Preface

1. Open Object Rexx General Concepts
   1.1. What is Object-Oriented Programming? .......................................................... 1
   1.2. Modularizing Data ...................................................................................... 1
   1.3. Modeling Objects ....................................................................................... 3
   1.4. How Objects Interact ............................................................................... 4
   1.5. Methods .................................................................................................. 4
   1.6. Data Abstraction .................................................................................... 5
   1.7. Classes and Instances ........................................................................... 5
   1.8. Polymorphism ....................................................................................... 7
   1.9. Subclasses, Superclasses, and Inheritance ............................................ 7
   1.10. Structure and General Syntax ............................................................. 8
       1.10.1. Characters .................................................................................... 8
       1.10.2. Whitespace ............................................................................... 9
       1.10.3. Comments ............................................................................... 9
       1.10.4. Tokens ................................................................................... 11
       1.10.5. Implied Semicolons ............................................................... 16
       1.10.6. Continuations ......................................................................... 16
   1.11. Terms, Expressions, and Operators ................................................... 17
       1.11.1. Terms and Expressions ............................................................. 17
       1.11.2. Operators ............................................................................... 18
       1.11.3. Parentheses and Operator Precedence .................................. 21
       1.11.4. Message Terms ....................................................................... 23
       1.11.5. Message Sequences ............................................................... 24
       1.11.6. Array Term ........................................................................... 25
       1.11.7. Variable Reference Term ...................................................... 26
   1.12. Clauses and Instructions .................................................................. 26
       1.12.1. NullClauses .......................................................................... 26
       1.12.2. Directives .............................................................................. 27
       1.12.3. Labels ................................................................................... 27
       1.12.4. Instructions ........................................................................... 27
       1.12.5. Assignments .......................................................................... 27
       1.12.6. Message Instructions ............................................................. 28
       1.12.7. Keyword Instructions ............................................................. 28
       1.12.8. Commands .......................................................................... 28
   1.13. Assignments and Symbols ................................................................. 28
       1.13.1. Extended Assignments ......................................................... 29
       1.13.2. Constant Symbols ................................................................ 30
       1.13.3. Simple Symbols .................................................................... 30
       1.13.4. Stems .................................................................................. 30
       1.13.5. Compound Symbols ............................................................. 33
       1.13.6. Environment Symbols ........................................................... 34
   1.14. Namespaces ...................................................................................... 35
   1.15. Message Instructions ....................................................................... 36
2. Keyword Instructions  
2.1. ADDRESS ................................................................. 42  
2.2. ARG ................................................................. 46  
2.3. CALL ................................................................. 47  
2.4. DO ................................................................. 50  
2.5. DROP ................................................................. 52  
2.6. EXIT ................................................................. 53  
2.7. EXPOSE ................................................................. 54  
2.8. FORWARD ................................................................. 55  
2.9. GUARD ................................................................. 57  
2.10. IF ................................................................. 58  
2.11. INTERPRET ................................................................. 59  
2.12. ITERATE ................................................................. 61  
2.13. LEAVE ................................................................. 62  
2.14. LOOP ................................................................. 63  
2.15. NOP ................................................................. 64  
2.16. NUMERIC ................................................................. 64  
2.17. OPTIONS ................................................................. 66  
2.18. PARSE ................................................................. 66  
2.19. PROCEDURE ................................................................. 69  
2.20. PULL ................................................................. 71  
2.21. PUSH ................................................................. 72  
2.22. QUEUE ................................................................. 72  
2.23. RAISE ................................................................. 73  
2.24. REPLY ................................................................. 75  
2.25. RETURN ................................................................. 76  
2.26. SAY ................................................................. 76  
2.27. SELECT ................................................................. 77  
2.28. SIGNAL ................................................................. 79  
2.29. TRACE ................................................................. 81  
2.29.1. Trace Alphabetic Character (Word) Options ......................................................... 82  
2.29.2. Prefix Option ................................................................. 83  
2.29.3. Numeric Options ................................................................. 84  
2.29.4. Tracing Tips ................................................................. 84  
2.29.5. The Format of Trace Output ................................................................. 84  
2.30. USE ................................................................. 86  
2.30.1. USE ARG, USE STRICT ARG ................................................................. 86  
2.30.2. USE LOCAL ................................................................. 88  
3. Directives ................................................................. 90  
3.1. ::ANNOTATE ................................................................. 90  
3.2. ::ATTRIBUTE ................................................................. 91  
3.3. ::CLASS ................................................................. 94  
3.4. ::CONSTANT ................................................................. 96  
3.5. ::METHOD ................................................................. 97  
3.6. ::OPTIONS ................................................................. 100
5.4. Utility Classes ................................................................. 328
  5.4.1. Alarm Class .......................................................... 328
  5.4.2. AlarmNotification Class ........................................ 331
  5.4.3. Buffer Class ........................................................ 332
  5.4.4. Comparable Class .................................................. 332
  5.4.5. Comparator Classes ................................................ 333
  5.4.6. DateTime Class ...................................................... 339
  5.4.7. EventSemaphore Class ........................................... 357
  5.4.8. File Class ............................................................ 360
  5.4.9. MessageNotification Class ..................................... 371
  5.4.10. Monitor Class ..................................................... 371
  5.4.11. MutableBuffer Class ............................................. 373
  5.4.12. MutexSemaphore Class .......................................... 387
  5.4.13. Orderable Class ................................................... 390
  5.4.14. Pointer Class ...................................................... 391
  5.4.15. RegularExpression Class ....................................... 392
  5.4.16. RexxContext Class .............................................. 397
  5.4.17. RexxInfo Class .................................................... 400
  5.4.18. RexxQueue Class ................................................. 409
  5.4.19. Singleton Class (Metaclass) ................................. 413
  5.4.20. StackFrame Class ............................................... 415
  5.4.21. StreamSupplier Class ........................................... 417
  5.4.22. Supplier Class .................................................... 419
  5.4.23. Ticker Class ....................................................... 421
  5.4.24. TimeSpan Class .................................................. 424
  5.4.25. Validate Class .................................................... 431
  5.4.26. VariableReference Class ....................................... 434
  5.4.27. WeakReference Class ........................................... 435

