NetRexx Language Reference

Mike Cowlishaw and RexxLA

Version 3.02 of June 24, 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

Contents

The NetRexx Programming Series

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

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

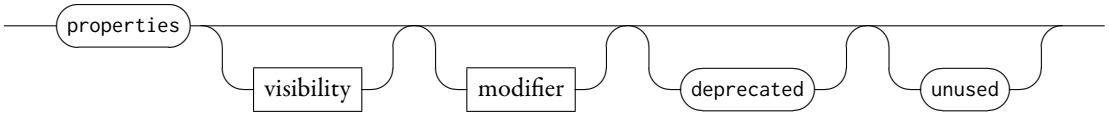
- Body text is in this font
- Examples of language statements are in a **bold** type
- Variables or strings as mentioned in source code, or things that appear on the console, are in a `typewriter` type
- Items that are introduced, or emphasized, are in an *italic* type
- Included program fragments are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

Listing 1: Example Listing

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

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

Properties



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 to LaTeX.

Mike Cowlishaw, 1997 and 2009

2

A Quick Tour of NetRexx

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

This is not a complete tutorial, though – think of it more as a *taster*; it covers the main points of the language and shows some examples you can try or modify.

2.1 NetRexx programs

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

Listing 2.1: Toast

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

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

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

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

```
java toast
```

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

```
netrexxc toast  run
```

Of course, NetRexx can do more than just display a character string. Although the language has a simple syntax, and has a small number of instruction types, it is powerful; the reference implementation of the language allows full access to the rapidly growing collection of Java programs known as class libraries, and allows new class libraries to be written in NetRexx. The rest of this overview introduces most of the features of NetRexx. Since the economy, power, and clarity of expression in NetRexx is best appreciated with use, you are urged to try using the language yourself.

2.2 Expressions and variables

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

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

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

would all result in '5'.

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

Listing 2.2: Assignment

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

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

Listing 2.3: Greet

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

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

```
Hello Fred!
```

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

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

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

Listing 2.4: Continuation

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

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

Listing 2.5: Multiple Instructions

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

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

2.3 Control instructions

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

Listing 2.6: Conditional

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

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

Listing 2.7: select - when - otherwise

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

do... end for grouping:

Listing 2.8: do - end

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

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

Listing 2.9: loop - end

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

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

Loop execution may be modified by **leave** and **iterate** instructions that significantly reduce the complexity of many programs. The **select**, **do**, and **loop** constructs also have the ability to "catch" exceptions (see 2.13 on page 22.) 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.

2.4 NetRexx arithmetic

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

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

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

Listing 2.10: Digits

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

which would display

```
0.14285714285714285714285714285714285714285714285714
```

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

```
options binary
```

at the top of a NetRexx program. The language processor will then use binary arithmetic (see page 21) instead of NetRexx decimal arithmetic for calculations, if it can, throughout the program.

2.5 Doing things with strings

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

Listing 2.11: Strings

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

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

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

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

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

Listing 2.12: Rexx: Nested

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

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

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

- `changestr` (change all occurrences of a substring to another)
- `copies` (make multiple copies of a string)
- `lastpos` (find rightmost occurrence)
- `left` and `right` (return leftmost/rightmost character(s))
- `pos` and `wordpos` (find the position of string or a word in a string)
- `reverse` (swap end-to-end)
- `space` (pad between words with fixed spacing)
- `strip` (remove leading and/or trailing white space)
- `verify` (check the contents of a string for selected characters)
- `word`, `wordindex`, `wordlength`, and `words` (work with words).

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

2.6 Parsing strings

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

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

2.6.1 Parsing into words

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

Listing 2.13: Parsing Strings

```
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.".

2.6.2 Literal patterns

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

Listing 2.14: Parse

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

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

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

Listing 2.15: Parse with comma

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

therefore have the same effect as the previous example.

2.6.3 Positional patterns

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

2.7 Indexed strings

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

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

Listing 2.16: Index

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

or retrieved:

Listing 2.17: Retrieving

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

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

Listing 2.18: Woof

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

would display

```
yap woof grrrrr
```

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

Listing 2.19: Multiple Dimensions

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

which would display

```
ruff woof grrrrr
```

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

Listing 2.20: justonetest.nrx

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

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

2.8 Arrays

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

Listing 2.21: Arrays

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

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

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

Listing 2.22: Initializing elements

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

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

Listing 2.23: Address Array Element

```
say words[1]
```

would then display

```
Ffynnon
```

2.9 Things that aren't strings

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

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

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

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

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

Listing 2.24: Oblong

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

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

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

```

```
14    /* Change the size of a Oblong */
15    method size(new_width, new_height) returns Oblong
16      width=new_width; height=new_height
17      return this  -- return the resized object
18
19      /* Change the size of a Oblong, relatively */
20  method sizerelative(rel_width, rel_height) returns Oblong
21    width=width+rel_width; height=height+rel_height
22    return this
23
24    /* 'Print' what we know about the oblong */
25  method print()
26    say 'Oblong' width 'x' height
```

To summarize:

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

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

Listing 2.25: Try Oblong

```
1 /* tryOblong.nrx -- try the Oblong class */
2 first=Oblong(5,3)      -- make an oblong
3 first.print            -- show it
4 first.relsize(1,1).print -- enlarge and print again
5 second=Oblong(1,2)     -- make another oblong
6 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.

2.9.1 Programs are classes, too

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

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

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

Listing 2.26: New Toast

```
1    /* This wishes you the best of health. */
2    class toast
3      method main(argwords=String[]) static
```

```
4         arg=Rexx(argwords)
5         say 'Cheers!'
```

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

2.10 Extending classes

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

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

Listing 2.27: charOblong.nrx

```
1     /* charOblong.nrx -- an oblong class with character */
2     class charOblong extends Oblong
3       printchar            -- the character for display
4       /* Constructor to make a new oblong with character */
5       method charOblong(newwidth, newheight, newprintchar)
6         super(newwidth, newheight) -- make an oblong
7         printchar=newprintchar      -- and set the character
8       /* 'Print' the oblong */
9       method print
10        loop for super.height
11          say printchar.copies(super.width)
12          end
```

There are several things worth noting about this example:

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

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

Here's a program to try it out

Listing 2.28: tryCharOblong.nrx

```
1     /* trycharOblong.nrx -- try the charOblong class */
2     first=charOblong(5,3,'#') -- make an oblong
3     first.print               -- show it
4     first.relsize(1,1).print -- enlarge and print again
5     second=charOblong(1,2,'*') -- make another oblong
6     second.print              -- and print it
```

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

2.10.1 Optional arguments

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

Listing 2.29: Default value X

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

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

Listing 2.30: Default value

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

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

2.11 Tracing

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

Listing 2.31: Trace

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

would result in:

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

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

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

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

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

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

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

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

Listing 2.32: trace2.nrx

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

would result in:

```
        --- trace2.nrx
  3 *=* a=3
    >v> a "3"
  4 *=* b=4
    >v> b "4"
  8 *=* a=a+1
    >v> a "4"
 10 *=* c=c+1
    >v> c "6"
4 5 6
```

2.12 Binary types and conversions

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

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

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

Further, NetRexx automatically allows the conversion between the various forms of character strings in implementations[10] 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.

2.12.1 Explicit type assignment

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

Listing 2.33: Assigning Type

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

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

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.

2.12.2 Binary types in practice

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

Listing 2.34: A Simple Applet

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

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

2.13 Exception and error handling

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

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

Listing 2.35: Exception

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

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

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

2.14 Summary and Information Sources

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

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

Index

ISBN 978-90-819090-1-3