# NetRexx Language Reference

Mike Cowlishaw and RexxLA

Version 3.02 of June 21, 2013

THE REXX LANGUAGE ASSOCIATION
NetRexx Programming Series
ISBN 978-90-819090-1-3

## Publication Data

The responsible publisher of this edition is identified as *IBizz IT Services and Consultancy*, Amsteldijk 14, 1074 HR Amsterdam, a registered company governed by the laws of the Kingdom of The Netherlands.

This edition is registered under ISBN 978-90-819090-1-3

ISBN 978-90-819090-1-3

9 789081 909013 >

# Contents

**The NetRexx Programming Series** **i**

**Typographical conventions** **iii**

**1 Introduction** **1**
1.1 Language Objectives 1
1.2 Language Concepts 3
1.3 Acknowledgements 8

**Index** **9**

# The NetRexx Programming Series

This book is part of a library, the *NetRexx Programming Series*, documenting the NetRexx programming language and its use and applications. This section lists the other publications in this series, and their roles. These books can be ordered in convenient hardcopy and electronic formats from the Rexx Language Association.

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

# Typographical conventions

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

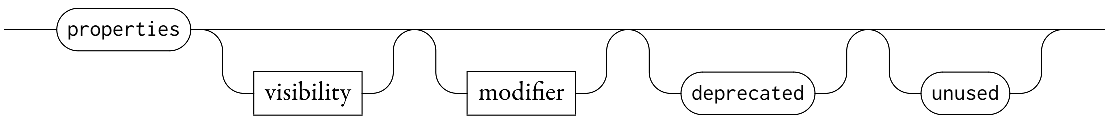
- Body text is in this font
- Examples of language statements are in a **bold** type
- Variables or strings as mentioned in source code, or things that appear on the console, are in a `typewriter` type
- Items that are introduced, or emphasized, are in an *italic* type
- Included program fragments are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

Listing 1: Example Listing

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

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

*Properties*



1

# 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 open source reference implementation (version 3 and later) 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.

## 1.1 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.

### 1.1.1 Features of Rexx

The Rexx programming language[1] 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

[1] Cowlishaw, M. F., **The REXX Language** (second edition), ISBN 0-13-780651-5, Prentice-Hall, 1990.

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 stringhandling
- 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].

### 1.1.2 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

[2]Kernighan, B. W., and Ritchie, D. M., **The C Programming Language** (second edition), ISBN 0-13-110362-8, Prentice- Hall, 1988.

[3]Gosling, J. A., *et al.* **The Java Language Specification**, ISBN 0-201-63451-1, Addison-Wesley, 1996.

[4]Nash, S. C., **Object-Oriented REXX** in Goldberg, G, and Smith, P. H. III, **The Rexx Handbook**, pp115-125, ISBN 0-07-023682-8, McGraw-Hill, Inc., New York, 1992.

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.

### 1.1.3 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.

## 1.2 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 re-

[5] See Cowlishaw, M. F., **The Early History of REXX**, IEEE Annals of the History of Computing, ISSN 1058-6180, Vol 16, No. 4, Winter 1994, pp15-24, and Cowlishaw, M. F., **The Future of Rexx**, Proceedings of Winter 1993 Meeting/SHARE 80, Volume II, p.2709, SHARE Inc., Chicago, 1993.

[6] See **American National Standard for Information Technology – Programming Language REXX**, X3.274-1996, American National Standards Institute, New York, 1996.

mainder of this section is therefore a summary of the principal concepts of NetRexx.

### 1.2.1 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.

### 1.2.2 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[7] 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."[8]

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.

### 1.2.3 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

```
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

[7] Kemeny, J. G. and Kurtz, T. E., **BASIC programming**, John Wiley & Sons Inc., New York, 1967.

[8] Kurtz, T. E., **BASIC** in Wexelblat, R. L. (Ed), **History of Programming Languages**, ISBN 0-12-745040-8, Academic Press, New York 1981.

by the same operators, at the same priority, making for a very natural and consistent syntax for changing the types of results. For example,

```
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`.

### 1.2.4 Nothing to declare

Consistent with the philosophy of simplicity, NetRexx does not require that variables within methods be declared before use. Only the *properties*[9] 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.

### 1.2.5 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.

### 1.2.6 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 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.

### 1.2.7 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

[9] Class variables and instance variables.

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.

### 1.2.8 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.

### 1.2.9 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.

### 1.2.10 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.

## 1.3 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 Lafaix, 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. IBM has also donated the NetRexx implementation to the Rexx Language Association, with special thanks due to Matthew Emmons for piloting NetRexx through the convoluted legal and other processes, and to René Jansen for massaging the NetRexx reference implementation into shape for its Open Source release.

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 uses a set of Rexx programs to translate that same GML markup in to LaTeX.

*Mike Cowlishaw, 1997 and 2009*

# Index

say, iii

ISBN 978-90-819090-1-3

9 789081 909013 >