the `finally` without an intervening semicolon.

**Exceptions after catch and finally clauses**

If an exception is signalled in the `instructionlist` following a `catch` or `finally` clause, then the current exception is considered handled, the `instructionlist` is terminated, and the new exception is signalled. It will not be caught by `catch` clauses in the current construct. If it occurs in the `instructionlist` following a `catch` clause, then any `finally` instructions will be executed, as usual.

Similarly, executing a `return` or `exit` instruction within either of the `instructionlists` completes the handling of any pending signal.

**Checked exceptions**

NetRexx implementations may define certain exceptions as checked exceptions. These are exceptions that the implementation considers it useful to check; the checked exceptions that each method may
signal are recorded. Within do groups, loop loops, and select constructs, for example, it is then possible to report if a catch clause tries to catch a checked exception that is not signalled within the body of the construct.

Checked exceptions that are signalled within a method (by a signal instruction or a method invocation) but not caught by a catch clause in the method are automatically added to the signals list for a method. Implementations that support checked exceptions are encouraged to provide options that list the uncaught checked exceptions for methods or enforce the explicit inclusion of some or all checked exceptions in the signals list on the method instruction.

In the reference implementation, all exceptions are checked except those that are subclasses of java.lang.RuntimeException or java.lang.Error. These latter are considered so ubiquitous that almost all methods would signal them.

Expressions assigned as the initial values of properties must not invoke methods that may signal checked exceptions.

The strictsignal option on the options instruction may be used to enforce the inclusion of all uncaught checked exceptions in methods’ signals lists; this may be used to assure that any uncaught checked exceptions are intentional.
Methods for NetRexx strings

This section describes the set of methods defined for the NetRexx string class, Rexx. These are called built-in methods, and include character manipulation, word manipulation, conversion, and arithmetic methods.

Implementations will also provide other methods for the Rexx class (for example, to implement the NetRexx operators or to provide constructors with primitive arguments), but these are not part of the NetRexx language. 74

General notes on the built-in methods:

1. All methods work on a NetRexx string of type Rexx; this is referred to by the name string in the descriptions of the methods. For example, if the word method were invoked using the term: "Three word phrase".word(2)
then in the description of word the name string refers to the string “Three word phrase”, and the name n refers to the string “2”.

2. All method arguments are of type Rexx and all methods return a string of type Rexx; if a number is returned, it will be formatted as though 0 had been added with no rounding.

3. The first parenthesis in a method call must immediately follow the name of the method, with no space in between.

4. The parentheses in a method call can be omitted if no arguments are required and the method call is part of a compound term (see page 48). 75

5. A position in a string is the number of a character in the string, where the first character is at position 1, etc.

6. Where arguments are optional, commas may only be included between arguments that are present (that is, trailing commas in argument lists are not permitted).

7. A pad argument, if specified, must be exactly one character long.

8. If a method has a sub-option selected by the first character of a string, that character may be in upper or lowercase.

9. Conversion between character encodings and decimal or hexadecimal is dependent on the machine representation (encoding) of characters and hence will return appropriately different results for Unicode, ASCII, EBCDIC, and other implementations.

74 Details of the methods provided in the reference implementation are included in Appendix C (see page 179).
75 Unless an implementation-provided option to disallow parenthesis omission is in force.
The built-in methods

**abbrev(info[,length])**

returns 1 if `info` is equal to the leading characters of `string` and `info` is not less than the minimum length, `length`; 0 is returned if either of these conditions is not met. `length` must be a non-negative whole number; the default is the length of `info`.

**Examples:**

```
'Print'.abbrev('Pri')   == 1
'PRINT'.abbrev('Pri')   == 0
'PRINT'.abbrev('PRI',4) == 0
'PRINT'.abbrev('PRY')   == 0
'PRINT'.abbrev('')     == 1
'PRINT'.abbrev('',1)    == 0
```

**Note:** A null string will always match if a length of 0 (or the default) is used. This allows a default keyword to be selected automatically if desired.

**Example:**

```
say 'Enter option:'; option=ask
select /* keyword1 is to be the default */
    when 'keyword1'.abbrev(option) then ...
    when 'keyword2'.abbrev(option) then ...
    ...
    otherwise ...
end
```

**abs()**

returns the absolute value of `string`, which must be a number.

Any sign is removed from the number, and it is then formatted by adding zero with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). Scientific notation is used, if necessary.

**Examples:**

```
'12.3'.abs            == 12.3
'-0.307'.abs          == 0.307
'123.45E+16'.abs      == 1.2345E+18
'-1234567.7654321'.abs== 1234567.7654321
```

**b2x()**

Binary to hexadecimal. Converts `string`, a string of at least one binary (0 and/or 1) digits, to an equivalent string of hexadecimal characters. The returned string will use uppercase Roman letters for the values A-F, and will not include any blanks.

If the number of binary digits in the string is not a multiple of four, then up to three ‘0’ digits will be added on the left before conversion to make a total that is a multiple of four.

**Examples:**

```
'11000011'.b2x  == 'C3'
'10111'.b2x     == '17'
'01011'.b2x     == '5'
'1011'.b2x      == '5'
'111110000'.b2x == '1F0'
```
center(length [,pad]) or centre(length [,pad]) returns a string of length length with string centered in it, with pad characters added as necessary to make up the required length. length must be a non-negative whole number. The default pad character is blank. If the string is longer than length, it will be truncated at both ends to fit. If an odd number of characters are truncated or added, the right hand end loses or gains one more character than the left hand end.

Examples:

'ABC'.centre(7)          == '  ABC  '  
'ABC'.center(8,'-')      == '--ABC---'  
'The blue sky'.centre(8) == 'e blue s'  
'The blue sky'.center(7) == 'e blue '  

Note: This method may be called either centre or center, which avoids difficulties due to the difference between the British and American spellings.

changestr(needle, new) returns a copy of string in which each occurrence of the needle string is replaced by the new string. Each unique (non-overlapping) occurrence of the needle string is changed, searching from left to right and starting from the first (leftmost) position in string. Only the original string is searched for the needle, and each character in string can only be included in one match of the needle.

If the needle is the null string, the result is a copy of string, unchanged.

Examples:

'elephant'.changestr('e','X')    == 'XlXphant'  
'elephant'.changestr('ph','X')   == 'eleXant'  
'elephant'.changestr('ph','hph') == 'elephphant'  
'elephant'.changestr('e','!!')   == 'lphant'  
'elephant'.changestr('','!!')    == 'elephant'  

The countstr method (see page 158) can be used to count the number of changes that could be made to a string in this fashion.

compare(target [,pad]) returns 0 if string and target are the same. If they are not, the returned number is positive and is the position of the first character that is not the same in both strings. If one string is shorter than the other, one or more pad characters are added on the right to make it the same length for the comparison. The default pad character is a blank.

Examples:

'abc'.compare('abc')      == 0  
'abc'.compare('ak')       == 2  
'ab '.compare('ab')       == 0  
'ab '.compare('ab','!')   == 0  
'ab '.compare('ab','x')   == 3  
'ab-- '.compare('ab','-') == 5  

copies(n) returns n directly concatenated copies of string. n must be positive or 0; if 0, the null string is returned.

Examples:

'abc'.copies(3) == 'abcabcabc'  
'abc'.copies(0) == ''
copyindexed(sub) copies the collection of indexed sub-values (see page 73) of sub into the collection associated with string, and returns the modified string. The resulting collection is the union of the two collections (that is, it contains the indexes and their values from both collections). If a given index exists in both collections then the sub-value of string for that index is replaced by the sub-value from sub.

The non-indexed value of string is not affected.

Example:

Following the instructions:

```plaintext
foo='def'
foo['a']=1
foo['b']=2
bar='ghi'
bar['b']='B'
bar['c']='C'
merged=foo.copyIndexed(bar)
```

then:

```plaintext
merged['a'] == '1'
merged['b'] == 'B'
merged['c'] == 'C'
merged['d'] == 'def'
```

countstr(needle) returns the count of non-overlapping occurrences of the needle string in string, searching from left to right and starting from the first (leftmost) position in string.

If the needle is the null string, 0 is returned.

Examples:

```plaintext
'elephant'.countstr('e')  == '2'
'elephant'.countstr('ph') == '1'
'elephant'.countstr('')   == '0'
```

The changestr method (see page 157) can be used to change occurrences of needle to some other string.

c2d() Coded character to decimal. Converts the encoding of the character in string (which must be exactly one character) to its decimal representation. The returned string will be a non-negative number that represents the encoding of the character and will not include any sign, blanks, insignificant leading zeros, or decimal part.

Examples:

```plaintext
'M'.c2d  == '77'  -- ASCII or Unicode
'7'.c2d  == '247' -- EBCDIC
'\r'.c2d == '13'  -- ASCII or Unicode
'\0'.c2d == '0'
```

The c2x method (see page 158) can be used to convert the encoding of a character to a hexadecimal representation.

c2x() Coded character to hexadecimal. Converts the encoding of the character in string (which must be exactly one character) to its hexadecimal representation (unpacks). The returned string will use uppercase Roman letters for the values A-
F, and will not include any blanks. Insignificant leading zeros are removed.

Examples:

- `'M'.c2x == '4D'` -- ASCII or Unicode
- `'7'.c2x == 'F7'` -- EBCDIC
- `'\r'.c2x == 'D'` -- ASCII or Unicode
- `'\0'.c2x == '0'

The `c2d` method (see page 158) can be used to convert the encoding of a character to a decimal number.

`datatype(option)` returns 1 if `string` matches the description requested with the `option`, or 0 otherwise. If `string` is the null string, 0 is always returned.

Only the first character of `option` is significant, and it may be in either uppercase or lowercase. The following `option` characters are recognized:

A (Alphanumeric); returns 1 if `string` only contains characters from the ranges “a-z”, “A-Z”, and “0-9”.

B (Binary); returns 1 if `string` only contains the characters “0” and/or “1”.

D (Digits); returns 1 if `string` only contains characters from the range “0-9”.

L (Lowercase); returns 1 if `string` only contains characters from the range “a-z”.

M (Mixed case); returns 1 if `string` only contains characters from the ranges “a-z” and “A-Z”.

N (Number); returns 1 if `string` is a syntactically valid NetRexx number that could be added to `0` without error,

S (Symbol); returns 1 if `string` only contains characters that are valid in non-numeric symbols (the alphanumeric characters and underscore), and does not start with a digit. Note that both uppercase and lowercase letters are permitted.

U (Uppercase); returns 1 if `string` only contains characters from the range “A-Z”.

W (Whole Number); returns 1 if `string` is a syntactically valid NetRexx number that can be added to `0` without error, and whose decimal part after that addition, with no rounding, is zero.

X (heXadecimal); returns 1 if `string` only contains characters from the ranges “a-f”, “A-F”, and “0-9”.

Examples:

- `'101'.datatype('B') == 1`
- `'12.3'.datatype('D') == 0`
- `'12.3'.datatype('N') == 1`
- `'12.3'.datatype('W') == 0`
- `'LaArca'.datatype('M') == 1`
- `'1'.datatype('M') == 0`
- `'Llanes'.datatype('L') == 0`
- `'3 d'.datatype('s') == 1`
- `'BCd3'.datatype('X') == 1`
- `'BCgd3'.datatype('X') == 0"
**Note:** The `datatype` method tests the meaning of the characters in a string, independent of the encoding of those characters. Extra letters and extra digits cause `datatype` to return 0 except for the number tests (“N” and “W”), which treat extra digits whose value is in the range 0-9 as though they were the corresponding Arabic numeral.

**delstr(n [,length])** returns a copy of `string` with the sub-string of `string` that begins at the `n`th character, and is of length `length` characters, deleted. If `length` is not specified, or is greater than the number of characters from `n` to the end of the string, the rest of the string is deleted (including the `n`th character). `length` must be a non-negative whole number, and `n` must be a positive whole number. If `n` is greater than the length of `string`, the string is returned unchanged.