5.3.14. Relation Class ........................................................................ 298
5.3.15. Set Class ............................................................................. 304
5.3.16. Stem Class ........................................................................... 308
5.3.17. StringTable Class .......................................................... 313
5.3.18. Table Class ........................................................................... 318
5.3.19. Sorting Ordered Collections ........................................... 321
5.3.20. Concept of Set Operations ............................................... 324

6. Rexx Runtime Objects ................................................................ 437
  6.1. The Environment Directory (.ENVIRONMENT) .................. 437
     6.1.1. The ENDFILE Line Constant (.ENDFILE) .................. 437
     6.1.2. The FALSE Constant (.FALSE) .............................. 437
     6.1.3. The NULL Object (.NULL) .................................. 437
     6.1.4. The RexxInfo Object (.RexxInfo) ......................... 437
     6.1.5. The TRUE Constant (.TRUE) ................................. 438
  6.2. The Local Directory (.LOCAL) ............................................. 438
  6.3. The Debug Input Monitor (.DEBUGINPUT) ......................... 439
  6.4. The Error Monitor (.ERROR) ............................................ 439
  6.5. The Input Monitor (.INPUT) ............................................ 439
  6.6. The Output Monitor (.OUTPUT) ........................................ 439
  6.7. The Trace Output Monitor (.TRACEOUTPUT) .................... 439
  6.8. The STDERR Stream (.STDERR) ...................................... 439
  6.9. The STDIN Stream (.STDIN) .......................................... 439
  6.10. The STDOUT Stream (.STDOUT) ..................................... 439
  6.11. The STDQUE Queue (.STDQUE) .................................... 440
  6.12. The SYSCARGS Array (.SYSCARGS) .............................. 440
6.13. The Rexx Context (.CONTEXT) ................................................................. 440
6.14. The Line Number (.LINE) ................................................................. 440
6.15. The METHODS StringTable (.METHODS) ........................................ 441
6.16. The ROUTINES StringTable (.ROUTINES) ......................................... 441
6.17. The RESOURCES StringTable (.RESOURCES) .................................... 441
6.18. The Return Status (.RS) ................................................................. 442

7. Functions ............................................................................................................. 443

7.1. Syntax .................................................................................................................. 443
7.2. Functions and Subroutines .............................................................................. 443
  7.2.1. Search Order .............................................................................................. 444
  7.2.2. Errors during Execution ........................................................................... 448