**Examples:**

```plaintext
abcd'.delstr(3) == 'ab'
abcde'.delstr(3,2) == 'abe'
abcde'.delstr(6) == 'abcde'
```

**delword(n [,length])** returns a copy of `string` with the sub-string of `string` that starts at the `n`th word, and is of length `length` blank-delimited words, deleted. If `length` is not specified, or is greater than number of remaining words in the string, it defaults to be the remaining words in the string (including the `n`th word). `length` must be a non-negative whole number, and `n` must be a positive whole number. If `n` is greater than the number of words in `string`, the string is returned unchanged. The string deleted includes any blanks following the final word involved, but none of the blanks preceding the first word involved.

**Examples:**

```plaintext
'Now is the time'.delword(2,2) == 'Now time'
'Now is the time '.delword(3) == 'Now is '
'Now  time'.delword(5) == 'Now  time'
```

**d2c()** Decimal to coded character. Converts the `string` (a NetRexx number) to a single character, where the number is used as the encoding of the character.

`string` must be a non-negative whole number. An error results if the encoding described does not produce a valid character for the implementation (for example, if it has more significant bits than the implementation’s encoding for characters).

**Examples:**

```plaintext
'77'.d2c == 'M' -- ASCII or Unicode
'+77'.d2c == 'M' -- ASCII or Unicode
'247'.d2c == '7' -- EBCDIC
'0'.d2c == '\0'
```

**d2x([n])** Decimal to hexadecimal. Returns a string of hexadecimal characters of length as needed or of length `n`, which is the hexadecimal (unpacked) representation of the decimal number. The returned string will use uppercase Roman letters for the values A-F, and will not include any blanks.

`string` must be a whole number, and must be non-negative unless `n` is specified, or an error will result. If `n` is not specified, the length of the result returned is such that there are no leading 0 characters, unless `string` was equal to 0 (in which case the result returned is a single zero character).
case '0' is returned).

If $n$ is specified it is the length of the final result in characters; that is, after conversion the input string will be sign-extended to the required length (negative numbers are converted assuming two's-complement form). If the number is too big to fit into $n$ characters, it will be truncated on the left. $n$ must be a non-negative whole number.

**Examples:**

- `'9'.d2x` == `'9'`
- `'129'.d2x` == `'81'`
- `'129'.d2x(1)` == `'1'`
- `'129'.d2x(2)` == `'81'`
- `'127'.d2x(3)` == `'07F'`
- `'129'.d2x(4)` == `'0081'`
- `'257'.d2x(2)` == `'01'`
- `'−127'.d2x(2)` == `'81'`
- `'−127'.d2x(4)` == `'FF81'`
- `'12'.d2x(0)` == `' '``

**exists(index)** returns 1 if `index` names a sub-value (see page 73) of `string` that has explicitly been assigned a value, or 0 otherwise.

**Example:**

Following the instructions:

```plaintext```
    vowel=0
    vowel['a']=1
    vowel['b']=1
    vowel['b']=null -- drops previous assignment
```

then:

```plaintext```
    vowel.exists('a') == '1'
    vowel.exists('b') == '0'
    vowel.exists('c') == '0'
```

**format([before [,after]])** formats (lays out) `string`, which must be a number.

The number, `string`, is first formatted by adding zero with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). If no arguments are given, the result is precisely that of this operation.

The arguments `before` and `after` may be specified to control the number of characters to be used for the integer part and decimal part of the result respectively. If either of these is omitted (with no arguments specified to its right), or is `null`, the number of characters used will be as many as are needed for that part.

`before` must be a positive number; if it is larger than is needed to contain the integer part, that part is padded on the left with blanks to the requested length. If `before` is not large enough to contain the integer part of the number (including the sign, for negative numbers), an error results.

`after` must be a non-negative number; if it is not the same size as the decimal part of the number, the number will be rounded (or extended with zeros) to fit. Specifying 0 for `after` will cause the number to be rounded to an integer (that is,
it will have no decimal part or decimal point).

Examples:

' - 12.73'.format               == '-12.73'
'0.000'.format                  == '0'
'3'.format(4)                   == '  3'
'1.73'.format(4,0)              == '  2'
'1.73'.format(4,3)              == ' 1.730'
'-.76'.format(4,1)              == ' -0.8'
'3.03'.format(4)                == '  3.03'
' - 12.73'.format(null,4)      == '-12.7300'

Further arguments may be passed to the format method to control the use of exponential notation. The full syntax of the method is then:

format([before[,after[,explaces[,exdigits[,exform]]]]])

The first two arguments are as already described. The other three (explaces, exdigits, and exform) control the exponent part of the result. The default for any of the arguments may be selected by omitting them (if there are no arguments to be specified to their right) or by using the value null.

explaces must be a positive number; it sets the number of places (digits after the sign of the exponent) to be used for any exponent part, the default being to use as many as are needed. If explaces is specified and is not large enough to contain the exponent, an error results. If explaces is specified and the exponent will be 0, then explaces+2 blanks are supplied for the exponent part of the result.

exdigits sets the trigger point for use of exponential notation. If, after the first formatting, the number of places needed before the decimal point exceeds exdigits, or if the absolute value of the result is less than 0.000001, then exponential form will be used, provided that exdigits was specified. When exdigits is not specified, exponential notation will never be used. The current setting of numeric digits may be used for exdigits by specifying the special word digits (see page 130). If 0 is specified for exdigits, exponential notation is always used unless the exponent would be 0.

exform sets the form for exponential notation (if needed). exform may be either 'Scientific' (the default) or 'Engineering'. Only the first character of exform is significant and it may be in uppercase or in lowercase. The current setting of numeric form may be used by specifying the special word form (see page 130). If engineering form is in effect, up to three digits (plus sign) may be needed for the integer part of the result (before).

Examples:

'12345.73'.format(null,null,2,2) == '1.234573E+04'
'12345.73'.format(null,3,null,0) == '1.235E+4'
'1.234573'.format(null,3,null,0) == '1.235'
'123.45'.format(null,3,2,0)     == '1.235E+02'
'1234.5'.format(null,3,2,0,'e') == '1.235E+03'
'1.2345'.format(null,3,2,0)    == '1.235'
'12345.73'.format(null,null,3,6) == '12345.73'
'12345e+5'.format(null,3)      == '1234500000.000'

Implementation minimum: If exponents are supported in an implementation, then they must be supported for exponents whose absolute value is at least as
large as the largest number that can be expressed as an exact integer in default precision, i.e., 999999999. Therefore, values for `explaces` of up to 9 should also be supported.

**insert(new [,n [,.length [,pad]]])**
inserts the string `new`, padded or truncated to length `length`, into a copy of the target `string` after the `n`th character; the string with any inserts is returned. `length` and `n` must be non-negative whole numbers. If `n` is greater than the length of the target string, padding is added before the `new` string also. The default value for `n` is 0, which means insert before the beginning of the string. The default value for `length` is the length of `new`. The default `pad` character is a blank.

**Examples:**
```
'abc'.insert('123')               == '123abc'
'abcdef'.insert(' ',3)            == 'abc def'
'abc'.insert('123',5,6)           == 'abc 123'
'abc'.insert('123',5,6,'+')       == 'abc++123+++'
'abc'.insert('123',0,5,'-')       == '123--abc'
```

**lastpos(needle [,start])**
returns the position of the last occurrence of the string `needle` in `string` (the “haystack”), searching from right to left. If the string `needle` is not found, or is the null string, 0 is returned. By default the search starts at the last character of `string` and scans backwards. This may be overridden by specifying `start`, the point at which to start the backwards scan. `start` must be a positive whole number, and defaults to the value `string.length` if larger than that value or if not specified (with a minimum default value of one).

**Examples:**
```
'abc def ghi'.lastpos(' ')   == 8
'abc def ghi'.lastpos(' ',7) == 4
'abcdefg'.lastpos(' ')      == 0
'abcdefg'.lastpos('cd')     == 3
''.lastpos('?')             == 0
```

**left(length [,pad])**
returns a string of length `length` containing the left-most `length` characters of `string`. The string is padded with `pad` characters (or truncated) on the right as needed. The default `pad` character is a blank. `length` must be a non-negative whole number. This method is exactly equivalent to `string.substr(1, length [, pad])`.

**Examples:**
```
'abc d'.left(8)     == 'abc d   '
'abc d'.left(8,'.') == 'abc d...'
'abc defg'.left(6)  == 'abc de'
```

**length()**
returns the number of characters in `string`.

**Examples:**
```
'abcdefg'.length == 8
''.length       == 0
```

**lower([n [,length]])**
returns a copy of `string` with any uppercase characters in the sub-string of `string` that begins at the `n`th character, and is of length `length` characters, replaced by their lowercase equivalent.

`n` must be a positive whole number, and defaults to 1 (the first character in
string). If \( n \) is greater than the length of string, the string is returned unchanged.

\( \text{length} \) must be a non-negative whole number. If \( \text{length} \) is not specified, or is greater than the number of characters from \( n \) to the end of the string, the rest of the string (including the \( n \)th character) is assumed.

**Examples:**

\[
\begin{align*}
'\text{SumA}'\.lower & \quad == \quad '\text{suma}' \\
'\text{SumA}'\.lower(2) & \quad == \quad '\text{Suma}' \\
'\text{SuMB}'\.lower(1,1) & \quad == \quad '\text{suMB}' \\
'\text{SUMB}'\.lower(2,2) & \quad == \quad '\text{SumB}' \\
'.'\.lower & \quad == \quad '.'
\end{align*}
\]

**max(number)** returns the larger of string and number, which must both be numbers. If they compare equal (that is, when subtracted, the result is 0), then string is selected for the result.

The comparison is effected using a numerical comparison with a digits setting that is either nine or, if greater, the larger of the number of digits in the mantissas of the two numbers (excluding leading insignificant zeros).

The selected result is formatted by adding zero to the selected number with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). Scientific notation is used, if necessary.

**Examples:**

\[
\begin{align*}
0\.max(1) & \quad ==1 \\
'1'\.max(1) & \quad ==1 \\
'+1'\.max(-1) & \quad ==1 \\
'1.0'\.max(1.00) & \quad =='1.0' \\
'1.00'\.max(1.0) & \quad =='1.00' \\
'123456700000'.max(1234567E+5) & \quad == '123456700000' \\
'1234567E+5'.max('123456700000') & \quad == '1.234567E+11'
\end{align*}
\]

**min(number)** returns the smaller of string and number, which must both be numbers. If they compare equal (that is, when subtracted, the result is 0), then string is selected for the result.

The comparison is effected using a numerical comparison with a digits setting that is either nine or, if greater, the larger of the number of digits in the mantissas of the two numbers (excluding leading insignificant zeros).

The selected result is formatted by adding zero to the selected number with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). Scientific notation is used, if necessary.

**Examples:**

\[
\begin{align*}
0\.min(1) & \quad ==0 \\
'1'\.min(1) & \quad =='1' \\
'+1'\.min(-1) & \quad =='1' \\
'1.0'\.min(1.00) & \quad =='1.0' \\
'1.00'\.min(1.0) & \quad =='1.00' \\
'123456700000'.min(1234567E+5) & \quad == '123456700000' \\
'1234567E+5'.min('123456700000') & \quad == '1.234567E+11'
\end{align*}
\]

**overlay(new [,n])** overlays the string new, padded or truncated to length length, onto a copy of the
target \textit{string} starting at the \textit{n}th character; the string with any overlays is returned. Overlays may extend beyond the end of the original \textit{string}. If \textit{length} is specified it must be a non-negative whole number. If \textit{n} is greater than the length of the target string, padding is added before the \textit{new} string also. The default \textit{pad} character is a blank, and the default value for \textit{n} is 1. \textit{n} must be greater than 0. The default value for \textit{length} is the length of \textit{new}.

**Examples:**

\begin{verbatim}
'abcdef'.overlay(' ',3)      == 'ab def'
'abcdef'.overlay('.',3,2)    == 'ab. ef'
'abcd'.overlay('qq')        == 'qqcd'
'abcd'.overlay('qq',4)      == 'abcqq'
'abc'.overlay('123',5,6,'+') == 'abc+123+++'
\end{verbatim}

\textbf{pos(needle [,start])} returns the position of the string \textit{needle}, in \textit{string} (the \textit{“haystack”}), searching from left to right. If the string \textit{needle} is not found, or is the null string, 0 is returned. By default the search starts at the first character of \textit{string} (that is, \textit{start} has the value 1). This may be overridden by specifying \textit{start} (which must be a positive whole number), the point at which to start the search; if \textit{start} is greater than the length of \textit{string} then 0 is returned.

**Examples:**

\begin{verbatim}
'Saturday'.pos('day')    == 6
'abc def ghi'.pos('x')   == 0
'abc def ghi'.pos(' ')   == 4
'abc def ghi'.pos(' ',5) == 8
\end{verbatim}

\textbf{reverse()} returns a copy of \textit{string}, swapped end for end.

**Examples:**

\begin{verbatim}
'ABc.'.reverse        == '.cBA'
'XYZ '.reverse        == ' ZYX'
'Tranquility'.reverse == 'ytiliuqnarT'
\end{verbatim}

\textbf{right(length [,pad])} returns a string of length \textit{length} containing the right-most \textit{length} characters of \textit{string} – that is, padded with \textit{pad} characters (or truncated) on the left as needed. The default \textit{pad} character is a blank. \textit{length} must be a non-negative whole number.

**Examples:**

\begin{verbatim}
'abc d'.right(8)  == ' abc d'
'abc def'.right(5) == 'c def'
'12'.right(5,'0') == '00012'
\end{verbatim}

\textbf{sequence(final)} returns a string of all characters, in ascending order of encoding, between and including the character in \textit{string} and the character in \textit{final}. \textit{string} and \textit{final} must be single characters; if \textit{string} is greater than \textit{final}, an error is reported.

**Examples:**

\begin{verbatim}
'a'.sequence('f')       == 'abcdef'
'0'.sequence('x03')     == '\x00\x01\x02\x03'
'\ufffe'.sequence('\uffff') == '\ufffe\uffff'
\end{verbatim}

\textbf{sign()} returns a number that indicates the sign of \textit{string}, which must be a number. \textit{string} is first formatted, just as though the operation \textit{“string+0”} had been carried out with sufficient digits to avoid rounding. If the number then starts with ‘-’ then
' -1' is returned; if it is '0' then '0' is returned; and otherwise '1' is returned.

Examples:

'12.3'.sign    == 1
'0.0'.sign     == 0
' -0.307'.sign == -1

space([n [,pad]]) returns a copy of string with the blank-delimited words in string formatted with n (and only n) pad characters between each word. n must be a non-negative whole number. If n is 0, all blanks are removed. Leading and trailing blanks are always removed. The default for n is 1, and the default pad character is a blank.

Examples:

'abc def '.space           == 'abc def'
' abc def '.space(3)      == ' abc def'
' abc def '.space(1)      == ' abc def'
' abc def '.space(0)      == 'abcdef'
' abc def '.space(2, '+') == 'abc++def'

strip([option [,char]]) returns a copy of string with Leading, Trailing, or Both leading and trailing characters removed, when the first character of option is L, T, or B respectively (these may be given in either uppercase or lowercase). The default is B. The second argument, char, specifies the character to be removed, with the default being a blank. If given, char must be exactly one character long.

Examples:

'  ab c  '.strip        == 'ab c'
'  ab c  '.strip('L')   == 'ab c  '
'  ab c  '.strip('t')   == '  ab c'
'12.70000'.strip('t',0) == '12.7'
'0012.700'.strip('b',0) == '12.7'

substr(n [,length [,pad]]) returns the sub-string of string that begins at the n
th character, and is of length length, padded with pad characters if necessary. n must be a positive whole number, and length must be a non-negative whole number. If n is greater than string.length, then only pad characters can be returned.

If length is omitted it defaults to be the rest of the string (or 0 if n is greater than the length of the string). The default pad character is a blank.

Examples:

'abc'.substr(2)       == 'bc'
'abc'.substr(2,4)     == 'bc '
'abc'.substr(5,4)     == ' '
'abc'.substr(2,6,'.') == 'bc....'
'abc'.substr(5,6,'.') == '......'

Note: In some situations the positional (numeric) patterns of parsing templates are more convenient for selecting sub-strings, especially if more than one sub-string is to be extracted from a string.

subword(n [,length]) returns the sub-string of string that starts at the n
th word, and is up to length blank-delimited words long. n must be a positive whole number; if greater than the number of words in the string then the null string is returned. length must be a non-negative whole number. If length is omitted it defaults to be the remaining words in the string. The returned string will never have leading or trailing blanks,
translate(tableo, tablei [,pad]) returns a copy of string with each character in string either unchanged or translated to another character.

The translate method acts by searching the input translate table, tablei, for each character in string. If the character is found in tablei (the first, leftmost, occurrence being used if there are duplicates) then the corresponding character in the same position in the output translate table, tableo, is used in the result string; otherwise the original character found in string is used. The result string is always the same length as string.

The translate tables may be of any length, including the null string. The output table, tableo, is padded with pad or truncated on the right as necessary to be the same length as tablei. The default pad is a blank.

Examples:

'abbc'.translate('&','b')           == 'a&&c'
'abcdef'.translate('12','ec')       == 'ab2d1f'
'abcdef'.translate('12','abcd','.') == '12..ef'
'4123'.translate('abcd','1234')     == 'dabc'
'4123'.translate('hods','1234')     == 'shod'

Note: The last two examples show how the translate method may be used to move around the characters in a string. In these examples, any 4-character string could be specified as the first argument and its last character would be moved to the beginning of the string. Similarly, the term:

'gh.ef.abcd'.translate(19970827,'abcdefgh')

(which returns “27.08.1997”) shows how a string (in this case perhaps a date) might be re-formatted and merged with other characters using the translate method.

trunc([n]) returns the integer part of string, which must be a number, with n decimal places (digits after the decimal point). n must be a non-negative whole number, and defaults to zero.

The number string is formatted by adding zero with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). It is then truncated to n decimal places (or trailing zeros are added if needed to make up the specified length). If n is 0 (the default) then an integer with no decimal point is returned. The result will never be in exponential form.

Examples:

'12.3'.trunc         == 12
'127.09782'.trunc(3) == 127.097
'127.1'.trunc(3)     == 127.100
'127'.trunc(2)       == 127.00
'0'.trunc(2)         == 0.00
**upper([n [,length]])** returns a copy of *string* with any lowercase characters in the sub-string of *string* that begins at the *n*th character, and is of length *length* characters, replaced by their uppercase equivalent.

*n* must be a positive whole number, and defaults to 1 (the first character in *string*). If *n* is greater than the length of *string*, the string is returned unchanged.

*length* must be a non-negative whole number. If *length* is not specified, or is greater than the number of characters from *n* to the end of the string, the rest of the string (including the *n*th character) is assumed.

**Examples:**

```
'Fou-Baa'.upper          == 'FOU-BAA'
'Mad Sheep'.upper        == 'MAD SHEEP'
'Mad sheep'.upper(5)     == 'Mad SHEEP'
'Mad sheep'.upper(5,1)   == 'Mad Shep'
'Mad sheep'.upper(5,4)   == 'Mad SHEep'
'tinganon'.upper(1,1)    == 'Tinganon'
''.upper                 == ''
```

**verify(reference [,option [,start]])** verifies that *string* is composed only of characters from *reference*, by returning the position of the first character in *string* that is not also in *reference*. If all the characters were found in *reference*, 0 is returned.

The *option* may be either 'Nomatch' (the default) or 'Match'. Only the first character of *option* is significant and it may be in uppercase or in lowercase. If 'Match' is specified, the position of the first character in *string* that is in *reference* is returned, or 0 is returned if none of the characters were found.

The default for *start* is 1 (that is, the search starts at the first character of *string*). This can be overridden by giving a different *start* point, which must be positive.

If *string* is the null string, the method returns 0, regardless of the value of the *option*. Similarly if *start* is greater than *string*.length, 0 is returned.

If *reference* is the null string, then the returned value is the same as the value used for *start*, unless 'Match' is specified as the *option*, in which case 0 is returned.

**Examples:**

```
'123'.verify('1234567890')          == 0
'123'.verify('1234567890')          == 2
'AB4T'.verify('1234567890','M')     == 3
'1P3Q4'.verify('1234567890','N',3)  == 4
'ABCD'E'.verify('','n',3)           == 3
'AB3CD5'.verify('1234567890','m',4) == 6
```

**word(n)** returns the *n*th blank-delimited word in *string*. *n* must be positive. If there are fewer than *n* words in *string*, the null string is returned. This method is exactly equivalent to *string*.subword(*n*,1).

**Examples:**

```
'Now is the time'.word(3) == 'the'
'Now is the time'.word(5) == ''
```

**wordindex(n)** returns the character position of the *n*th blank-delimited word in *string*. *n* must be positive. If there are fewer than *n* words in the string, 0 is returned.
Examples:

'Now is the time'.wordindex(3) == 8
'Now is the time'.wordindex(6) == 0

wordlength(n) returns the length of the nth blank-delimited word in string. n must be positive. If there are fewer than n words in the string, 0 is returned.

Examples:

'Now is the time'.wordlength(2) == 2
'Now comes the time'.wordlength(2) == 5
'Now is the time'.wordlength(6) == 0

wordpos(phrase [,start]) searches string for the first occurrence of the sequence of blank-delimited words phrase, and returns the word number of the first word of phrase in string. Multiple blanks between words in either phrase or string are treated as a single blank for the comparison, but otherwise the words must match exactly. Similarly, leading or trailing blanks on either string are ignored. If phrase is not found, or contains no words, 0 is returned.

By default the search starts at the first word in string. This may be overridden by specifying start (which must be positive), the word at which to start the search.

Examples:

'now is the time'.wordpos('the') == 3
'now is the time'.wordpos('The') == 0
'now is the time'.wordpos('is the') == 2
'now is the time'.wordpos('is the') == 2
'now is the time'.wordpos('is time') == 0
'To be or not to be'.wordpos('be') == 2
'To be or not to be'.wordpos('be',3) == 6

words() returns the number of blank-delimited words in string.

Examples:

'Now is the time'.words == 4
'.words == 0
' '.words == 0

x2b() Hexadecimal to binary. Converts string (a string of at least one hexadecimal characters) to an equivalent string of binary digits. Hexadecimal characters may be any decimal digit character (0-9) or any of the first six alphabetic characters (a-f), in either lowercase or uppercase.

string may be of any length; each hexadecimal character with be converted to a string of four binary digits. The returned string will have a length that is a multiple of four, and will not include any blanks.

Examples:

'C3'.x2b == '11000011'
'7'.x2b == '0111'
'1C1'.x2b == '000111000001'

x2c() Hexadecimal to coded character. Converts the string (a string of hexadecimal characters) to a single character (packs). Hexadecimal characters may be any decimal digit character (0-9) or any of the first six alphabetic characters (a-f), in either lowercase or uppercase.
string must contain at least one hexadecimal character; insignificant leading zeros are removed, and the string is then padded with leading zeros if necessary to make a sufficient number of hexadecimal digits to describe a character encoding for the implementation.

An error results if the encoding described does not produce a valid character for the implementation (for example, if it has more significant bits than the implementation’s encoding for characters).

Examples:

'004D'.x2c == 'M' -- ASCII or Unicode
'4d'.x2c == 'M' -- ASCII or Unicode
'A2'.x2c == 'S' -- EBCDIC
'0'.x2c == '\0'

The d2c method (see page 160) can be used to convert a NetRexx number to the encoding of a character.

x2d([n])

Hexadecimal to decimal. Converts the string (a string of hexadecimal characters) to a decimal number, without rounding. If string is the null string, 0 is returned.

If n is not specified, string is taken to be an unsigned number.

Examples:

'0E'.x2d == 14
'81'.x2d == 129
'F81'.x2d == 3969
'FF81'.x2d == 65409
'c6f0'.x2d == 50928

If n is specified, string is taken as a signed number expressed in n hexadecimal characters. If the most significant (left-most) bit is zero then the number is positive; otherwise it is a negative number in twos-complement form. In both cases it is converted to a NetRexx number which may, therefore, be negative. If n is 0, 0 is always returned.

If necessary, string is padded on the left with '0' characters (note, not “sign-extended”), or truncated on the left, to length n characters; (that is, as though string.right(n, '0') had been executed.)

Examples:

'81'.x2d(2) == -127
'81'.x2d(4) == 129
'F081'.x2d(4) == -3967
'F081'.x2d(3) == 129
'F081'.x2d(2) == -127
'F081'.x2d(1) == 1
'0031'.x2d(0) == 0

The c2d method (see page 158) can be used to convert a character to a decimal representation of its encoding.
Appendix A – A Sample NetRexx Program

This appendix includes a short program, called qtime, which is an example of a “real” NetRexx program. The programs included elsewhere in this document have been contrived to illustrate specific points. By contrast, qtime is a simple but useful tool that genuinely improves the human factors of computer systems. People frequently wish to know the time of day, and this program presents this information in a natural way.

The style used for this example is the same as that used throughout the document, with all symbols except those describing classes being written in lower case. Other NetRexx programming styles are possible, of course; NetRexx syntax is designed to permit a wide variety of styles with a minimum of punctuation.

The qtime program is a modification of one of the first Rexx programs ever written (much of the program is identical). The main changes are:

- Indexed variables (brackets notation) are used instead of Rexx stems.
- The word method from the Rexx class is used instead of the word Rexx built-in function.
- The Java Date class is used to determine the current time.
```netrexx
/*--------------------------------------------------------*/
/* qtime.nrx.  This program displays the time in English. */
/* If '?' is given as the first argument word then the */
/* program displays a description of itself.              */
/*--------------------------------------------------------*/

/*--------------------------------------------------------*/
/*-------- First process any argument words ---------------*/
parse arg parm .               /* get the first argument word */
select
  when parm='?' then tell         /* say what we do */
  when parm=' ' then nop        /* OK (no first argument) */
  otherwise
    say 'The only valid argument to QTIME is "?". The word'
    say 'that you supplied ("parm") has been ignored.'
    tell /* usually helpful to describe the program */
end

/*--------------------------------------------------------*/
/*-------- Now start processing in earnest ---------------*/
/* Nearness phrases - using associative array lookup */
near=''                       /* default */
near[0]=''                  /* exact */
/* Extract the hours, minutes, and seconds from the time. */
/* Use the Java Date class to get the time-of-day. */
parse Date() . . . now .     /* time is the fourth word */
parse now hour:'min':'sec
if sec>29 then min=min+1     /* round up minutes */
mod=min//5                   /* where we are in 5-minute bracket */
out="It's"near[mod]         /* start building the result */
if min>32 then hour=hour+1   /* we are TO the hour... */
min=min+2                    /* shift minutes to straddle a 5-minute point */
/* Now special-case the result for Noon and Midnight. */
if hour//12=0 & min//60<=4 then do
  if hour=12 then say out 'Noon.'
  else say out 'Midnight.'
  exit /* we are finished here */
end
/* Find five-minute segment and convert to 12-hour clock. */
min=min-(min//5)             /* find nearest 5 mins */
if hour>12
  then hour=hour-12          /* get rid of 24-hour clock */
else if hour=0 then hour=12 /* .. and allow for midnight */
```
/* Determine the phrase to use for each 5-minute segment. */
select
  when min= 0 then nop /* add "o'clock" later */
  when min=60 then min=0 /* ditto */
  when min= 5 then out=out 'five past'
  when min=10 then out=out 'ten past'
  when min=15 then out=out 'a quarter past'
  when min=20 then out=out 'twenty past'
  when min=25 then out=out 'twenty-five past'
  when min=30 then out=out 'half past'
  when min=35 then out=out 'twenty-five to'
  when min=40 then out=out 'twenty to'
  when min=45 then out=out 'a quarter to'
  when min=50 then out=out 'ten to'
  when min=55 then out=out 'five to'
end

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

/*--------------------------------------------------------*/
/* Tell: function that describes the use of the program. */
/*--------------------------------------------------------*/
method tell static
  say 'QTIME displays the current time in natural English.'
say 'Call without any arguments to display the time, or'
say 'with "?" to display this information.'
say 'British English idioms are used in this program.'
say /* space -- we are about to continue and show time. */
  return

/* Mike Cowlishaw, December 1979 - January 1985 */
/* NetRexx version March 1996 */
Appendix B – JavaBean Support

This appendix describes an experimental feature, indirect properties, which is supported by the NetRexx reference implementation.

The intention of the feature is to make it easier to write a certain kind of class known as a JavaBean. Almost all JavaBeans will have properties, which are data items that a user of a JavaBean is expected to be able to customize (for example, the text on a pushbutton). The names and types of the properties of a JavaBean are inferred from “design patterns” (in this context, conventions for naming methods) or from PropertyDescriptor objects associated with the JavaBean.

The JavaBean properties do not necessarily correspond to instance variables in the class – although very often they do. The JavaBean specification does not guarantee that JavaBean properties that can be set can also be inspected, nor does it describe how ambiguities of naming and method signatures are to be handled.

The NetRexxC compiler allows a more rigorous treatment of JavaBean properties, by allowing an optional attribute of properties in a class that declares them to be indirect properties. Indirect properties are properties of a known type that are private to the class, but which are expected to be publicly accessible indirectly, though certain conventional method calls.

Declaring properties to be indirect offers the following advantages:

• For many simple cases, the access methods for the properties can be generated automatically; there is no need to explicitly code them in the source file for the class. This is especially helpful for Indexed Properties (where four methods are needed, in general).

• Where access methods are explicitly provided in the class, they can be checked for correct form, signature and accessibility. This detects errors at compile time that otherwise would only be determined by testing.

• Similarly, attention can be drawn to the presence of methods that may be intended to be an access method for an indirect property, but will not be recognized as such by builders.

The next section describes the use of indirect properties in more detail.
**Indirect properties**

The **properties** instruction (see page 112) is used to define the attributes of following **property** variables. The **visibility** of properties may include a new alternative: **indirect**. Properties with this form of visibility are known as **indirect properties**. These are properties of a known type that are private to the class, but which are expected to be publicly accessible indirectly, though certain conventional method calls.

For example, consider the simple program:

```java
class Sandwich extends Canvas implements Serializable
    properties indirect
        slices=Color.gray
        filling=Color.red

    method Sandwich
        resize(100,30)

    method paint(g=Graphics)
        g.setColor(slices)
        g.fillRect(0, 0, size.width, size.height)
        g.setColor(filling)
        g.fillRect(12, 12, size.width-12, size.height-12)
```

This declares the **Sandwich** class as having two indirect properties, called **slices** and **filling**, both being of type **java.awt.Color**.

In the example, no access methods are provided for the properties, so the compiler will add them. By implementation-dependent convention, the names are prefixed with verbs such as **get** and **set**, etc., and have the first character of their name uppercased to form the method names. Hence, in this Java-based example, the following four methods are added:

```java
    method getSlices  returns java.awt.Color
        return slices

    method getFilling returns java.awt.Color
        return filling

    method setSlices($1=java.awt.Color)
        slices=$1

    method setFilling($2=java.awt.Color)
        filling=$2
```

(where $1 and $2 are “hidden” names used for accessing the method arguments).

Note that the **indirect** attribute for a property is an alternative to the **public**, **private**, and **inheritable**, and **shared** attributes. Like private properties, indirect properties can only be accessed directly by name from within the class in which they occur; other classes can only access them using the access methods (or other methods that may use, or have a side-effect on, the properties).

Indirect properties may be **constant** (implying that only a get method is generated or allowed, though the private property may be changed by methods within the class) or **transient** (see page 113). They may not be **static** or **volatile**.

In detail, the rules used for generating automatic methods for a property whose name is **xxxx** are as follows:

1. A method called **getXxxx** which returns the value of the property is generated. The returned value will have the same type as **xxxx**.
2. If the type of **xxxx** is **boolean** then the generated method will be called **isXxxx** instead of...
If the property is not constant then a method for setting the property will also be generated. This will be called setXxxx, and take a single argument of the same type as xxxx. This assigns the argument to the property and returns no value.

If the property has an array type (for example, char[]), then it must only have a single dimension. Two further methods may then be generated, according to the rules:

1. A method called getXxxx which takes a single int as an argument and which returns an item from the property array is generated. The returned value will have the same type as xxxx, without the []. The integer argument is used to index into the array.

2. As before, if the result type of the method would be boolean then the name of the method will be isXxxx instead of getXxxx.

3. If the property is not constant then a method for setting an item in the property array will also be generated. This will be called setXxxx, and take two arguments: the first is an int that is used to select the item to be changed, and the second is an undimensioned argument of the same type as xxxx. It assigns the second argument to the item in the property array indexed by the first argument, and returns no value.

For example, for an indirect property declared thus:

```java
properties indirect
fred=foo.Bar[]
```

the four methods generated would be:

```java
method getFred returns foo.Bar[]; return fred
method getFred($1=int) returns foo.Bar; return fred[$1]
method setFred($2=foo.Bar[]); fred=$2
method setFred($3=int, $4=foo.Bar); fred[$3]=$4
```

Note that in all cases a method will only be generated if it would not exactly match a method explicitly coded in the current class.

**Explicit provision of access methods**

Often, for example when an indirect property has an on-screen representation, it is desirable to redraw the property when the property is changed (and in more complicated cases, there may be interactions between properties). These and other actions will require extra processing which will not be carried out by automatically generated methods. To add this processing the access methods will have to be coded explicitly. In the “Sandwich” example, we only need to supply the set methods, perhaps by adding the following to the example class above:

```java
method setSlices(col=Color)
    slices=col -- update the property
    this.repaint -- redraw the component

method setFilling(col=Color)
    filling=col
    this.repaint
```

If we add these two methods, they will no longer be added automatically (the two get methods will continue to be provided automatically, however). Further, since the names match possible access methods for properties that are declared to be indirect, the compiler will check the method declaration: the method signatures and return type (if any) must be correct, for example. Also, since the names of
access methods are case-sensitive (in a Java environment), you will be warned if a method appears to be intended to be an access method but the case of one or more letters is wrong.

Specifically, the checks carried out are as follows:

1. For methods whose names exactly match a potential access method for an indirect property (that is, start with `is`, `get`, or `set`, which is then followed by the name of an indirect property with the first character of the name uppercased):
   - The argument list for (signature of) the method must match one of those that could possibly be automatically generated for the property.
   - The returns type (if any) must match the expected returns type for that method.
   - If the returns type is simply `boolean`, then the method name must start with `is`. Conversely, if the method name starts with `is` then the returns type must be just `boolean`.
   - If the property is `constant` then the name of the method cannot start with `set`.
   - A warning is given if the method is not `public` (the default).

2. For methods whose names match a potential access method, as above, except in case:
   - A warning is given that the method in question may be intended to be an indirect property access method, but will not be recognized as such by builders.

These checks detect a wide variety of errors at compile time, hence speeding the development of classes that use indirect properties.
Appendix C – The netrexx.lang Package

This appendix documents the netrexx.lang package, which includes the classes used for creating string objects of type Rexx along with several classes that are often used while running NetRexx programs.

This appendix describes the public methods and properties of these classes, as implemented by the reference implementation. It does not include those “built-in” Methods for NetRexx strings (see page 155) in the Rexx class that form part of the NetRexx language, or those classes and methods that are internal “helper” components (which, for example, are used as repositories for rarely-executed code).

The classes in the netrexx.lang package are:

- The Exception classes (see page 180)
- Rexx (see page 181)
- RexxIO (helper class, for say and ask)
- RexxNode (helper class, for indexed strings)
- RexxOperators interface (see page 185)
- RexxParse (helper class, for parse)
- RexxSet (see page 186)
- RexxTrace (helper class, for trace)
- RexxUtil (helper class, for the Rexx class)
- RexxWords (helper class, for the Rexx class)
Exception classes

The classes provided for exceptions in the netrexx.lang package are all subclasses of java.lang.RuntimeException and all have the same content. Each has two constructors: one taking no argument and the other taking a string of type java.lang.String, which is used for additional detail describing the exception.

The Exceptions are signalled as follows.

**BadArgumentException** signalled when an argument to a method is incorrect.

**BadColumnException** signalled when a column number in a parsing template is not valid (for example, not a number).

**BadNumericException** signalled when a numeric digits instruction tries to set a value that is not a whole number, or is not positive, or is more than nine digits.

**DivideException** signalled when an error occurs during a division. This may be due to an attempt to divide by zero, or when the intermediate result of an integer divide or remainder operation is not valid.

**ExponentOverflowException** signalled when the exponent resulting from an operation would require more than nine digits.

**NoOtherwiseException** signalled when a select construct does not supply an otherwise clause and none of expressions on the when clauses resulted in '1'.

**NotCharacterException** signalled when a conversion from a string to a single character was attempted but the string was not exactly one character long.

**NotLogicException** signalled when a conversion from a string to a boolean was attempted but the string was neither the string '0' nor the string '1'.

Other exceptions, from the java.lang package, may also be signalled, for example 

NumberFormatException or NullPointerException.
The Rexx class

The class `netrexx.lang.Rexx` implements the NetRexx string class, and includes the “built-in” Methods for NetRexx strings (see page 155).

Described here are the platform-dependent methods as provided in the reference implementation: constructors (see page 181) for the class, the methods for arithmetic operations (see page 182), and miscellaneous methods (see page 184) intended for general use.

The class `netrexx.lang.Rexx` is serializable.

Rexx constructors

These constructors all create a string of type `netrexx.lang.Rexx`.

- **Rexx(arg=boolean)** Constructs a string which will have the value '1' if `arg` is 1 (`true`) or the value '0' if `arg` is 0 (`false`).

- **Rexx(arg=byte)** Constructs a string which is the decimal representation of the 8-bit signed binary integer `arg`. The string will contain only decimal digits, prefixed with a leading minus sign (hyphen) if `arg` is negative. A leading zero will be present only if `arg` is zero.

- **Rexx(arg=char)** Constructs a string of length 1 whose first and only character is a copy of `arg`.

- **Rexx(arg=char[])** Constructs a string by copying the characters of the character array `arg` in sequence. The length of the string is the number of elements in the character array (that is, `arg.length`).

- **Rexx(arg=int)** Constructs a string which is the decimal representation of the 32-bit signed binary integer `arg`. The string will contain only decimal digits, prefixed with a leading minus sign (hyphen) if `arg` is negative. A leading zero will be present only if `arg` is zero.

- **Rexx(arg=double)** Constructs a string which is the decimal representation of the 64-bit signed binary floating point number `arg`. (The precise format of the result may change and will be defined later.)

- **Rexx(arg=float)** Constructs a string which is the decimal representation of the 32-bit signed binary floating point number `arg`. (The precise format of the result may change and will be defined later.)

- **Rexx(arg=long)** Constructs a string which is the decimal representation of the 64-bit signed binary integer `arg`. The string will contain only decimal digits, prefixed with a leading minus sign (hyphen) if `arg` is negative. A leading zero will be present only if `arg` is zero.

- **Rexx(arg=Rexx)** Constructs a string which is copy of `arg`, which is of type `netrexx.lang.Rexx`. `arg` must not be null. Any sub-values (see page 73) are ignored (that is, they are not present in the object returned by the constructor).

- **Rexx(arg=short)** Constructs a string which is the decimal representation of the 16-bit signed binary integer `arg`. The string will contain only decimal digits, prefixed with a
leading minus sign (hyphen) if \( \text{arg} \) is negative. A leading zero will be present only if \( \text{arg} \) is zero.

**Rexx(arg=String)**  Constructs a NetRexx string by copying the characters of \( \text{arg} \), which is of type `java.lang.String`, in sequence. The length of the string is same as the length of \( \text{arg} \) (that is, \( \text{arg}.\text{length()} \)). \( \text{arg} \) must not be `null`.

**Rexx(arg=String[])**  Constructs a NetRexx string by concatenating the elements of the `java.lang.String` array \( \text{arg} \) together in sequence with a blank between each pair of elements. This may be used for converting the argument word array passed to the `main` method of a Java application into a single string. If the number of elements of \( \text{arg} \) is zero then an empty string (of length 0) is returned. Otherwise, the length of the string is the sum of the lengths of the elements of \( \text{arg} \), plus the number of elements of \( \text{arg} \), less one. \( \text{arg} \) must not be `null`.

### Rexx arithmetic methods

These methods implement the NetRexx arithmetic operators, as described in the section on *Numbers and arithmetic* (see page 140). Each corresponds to and implements a method in the `RexxOperators` interface class (see page 185).

Each of the methods here takes a `RexxSet` (see page 186) object as an argument. This argument provides the `numeric` settings for the operation; if `null` is provided for the argument then the default settings are used (`digits=9, form=scientific`).

For monadic operators, only the `RexxSet` argument is present; the operation acts upon the current object. For dyadic operators, the `RexxSet` argument and a `Rexx` argument are present; the operation acts with the current object being the left-hand operand and the second argument being the right-hand operand. For example, under default numeric settings, the expression:

\[
\text{award} + \text{extra}
\]

(where `award` and `extra` are references to objects of type `Rexx`) could be written as:

\[
\text{award}.\text{OpAdd}(\text{null}, \text{extra})
\]

which would return the result of adding `award` and `extra` under the default numeric settings.

**OpAdd(set=RexxSet, rhs=Rexx)**  Implements the NetRexx `+` (Add) operator, and returns the result as a string of type `Rexx`.

**OpAnd(set=RexxSet, rhs=Rexx)**  Implements the NetRexx `&` (And) operator, and returns a result (0 or 1) of type `boolean`.

**OpCc(set=RexxSet, rhs=Rexx)**  Implements the NetRexx `||` or `abuttal` (Concatenate without blank) operator, and returns the result as a string of type `Rexx`.

**OpCcblank(set=RexxSet, rhs=Rexx)**  Implements the NetRexx `blank` (Concatenate with blank) operator, and returns the result as a string of type `Rexx`.

**OpDiv(set=RexxSet, rhs=Rexx)**  Implements the NetRexx `/` (Divide) operator, and returns the result as a string of type `Rexx`.

**OpDivI(set=RexxSet, rhs=Rexx)**  Implements the NetRexx `%` (Integer divide) operator, and returns the
result as a string of type Rexx.

OpEq(set=RexxSet, rhs=Rexx) Implements the NetRexx = (Equal) operator, and returns a result (0 or 1) of type boolean.

OpEqS(set=RexxSet, rhs=Rexx) Implements the NetRexx == (Strictly equal) operator, and returns a result (0 or 1) of type boolean.

OpGt(set=RexxSet, rhs=Rexx) Implements the NetRexx > (Greater than) operator, and returns a result (0 or 1) of type boolean.

OpGtEq(set=RexxSet, rhs=Rexx) Implements the NetRexx >= (Greater than or equal) operator, and returns a result (0 or 1) of type boolean.

OpGtEqS(set=RexxSet, rhs=Rexx) Implements the NetRexx >>= (Strictly greater than or equal) operator, and returns a result (0 or 1) of type boolean.

OpGtS(set=RexxSet, rhs=Rexx) Implements the NetRexx >> (Strictly greater than) operator, and returns a result (0 or 1) of type boolean.

OpLt(set=RexxSet, rhs=Rexx) Implements the NetRexx < (Less than) operator, and returns a result (0 or 1) of type boolean.

OpLtEq(set=RexxSet, rhs=Rexx) Implements the NetRexx <= (Less than or equal) operator, and returns a result (0 or 1) of type boolean.

OpLtEqS(set=RexxSet, rhs=Rexx) Implements the NetRexx <<= (Strictly less than or equal) operator, and returns a result (0 or 1) of type boolean.

OpLtS(set=RexxSet, rhs=Rexx) Implements the NetRexx << (Strictly less than) operator, and returns a result (0 or 1) of type boolean.

OpMinus(set=RexxSet) Implements the NetRexx Prefix - (Minus) operator, and returns the result as a string of type Rexx.

OpMult(set=RexxSet, rhs=Rexx) Implements the NetRexx * (Multiply) operator, and returns the result as a string of type Rexx.

OpNot(set=RexxSet) Implements the NetRexx Prefix \ (Not) operator, and returns a result (0 or 1) of type boolean.

OpNotEq(set=RexxSet, rhs=Rexx) Implements the NetRexx \= (Not equal) operator, and returns a result (0 or 1) of type boolean.

OpNotEqS(set=RexxSet, rhs=Rexx) Implements the NetRexx \== (Strictly not equal) operator, and returns a result (0 or 1) of type boolean.

OpOr(set=RexxSet, rhs=Rexx) Implements the NetRexx | (Inclusive or) operator, and returns a result (0 or 1) of type boolean.

OpPlus(set=RexxSet) Implements the NetRexx Prefix + (Plus) operator, and returns the result as a string of type Rexx.

OpPow(set=RexxSet, rhs=Rexx) Implements the NetRexx ** (Power) operator, and returns the result as a string of type Rexx.

OpRem(set=RexxSet, rhs=Rexx) Implements the NetRexx // (Remainder) operator, and returns the result as a string of type Rexx.

OpSub(set=RexxSet, rhs=Rexx) Implements the NetRexx - (Subtract) operator, and returns the result as a string of type Rexx.
string of type Rexx.

Implements the NetRexx $\&\&$ (Exclusive or) operator, and returns a result (0 or 1) of type boolean.

### Rexx miscellaneous methods

These methods provide standard Java methods for the class, together with various conversions.

- `charAt(offset=int)`
  
  Returns the character from the string at `offset` (that is, if `offset` is 0 then the first character is returned, etc.). The character is returned as type `char`.
  
  If `offset` is negative, or is greater than or equal to the length of the string, an exception is signalled.

- `equals(item=Object)`
  
  Compares the string with the value of `item`, using a strict character-by-character comparison, and returns a result of type `boolean`.
  
  If `item` is `null` or is not an instance of one of the types Rexx, java.lang.String, or char[], then 0 is returned. Otherwise, `item` is converted to type Rexx and the OpEqS (see page 183) method (or equivalent) is used to compare the current string with the converted string, and its result is returned.

- `hashCode()`
  
  Returns a hashcode of type `int` for the string. This hashcode is suitable for use by the java.util.Hashtable class.

- `tobool()`
  
  Converts the string to type `boolean`. If the string is neither “0” nor “1” then a NotLogicException (see page 180) is signalled.

- `tobyte()`
  
  Converts the string to type `byte`. If the string is not a number, has a non-zero decimal part, or is out of the possible range for a byte (8-bit signed integer) result then a NumberFormatException is signalled.

- `tobool()`
  
  Converts the string to type char. If the string is not exactly one character in length then a NotCharacterException (see page 180) is signalled.

- `tobool()`
  
  Converts the string to type `char[]`. A character array object of the same length as the string is created, and the characters of the string are copied to the array in sequence. The character array is then returned.

- `todouble()`
  
  Converts the string to type `double`. If the string is not a number, or is out of the possible range for a double (64-bit signed floating point) result then a NumberFormatException is signalled.

- `tofloat()`
  
  Converts the string to type `float`. If the string is not a number, or is out of the possible range for a float (32-bit signed floating point) result then a NumberFormatException is signalled.

- `toint()`
  
  Converts the string to type `int`. If the string is not a number, has a non-zero decimal part, or is out of the possible range for an int (32-bit signed integer) result then a NumberFormatException is signalled.

- `tolong()`
  
  Converts the string to type `long`. If the string is not a number, has a non-zero decimal part, or is out of the possible range for a long (64-bit signed integer) result then a NumberFormatException is signalled.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toshort()</td>
<td>Converts the string to type short. If the string is not a number, has a non-zero decimal part, or is out of the possible range for a short (16-bit signed) result then a NumberFormatException is signalled.</td>
</tr>
<tr>
<td>toString()</td>
<td>Converts the string to type java.lang.String. A String object of the same length as the string is created, and the characters of the string are copied to the new string in sequence. The String is then returned.</td>
</tr>
</tbody>
</table>

**The RexxOperators interface class**

The RexxOperators interface class defines the signatures of the methods that implement the NetRexx (and Rexx) operators. These methods are described in the section Rexx arithmetic methods (see page 182).

In the future this interface may be used to allow the overloading of operators for objects of types other than Rexx. The current NetRexx language definition does not permit operator overloading.
**The RexxSet class**

The RexxSet class is used to provide the numeric settings for the methods described in the section Rexx arithmetic methods (see page 182). When provided, a RexxSet Object supplies the numeric settings for the operation; when null is provided then the default settings are used (digits=9, form=SCIENTIFIC).

### Public properties

These properties supply the numeric settings and certain values they may take. After construction, the digits and form values should only be changed by using the setDigits and setForm methods.

- **DEFAULT_DIGITS** A constant of type int that describes the default number of digits for a numeric operation (9).
- **DEFAULT_FORM** A constant of type byte that describes the default exponential format for a numeric operation (SCIENTIFIC).
- **digits** A value of type int that describes the numeric digits to be used for a numeric operation. The Rexx arithmetic methods (see page 182) use this value to determine the significance of results. digits must always be greater than zero.
- **ENGINEERING** A constant of type byte that signifies that engineering exponential formatting should be used for a numeric operation.
- **form** A value of type byte that describes the exponential format to be used for a numeric operation. The Rexx arithmetic methods (see page 182) use this value to determine the formatting of results that require an exponent. form must be either ENGINEERING or SCIENTIFIC.
- **SCIENTIFIC** A constant of type byte that signifies that scientific exponential formatting should be used for a numeric operation.

### Constructors

These constructors are used to set the initial values of a RexxSet object.

- **RexxSet()** Constructs a RexxSet object which has default digits and form properties.
- **RexxSet(newdigits=int)** Constructs a RexxSet object which has its digits property set to newdigits and its form property set to DEFAULT_DIGITS.
- **RexxSet(newdigits=int, newform=byte)** Constructs a RexxSet object which has its digits property set to newdigits and its form property set to newform.
- **RexxSet(arg=RexxSet)** Constructs a RexxSet object which is copy of arg, which is of type netrexx.lang.RexxSet. arg must not be null.
Methods

The RexxSet class has the following additional methods:

**formword()**

Returns a string of type netrexx.lang.Rexx that describes the form property. This will either be the string 'engineering' or the string 'scientific', corresponding to the form value ENGINEERING or SCIENTIFIC, respectively.

**setDigits(newdigits=Rexx)**

Sets the digits value for the RexxSet object, from newdigits, after rounding and checking as defined for the numeric instruction; newdigits must be a positive whole number with no more than nine digits. No value is returned.

**setForm(newformword=Rexx)**

Sets the form value for the RexxSet object, from newformword. This must equal either the string 'engineering' or the string 'scientific', corresponding to the form value ENGINEERING or SCIENTIFIC, respectively. No value is returned.
Index

A
ABBREV method 156
Abbreviations
  testing with ABBREV method 156
ABS method 156
Absolute
  column specification in parsing 136
  positional pattern 137
  value, finding using ABS method 156
ABSTRACT
  on CLASS instruction 79
  on METHOD instruction 99
Abstract classes 79
Abstract methods 79, 99
Abuttal concatenation operator 62, 64
Acknowledgements 15
Active constructs 90
Active constructs 89
Adaptability 13
ADAPTER
  on CLASS instruction 79
Adapter classes 79
Addition 62
  definition 143
Algebraic precedence 66
ALL
  TRACE setting 120
Alphabets
  checking with DATATYPE 159
Alphanumeric
  checking with DATATYPE 159
AND
  logical operator 64
ANSI standard
  arithmetic definition 141
  for REXX 9
Arbitrary precision arithmetic 140
Arguments
  of methods 53
  on METHOD instruction 98
  optional 99
  passing to methods 53
  provided by caller 98
  required 99
Arithmetic 140
  comparisons 146
  errors 148
  exceptions 148
  implementation independence 148
  NUMERIC settings 104
  operation rules 143
  operators 62, 140, 142
  overflow 148
  overview 22
  precision 141
  underflow 148
Array initializer
  in terms 48, 75
Arrays 74
  constructors 74
  in terms 52
  initializing 75
  overview 26
  references 74
ASCII
  coded character set 39
ASK special word 130
Assignment 68, 69
  binary 150
  instruction 68, 69
  of literals 150
  property initialization 124
Astonishment factor 13

B
B2X method 156
Background 7
Backslash character
  escape sequence 41
  in strings 41
not operator 64
BadArgumentException 180
BadColumnException 180
BadNumericException 180
BASIC, programming language 11
Binary
    arithmetic 149
    checking with DATATYPE 159
    conversion to hexadecimal 156
    operations 149
    see Conversion 159
    values 149
BINARY
    in OPTIONS instruction 106
    on CLASS instruction 80
    on METHOD instruction 101
Binary classes 80, 149
    assignment 150
    binary methods 101
    control variables 150
    LOOP instruction 150
    NUMERIC instruction 150
Binary constructors 150
Binary literals 150
Binary methods 101, 149
    assignment 150
    control variables 150
    LOOP instruction 150
    NUMERIC instruction 150
Binary numbers 59, 149
Binary numbers
    overview 33
Binary numeric symbol 42, 45
Binary operations
    dyadic 149
    monadic 150
    prefix 150
Bits
    binary operators 64
    checking with DATATYPE 159
Blank 40
    adjacent to operator character 43
    adjacent to special character 43
    as concatenation operator 62
    as type conversion operator 64
    operator 62, 64
    removal with SPACE method 166
    removal with STRIP method 166
Block comments 40
Body
    of a loop 91
    of classes 78
    of group 82
    of methods 98
    of select 116
Boolean operations 64
boolean type, value of 59
Bottom of program, reaching during execution 84
Bounded loop 92
    controlled 92
    over values 94
    simple 92
Brackets
    in array initializers 48, 75
    in array references 74
    in indexed references 48
    in indexed strings 73
    in terms 48
Built-in methods 155
Built-in methods
    see Method, built-in 155
BY phrase of LOOP instruction 91

C
C, programming language 8
C2D method 158
C2X method 158
Carriage return character
    escape sequence 41
Case
    insensitivity to 10
    of names 44
CASE
    on SELECT instruction 117
Casting
    to type 64
Casts
    see Conversion 58
CATCH
    on DO instruction 83
    on LOOP instruction 96
    on SELECT instruction 118
    use of 153
Caught exceptions 152
CENTER method 157
CENTRE method 157
CHANGESTR method 157
Changing strings
    using CHANGESTR 157
    using TRANSLATE 167
char
    as a string 59
Character 39
    appearance 39
    conversion to decimal 158
    conversion to hexadecimal 158
    converting to binary 150
rounding when numbers used 147
digits property 186
Dimension of arrays 46
of types 46
Dimensioned types 46
Displaying data
see SAY instruction 115
DivideException 180
Division definition 143
integer 140
DO group 82
naming of 82
DO instruction 82
LABEL 82
see grouping 82
Dollar sign
in symbols 42
Double-quote
escape sequence 41
string delimiter 41
Dummy instruction, NOP 103
Duplicate methods 102
Dyadic operators 61

E
E-notation 65, 147
E-notation
definition 146
in symbols 42
EBCDIC
coded character set 39
ELSE keyword
see IF instruction 85
Empty reference, null 131
Encodings
binary 150
of characters 39
Encodings, of characters 39
END clause
see DO instruction 82
see LOOP instruction 91
see SELECT instruction 116
specifying control variable 94
End condition of a LOOP loop 92
End-of-file character 40
Engineering notation 104, 147
ENGINEERING property 186
ENGINEERING value for NUMERIC FORM 104
Environment, independency from 12
EOF character 40

Equality
of objects 63
testing of 62
equals method 184
Equals sign
see = equals sign 69
Error detection, localized 12
Errors during arithmetic 148
Escape sequences in strings 41
Euro character 42
in symbols 42
Evaluation
of expressions 61
of terms 49
Even/odd rounding 142
Example
applet 34
arrays 26
Hello World 124
indexed strings 25
of constructors 57
of exception handling 153
of two classes 125
program 18, 19, 25, 27, 29, 31-33, 35, 77, 172
trace 31, 32
Exception
BadArgumentException 180
BadColumnException 180
BadNumericException 180
DivideException 180
ExponentOverflowException 180
NoOtherwiseException 180
NotCharacterException 180
NotLogicException 180
NullPointerException 180
NumberFormatException 180
Exceptions 152
after CATCH clause 153
after FINALLY clause 153
checked 153
during arithmetic 148
during conversions 59
listed on METHOD instruction 101
overview 21, 35
raising 119
signalling 119
throwing 119
Exclusive OR
logical operator 64
EXISTS method 161
EXIT instruction 84
Experimental feature 175
EXPLICIT option 107
Exponential notation 65, 104, 140, 147
Exponential notation
  definition 146
  in symbols 42
Exponentiation 62
  definition 144
ExponentOverflowException 180
Expressions
  evaluation 61
  examples 66
  overview 19
  results of 61
Extending classes
  overview 29
EXTENDS
  on CLASS instruction 80
Extra digits
  in numbers 141
  in numeric symbols 42, 43
  in symbols 42
Extra letters, in symbols 42
Extracting
  a sub-string 166
  words from a string 166

F
False value 64
FINAL
  on CLASS instruction 79
  on METHOD instruction 99
Final classes 79
Final methods 100
FINALLY
  on DO instruction 83
  on LOOP instruction 96
  on SELECT instruction 118
  reached by LEAVE 90
  use of 153
Finding a mismatch using COMPARE 157
Finding a string in another string 163, 165
Fixed size, of arrays 74
Floating-point numbers, binary 149
Flow control
  abnormal, with SIGNAL 119
  with DO construct 82
  with IF construct 85
  with LOOP construct 91
  with SELECT construct 116
FOR
  phrase of LOOP instruction 91
  repetitor on LOOP instruction 91
FOREVER
  loops 92
  repetitor on LOOP instruction 91
FORM
  option of NUMERIC instruction 104, 147
    special word 130
Form feed character 40
form property 186
FORMAT
  method 161
  option 107
Formatting
  numbers for display 161
  numbers with TRUNC 167
  of output during tracing 122
  text centering 157
  text left justification 163
  text right justification 165
  text spacing 166
formword() method 187
Full name
  of classes 127
Fully-qualified name, of classes 110
Functions
  numeric arguments of 147
  return from 114
  see Methods, static 100
  used by classes 80
G
Glyphs 39
Group, DO 82
Guard digit in arithmetic 142
H
hashCode method 184
Hexadecimal
  checking with DATATYPE 159
  conversion to binary 169
  conversion to character 169
  conversion to decimal 170
  digits in escapes 42
  escape sequence 41
  see Conversion 159
Hexadecimal numeric symbol 42, 45
Hyphen
  as continuation character 44
I
IF instruction 85
IMPLEMENTS
  on CLASS instruction 80
Implied semicolons 44
IMPORT instruction 84
Imports
  automatic 88

Index
explicit 87
Inclusive OR operator
    see OR logical operator 64
Indefinite loops 91, 92
Indention during tracing 122
Index strings
    for sub-values 73
    testing for 161
Indexed references
    arrays 74
    in terms 48
    indexed strings 73
Indexed strings 73
    copying 158
    example 25
    merging 158
    overview 25
    testing for 161
INDIRECT
    on PROPERTIES instruction 176
Indirect properties 176
Inequality, testing of 62
Infinite loops 91
Influence
    of C 8
    of Java 8
    of Rexx 7
INHERITABLE
    on METHOD instruction 99
    on PROPERTIES instruction 112
Initializing arrays 75
Inner classes
    see Minor classes 127
INSERT method 163
Inserting a string into another 163
Instance, of a class 56
Instructions 77
    assignment 68, 69
    CLASS 78
    DO 82
    EXIT 84
    IF 85
    IMPORT 87
    ITERATE 89
    keyword 68, 77
    LEAVE 90
    LOOP 91
    METHOD 98, 101
    method call 68
    NOP 103
    NUMERIC 104
    OPTIONS 106
    PACKAGE 110
    PARSE 111
PROPERTIES 112, 176
RETURN 114
SAY 115
SELECT 116
SIGNAL 119
TRACE 120
Integer arithmetic 140
Integer division 62, 140
    definition 145
Integers, binary 149
INTERFACE
    on CLASS instruction 79
Interface classes 79
    properties in 113
Interfaces
    implemented by classes 80
Internal functions
    return from 114
Interpreter options 106
Introduction 7
ITERATE instruction 89
    see LOOP construct 89
    use of variable on 89
Java
    features of 8
    in reference implementation 37
    influence of 8
    programming language 8
JAVA option 107
JavaBean properties 175
Keyword instructions 68, 77
Keyword safety 7
Keywords 68
    mixed case 77
LABEL
    on DO instruction 82
    on LOOP instruction 95
    on SELECT instruction 95
Language concepts 10
Language processor options 106
LASTPOS method 163
Leading blanks
    removal with STRIP method 166
Leading zeros
    adding with the RIGHT method 165
    removal with STRIP method 166
LEAVE instruction 90
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>see DO construct</td>
<td>90</td>
</tr>
<tr>
<td>use of variable on</td>
<td>90</td>
</tr>
<tr>
<td>LEFT method</td>
<td>163</td>
</tr>
<tr>
<td>Legibility, perceived</td>
<td>10</td>
</tr>
<tr>
<td>Length</td>
<td></td>
</tr>
<tr>
<td>of arrays</td>
<td>52</td>
</tr>
<tr>
<td>of comments</td>
<td>40</td>
</tr>
<tr>
<td>LENGTH method</td>
<td>163</td>
</tr>
<tr>
<td>special word</td>
<td>52, 130</td>
</tr>
<tr>
<td>Letters</td>
<td></td>
</tr>
<tr>
<td>checking with DATATYPE</td>
<td>159</td>
</tr>
<tr>
<td>Limits of size</td>
<td>14</td>
</tr>
<tr>
<td>Line comments</td>
<td>40</td>
</tr>
<tr>
<td>Line ends, effect of</td>
<td>44</td>
</tr>
<tr>
<td>Line feed character</td>
<td></td>
</tr>
<tr>
<td>escape sequence</td>
<td>41</td>
</tr>
<tr>
<td>Line numbers, in tracing</td>
<td>122</td>
</tr>
<tr>
<td>Line, displaying</td>
<td>115</td>
</tr>
<tr>
<td>Literal patterns</td>
<td>135</td>
</tr>
<tr>
<td>Literal strings</td>
<td>41</td>
</tr>
<tr>
<td>in terms</td>
<td>48</td>
</tr>
<tr>
<td>see Strings</td>
<td>41</td>
</tr>
<tr>
<td>Literals, binary</td>
<td>150</td>
</tr>
<tr>
<td>Local variables</td>
<td>70</td>
</tr>
<tr>
<td>Locating</td>
<td></td>
</tr>
<tr>
<td>a string in another string</td>
<td>163, 165</td>
</tr>
<tr>
<td>a word or phrase in a string</td>
<td>169</td>
</tr>
<tr>
<td>Logical operations</td>
<td>64</td>
</tr>
<tr>
<td>LOGO option</td>
<td>107</td>
</tr>
<tr>
<td>LOOP instruction</td>
<td>91</td>
</tr>
<tr>
<td>see loops</td>
<td>91</td>
</tr>
<tr>
<td>Loops</td>
<td></td>
</tr>
<tr>
<td>active</td>
<td>89, 90</td>
</tr>
<tr>
<td>execution model</td>
<td>96</td>
</tr>
<tr>
<td>in binary classes and methods</td>
<td>150</td>
</tr>
<tr>
<td>label</td>
<td>95</td>
</tr>
<tr>
<td>modification of</td>
<td>89</td>
</tr>
<tr>
<td>naming of</td>
<td>95</td>
</tr>
<tr>
<td>repetitive</td>
<td>91, 92</td>
</tr>
<tr>
<td>see LOOP instruction</td>
<td>91</td>
</tr>
<tr>
<td>termination of</td>
<td>90</td>
</tr>
<tr>
<td>LOWER method</td>
<td>163</td>
</tr>
<tr>
<td>Lowercase</td>
<td></td>
</tr>
<tr>
<td>checking with DATATYPE</td>
<td>159</td>
</tr>
<tr>
<td>names</td>
<td>44</td>
</tr>
<tr>
<td>Lowercasing strings</td>
<td>163</td>
</tr>
</tbody>
</table>

**M**

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mantissa of exponential numbers</td>
<td>146</td>
</tr>
<tr>
<td>Matching methods</td>
<td>54</td>
</tr>
<tr>
<td>Mathematical method</td>
<td></td>
</tr>
<tr>
<td>ABS</td>
<td>156</td>
</tr>
<tr>
<td>DATATYPE options</td>
<td>159</td>
</tr>
<tr>
<td>FORMAT</td>
<td>161</td>
</tr>
<tr>
<td>MAX</td>
<td>164</td>
</tr>
<tr>
<td>MIN</td>
<td>164</td>
</tr>
<tr>
<td>SIGN</td>
<td>165</td>
</tr>
<tr>
<td>MAX method</td>
<td>164</td>
</tr>
<tr>
<td>Member classes</td>
<td></td>
</tr>
<tr>
<td>see Dependent classes</td>
<td>128</td>
</tr>
<tr>
<td>Merging indexed variables</td>
<td>158</td>
</tr>
<tr>
<td>Method</td>
<td>46</td>
</tr>
<tr>
<td>argument variables</td>
<td>70</td>
</tr>
<tr>
<td>body of</td>
<td>98</td>
</tr>
<tr>
<td>calls in terms</td>
<td>48</td>
</tr>
<tr>
<td>charAt</td>
<td>184</td>
</tr>
<tr>
<td>definition</td>
<td>124</td>
</tr>
<tr>
<td>equals</td>
<td>184</td>
</tr>
<tr>
<td>formword()</td>
<td>187</td>
</tr>
<tr>
<td>hashCode</td>
<td>184</td>
</tr>
<tr>
<td>names, case of</td>
<td>44</td>
</tr>
<tr>
<td>NotEq</td>
<td>183</td>
</tr>
<tr>
<td>NotEq$</td>
<td>183</td>
</tr>
<tr>
<td>OpAdd</td>
<td>182</td>
</tr>
<tr>
<td>OpAnd</td>
<td>182</td>
</tr>
<tr>
<td>OpCc</td>
<td>182</td>
</tr>
<tr>
<td>OpCcblank</td>
<td>182</td>
</tr>
<tr>
<td>OpDiv</td>
<td>182</td>
</tr>
<tr>
<td>OpDivI</td>
<td>182</td>
</tr>
<tr>
<td>OpEq</td>
<td>183</td>
</tr>
<tr>
<td>OpEqS</td>
<td>183</td>
</tr>
<tr>
<td>OpGt</td>
<td>183</td>
</tr>
<tr>
<td>OpGtEq</td>
<td>183</td>
</tr>
<tr>
<td>OpGtEqS</td>
<td>183</td>
</tr>
<tr>
<td>OpGtS</td>
<td>183</td>
</tr>
<tr>
<td>OpLt</td>
<td>183</td>
</tr>
<tr>
<td>OpLtEq</td>
<td>183</td>
</tr>
<tr>
<td>OpLtEqS</td>
<td>183</td>
</tr>
<tr>
<td>OpLtS</td>
<td>183</td>
</tr>
<tr>
<td>OpMinus</td>
<td>183</td>
</tr>
<tr>
<td>OpMult</td>
<td>183</td>
</tr>
<tr>
<td>OpNot</td>
<td>183</td>
</tr>
<tr>
<td>OpOr</td>
<td>183</td>
</tr>
<tr>
<td>OpPlus</td>
<td>183</td>
</tr>
<tr>
<td>OpPow</td>
<td>183</td>
</tr>
<tr>
<td>OpRem</td>
<td>183</td>
</tr>
<tr>
<td>OpSub</td>
<td>184</td>
</tr>
<tr>
<td>OpXor</td>
<td>184</td>
</tr>
<tr>
<td>setDigits(Rexx)</td>
<td>187</td>
</tr>
<tr>
<td>setForm(Rexx)</td>
<td>187</td>
</tr>
<tr>
<td>short name of</td>
<td>98</td>
</tr>
<tr>
<td>starting</td>
<td>98</td>
</tr>
<tr>
<td>toboolean</td>
<td>184</td>
</tr>
<tr>
<td>tobyte</td>
<td>184</td>
</tr>
<tr>
<td>tochar</td>
<td>184</td>
</tr>
<tr>
<td>todouble</td>
<td>184</td>
</tr>
</tbody>
</table>
see Conversion 159
truncating 167
use of by NetRexx 147

Numeric
part of a number 141, 146
NUMERIC
DIGITS 141
FORM 147
in binary classes and methods 150
instruction 104
Numeric symbols 42, 48
Numeric symbols
binary 45
hexadecimal 45

O
Object Rexx, programming language 8
Object-oriented programming concepts 10
Objectives of the NetRexx language 7
Objects
comparing 63
constructing 56
equality 63
overview 27
OFF
TRACE setting 120
OpAdd method 182
OpAnd method 182
OpCc method 182
OpCcbblank method 182
OpDiv method 182
OpDivI method 182
OpEq method 183
OpEqS method 183
Operators 61
arithmetic 62, 140, 142
blank 62, 64
characters used for 43
comparative 62, 146
composition of 61
concatenation 62
logical 64
precedence (priorities) of 66
type 64
OpGt method 183
OpGtEq method 183
OpGtEqS method 183
OpGtS method 183
OpLt method 183
OpLtEq method 183
OpLtEqS method 183
OpLtS method 183
OpMinus method 183
OpMult method 183
OpNot method 183
OpOr method 183
OpPlus method 183
OpPow method 183
OpRem method 183
OpSub method 184
Option words 106
Optional arguments 99
Options
on command line 109
OPTIONS
instruction 106
OpXor method 184
OR
logical exclusive 64
logical inclusive 64
OTHERWISE clause
see SELECT instruction 116
Over loops 94
OVER repetitor on LOOP instruction 91
Overflow, arithmetic 148
OVERLAY method 165
Overlaying a string onto another 165
Overloaded methods 102
OVERRIDING methods 55
Overview
Arithmetic 22
Arrays 26
binary types 33
control instructions 21
conversions 33
exceptions 35
expressions 19
extending classes 29
indexed strings 25
NetRexx 17
objects 27
parsing 24
programs 18
strings 23
tracing 31
variables 19

P
Package 46, 110
name of 87, 110
netrexx.lang 179
PACKAGE instruction 110
Packing a string
with B2X 156
with X2C 169
Parent
of dependent object 128
PARENT
  special word 129
Parent class 127
Parent object 128
Parentheses
  adjacent to blanks 43
  in expressions 61, 66
  in method calls 48, 53
  in parsing templates 138
  in terms 48
  omitting from method calls 48, 49
PARSE
  instruction 111
  parsing rules 133
Parsing 133
  absolute columns 137
  definition 134
  general rules 133, 134
  introduction 133
  literal patterns 135
  overview 24
  patterns 135
  positional patterns 136
  selecting words 135
  variable patterns 138
Parsing templates 133
Parsing templates
  in PARSE instruction 111
Patterns
  in parsing 135
Perceived legibility 10
Period
  as placeholder in parsing 136
  in numbers 141
  in terms 48
Philosophy of NetRexx 7, 10
POS position method 165
Positional patterns 136
Power operator 62
  definition 144
Powers of ten in numbers 65, 146
Precedence of operators 66
Precision
  arbitrary 10, 140
  of arithmetic 141
Prefix operators 61
  arithmetic 143
- 62
  with types 64
\ 64
  with types 64
+ 62
  with types 64
with types 64
Primitive types 46, 149
Primitive types
  conversions 58
Priorities of operators 66
PRIVATE
  on CLASS instruction 78
  on METHOD instruction 99
  on PROPERTIES instruction 112
Program
  filename of 131
  prolog 124
  structure 124
Programmer's model of LOOP 96
Programming style 10
Programs 124
  examples 25, 27, 29, 172
  overview 18
  structure 124
Prolog, of a program 124
Properties 46, 70, 112
  case of names 44
  constant 113
  deprecated 113
  for JavaBeans 175
  in dependent classes 129
  in interface classes 113
  in minor classes 129
  indirect 176
  inheritable 112
  initialization 124
  modifiers 113
  naming 112
  private 112
  public 112
  shared 112
  static 113
  transient 113
  unused 113
  visibility 112
  volatile 113
PROPERTIES instruction 112, 176
Property
  DEFAULT_DIGITS 186
  DEFAULT_FORM 186
digits 186
ENGINEERING 186
form 186
SCIENTIFIC 186
PROTECT
  on DO instruction 82
  on LOOP instruction 96
  on METHOD instruction 100
on SELECT instruction 117
Protected methods 100
PUBLIC
  on CLASS instruction 78
  on METHOD instruction 99
  on PROPERTIES instruction 112
Punctuation, optional 10
Pure numbers 146
Pure numbers
  see Numbers 140

Q
qtime example program 172
Qualified name, of classes 110
Qualified types 46
Quotes in strings 41

R
Raising exceptions 119
Raising exceptions
  see SIGNAL 119
Re-ordering characters
  with TRANSLATE method 167
Readability, of programs 10
Real numbers, binary 149
Reality, dealing with 13
Reference implementation 37
References
  in terms 48
  null 131
  to arrays 74
  to current object 131
  to indexed strings 73
  to methods 53
Relative column specification in parsing 137
Relative positional pattern 137
Reliability, of a language 13
Remainder operator 62, 140
definition 145
Repeating a string with COPIES 157
Repetitive loops 92
Repetitor phrase 92
REPLACE option 107
Replacing strings
  using CHANGESTR 157
  using TRANSLATE 167
Required arguments 99
Residue
  see Remainder operator 145
Resolution of methods 54
Results
  of methods 101
  returned by RETURN 114

size of 61
RESULTS
  TRACE setting 121
Return character
  escape sequence 41
Return code, setting on exit 84
RETURN instruction 114
Return string, setting on exit 84
RETURNS
  on METHOD instruction 101
REVERSE method 165
Rexx
  arithmetic 140
class
    conversions 58
    methods of 155
NetRexx strings 46
    use by PARSE 111
    features of 7
    influence of 7
Rexx(boolean) constructor 181
Rexx(byte) constructor 181
Rexx(char) constructor 181
Rexx(char[]) constructor 181
Rexx(double) constructor 181
Rexx(float) constructor 181
Rexx(int) constructor 181
Rexx(long) constructor 181
Rexx(Rexx) constructor 181
Rexx(String) constructor 182
Rexx(String[]) constructor 182
RexxSet() constructor 186
RexxSet(int,byte) constructor 186
RexxSet(int) constructor 186
RexxSet(RexxSet) constructor 186
RIGHT method 165
Robustness 13
Rounding 140
definition 142
  when numbers used 147
Routines
  see Methods 53
Running off the end of a program 84

S
Sample programs
  see Examples 172
SAVELOG option 107
SAY
  instruction 115
Scientific notation 104, 147
SCIENTIFIC property 186
SCIENTIFIC value for NUMERIC FORM 104
Search order
   for methods 54
   for term evaluation 50
Searching a string for a word or phrase 165, 169
Select
   label 117
   naming of 117
SELECT instruction 116
Semicolons 40
   can be omitted 38
   implied 44
SEQUENCE method 165
setDigits(Rexx) method 187
setForm(Rexx) method 187
SHARED
   on CLASS instruction 78
   on METHOD instruction 99
   on PROPERTIES instruction 112
Short name
   of classes 78, 127
   of methods 98
SIGN method 165
SIGNAL instruction 119
Signals 152
SIGNALS
   on METHOD instruction 101
Signature
   see Type 46
Significand of exponential numbers 146
Significant digits, in arithmetic 141
Signs in parsing templates 136
Simple DO group 82
Simple number 42
   see Numbers 140
Simple repetitor phrase 92
Simple terms 48
Single-quote
   escape sequence 41
   string delimiter 41
Size
   of language 14
   see Length 14
SOURCE special word 131
SOURCEDIR option 107
SOURCENAME
   special word 131
SPACE method 166
Special characters 43
Special characters
   used for operators 43
Special methods 132
   super 128, 132
   this 132
Special words 130
   ask 130
   class 130
   digits 130
   form 130
   length 130
   null 131
   parent 129
   source 131
   sourceline 131
   super 131
   this 129, 131
   trace 132
   version 132
Square brackets
   in array initializers 48, 75
   in indexed references 48
Standard classes 79
Standard methods 99
STATIC
   on METHOD instruction 99
   on PROPERTIES instruction 113
Static methods 100
Static methods
   used by classes 80
   Static variable typing 70
   stderr, used by TRACE 123
   stdin, reading with ASK 130
   stdout, writing to with SAY 115
Strict comparative operators 62
STRICTARGS option 107
STRICTASSIGN option 107
STRICTCASE option 108
STRICTIMPORT option 108
STRICTPROPS option 108
STRICTSIGNAL option 108
Strings 41
   as literal constants 41
   comparison of 62
   concatenation of 62
   escapes in 41
   in terms 48
   indexed 73
   length of 163
   lowercasing 163
   moving with TRANSULATE method 167
   null 41
   overview 23
   quotes in 41
   sub-values of 73
   types of 59
   uppercasing 168
   verifying contents of 168
STRIP method 166
see Case translation 167
with TRANSLATE method 167
Trapping of exceptions 119
Trapping of exceptions
see SIGNAL 119
True value 64
TRUNC method 167
Truncating numbers 167
Types 46
  checking instances of 64
  checking with DATATYPE 159
  concatenation of 64
  conversions 58
  declaring 70
  dimensioned 46
  of terms 61
  of values 61
  operations on 64
  primitive 46, 149
  qualified 46
  simplification 58
Typing (printing) data
  see SAY instruction 115

U
Underflow, arithmetic 148
Underscore
  in symbols 42
Unicode
  coded character set 39
  escape sequence 41
  UTF-8 encoding 108
Unpacking a string
  with C2X 158
  with X2B 169
UNTIL phrase of LOOP instruction 91
unused
  on PROPERTIES instruction 113
UPPER method 168
Uppercase
  checking with DATATYPE 159
  names 44
Uppercasing strings 168
USES
  on CLASS instruction 80
UTF-8 encoding 108
UTF8 option 108
Utility methods 155

V
Variable reference
  in parsing template 138
Variables 69
  controlling loops 92
  in parsing patterns 138
  indexed 73
  local 70
  method arguments 70
  names of 69
  overview 19
  parsing of 111
  properties 70
  scope of 70
  setting new value 69
  static typing of 70
  subscripts 73
  type of 69
  valid names 69
  visibility 70
VERBOSE option 109
VERBOSEn option 109
VERIFY method 168
VERSION special word 132
Visibility
  of classes 78
  of methods 99
  of properties 112
VOLATILE
  on PROPERTIES instruction 113

W
Well-known conversions 58
WHEN clause
  see SELECT instruction 116
WHILE phrase of LOOP instruction 91
White space 40
Whole numbers 65
  checking with DATATYPE 159
  definition 147
WORD method 168
Word processing
  see Formatting 155
  see Words 155
WORDINDEX method 168
WORDLENGTH method 169
WORDPOS method 169
Words
  counting, using WORDS 169
  deleting from a string 160
  extracting from a string 166, 168
  finding in a string 169
  finding length of 169
  in parsing 135
  locating in a string 168
  special
    ask 130
digits 130
form 130
length 130
null 131
source 131
super 131
this 131
trace 132
version 132

WORDS method 169
Words, special
class 130
sourceline 131

X
X2B method 169
X2C method 169
X2D method 170
XOR, logical operator 64

Z
Zero character
escape sequence 41
Zeros
adding on the left 165
padding 165
removal with STRIP method 166

_ underscore
in symbols 42

-
- continuation character 44
- minus sign
in parsing template 137
subtraction operator 62, 143
-- line comment delimiter 40

.
.(period)
as placeholder in parsing 136
in numbers 141
in terms 48

*
* multiplication operator 62, 143
*.* tracing flag 122
** power operator 62, 144
*/ block comment delimiter 40
*==* tracing flag 122

/ division operator 62, 143
/* block comment delimiter 40
// remainder operator 62, 145
\backslash
\escape character 41
not operator 64
\invalid sequence 43
< not less than operator 63
<< strictly not less than operator 63
<= not equal operator 63
=== strictly not equal operator 63
> not greater than operator 63
>=> strictly not greater than operator 63

&
& and operator 64
&& exclusive or operator 64

%
% integer division operator 62, 145

+
+ plus sign
addition operator 62, 143
in parsing template 137
++ invalid sequence 43
+++ tracing flag 122

<
< less than operator 63
<< strictly less than operator 63
<= strictly less than or equal operator 63
<= less than or equal operator 63
on types 65
<> less than or greater than operator 63

=
= equals sign
assignment indicator 69
equal operator 63
in LOOP instruction 91
in parsing template 137
=== strictly equal operator 63

>
greater than operator 63
>>< greater than or less than operator 63
>= greater than or equal operator 63
on types 65
strictly greater than operator 63
strictly greater than or equal operator 63
tracing flag 122
or operator 64
concatenation operator 62, 64
$ dollar sign
in symbols 42