7.3. Return Values ..................................................................................................... 448
7.4. Built-in Functions ............................................................................................. 449
  7.4.1. ABBREV (Abbreviation) ............................................................................ 450
  7.4.2. ABS (Absolute Value) .............................................................................. 451
  7.4.3. ADDRESS ................................................................................................. 451
  7.4.4. ARG (Argument) ....................................................................................... 451
  7.4.5. B2X (Binary to Hexadecimal) ................................................................. 453
  7.4.6. BEEP ........................................................................................................ 453
  7.4.7. BITAND (Bit by Bit AND) ........................................................................ 454
  7.4.8. BITOR (Bit by Bit OR) ............................................................................. 455
  7.4.9. BITXOR (Bit by Bit Exclusive OR) .......................................................... 455
  7.4.10. C2D (Character to Decimal) ................................................................. 456
  7.4.11. C2X (Character to Hexadecimal) .......................................................... 456
  7.4.12. CENTER (or CENTRE) .......................................................................... 457
  7.4.13. CHANGESTR .......................................................................................... 457
  7.4.14. CHARIN (Character Input) ................................................................. 458
  7.4.15. CHAROUT (Character Output) ............................................................... 459
  7.4.16. CHARS (Characters Remaining) ........................................................... 460
  7.4.17. COMPARE ............................................................................................ 460
  7.4.18. CONDITION .......................................................................................... 460
  7.4.19. COPIES ................................................................................................. 462
  7.4.20. COUNTSTR .......................................................................................... 462
  7.4.21. D2C (Decimal to Character) ................................................................. 463
  7.4.22. D2X (Decimal to Hexadecimal) ............................................................ 463
  7.4.23. DATATYPE ............................................................................................ 464
  7.4.24. DATE .................................................................................................... 465
  7.4.25. DELSTR (Delete String) ........................................................................ 469
  7.4.26. DELWORD (Delete Word) ................................................................. 470
  7.4.27. DIGITS ................................................................................................. 470
  7.4.28. DIRECTORY ......................................................................................... 470
  7.4.29. ENDOLOCAL (Linux only) ................................................................. 471
  7.4.30. ERRORTEXT ...................................................................................... 471
  7.4.31. FILESPEC ............................................................................................ 472
  7.4.32. FORM .................................................................................................. 472
  7.4.33. FORMAT .............................................................................................. 473
  7.4.34. FUZZ .................................................................................................... 474
  7.4.35. INSERT ................................................................................................. 474
  7.4.36. LASTPOS (Last Position) ................................................................. 474
  7.4.37. LEFT .................................................................................................... 475
  7.4.38. LENGTH .............................................................................................. 475
  7.4.39. LINEIN (Line Input) ....................................................................... 475
  7.4.40. LINEOUT (Line Output) ................................................................. 477
8. Rexx Utilities (RexxUtil) .......................................................... 512
  8.1. A Note on Error Codes ...................................................... 512
  8.2. List of Rexx Utility Functions ......................................... 512
  8.3. RxMessageBox (Windows only) ........................................ 514
  8.4. RxWinExec (Windows only) .............................................. 516
  8.5. SysAddRexxMacro ............................................................ 518
  8.6. SysBootDrive (Windows only) .......................................... 518
  8.7. SysClearRexxMacroSpace ............................................... 518
  8.8. SysCls ................................................................. 518
  8.9. SysCreatePipe (Unix-like systems only) ......................... 519
  8.10. SysCurPos (Windows only) ............................................ 519
9. Parsing 567
  9.1. Simple Templates for Parsing into Words ................................................. 567
    9.1.1. Message Term Assignments ...................................................... 569
    9.1.2. The Period as a Placeholder .................................................... 569
  9.2. Templates Containing String Patterns ..................................................... 569
  9.3. Templates Containing Positional (Numeric) Patterns ................................. 571
    9.3.1. Combining Patterns and Parsing into Words ................................. 574
  9.4. Parsing with Variable Patterns ............................................................... 576
  9.5. Using UPPER, LOWER, and CASELESS .................................................. 576
  9.6. Parsing Instructions Summary ............................................................... 577
  9.7. Parsing Instructions Examples ............................................................... 577
  9.8. Advanced Topics in Parsing ................................................................. 579
    9.8.1. Parsing Several Strings ............................................................ 579
    9.8.2. Combining String and Positional Patterns ..................................... 579
    9.8.3. Conceptual Overview of Parsing ............................................... 580

10. Numbers and Arithmetic 584
  10.1. Precision .............................................................................................. 585
  10.2. Arithmetic Operators ............................................................................. 585
    10.2.1. Power ......................................................................................... 586
    10.2.2. Integer Division ....................................................................... 586
    10.2.3. Remainder ............................................................................... 586
    10.2.4. Operator Examples .................................................................... 586
  10.3. Exponential Notation ............................................................................ 587
  10.4. Numeric Comparisons .......................................................................... 588
  10.5. Limits and Errors when Rexx Uses Numbers Directly ............................ 589

11. Conditions and Condition Traps 591
  11.1. Action Taken when a Condition Is Not Trapped ...................................... 595
  11.2. Action Taken when a Condition Is Trapped ........................................... 595
  11.3. Condition Information .......................................................................... 597
    11.3.1. Descriptive Strings ................................................................... 597
    11.3.2. Additional Object Information ................................................. 598
    11.3.3. The Special Variable RC .......................................................... 598
    11.3.4. The Special Variable SIGL ....................................................... 598
    11.3.5. Condition Object .................................................................... 599

12. Concurrency 601
  12.1. Early Reply ......................................................................................... 601
  12.2. Message Objects .................................................................................. 603
  12.3. Default Concurrency ........................................................................... 603
    12.3.1. Sending Messages within an Activity ....................................... 605
  12.4. Using Additional Concurrency Mechanisms ......................................... 606
12.4.1. SETUNGUARDED Method and UNGUARDED Option ....................................... 606
12.4.2. GUARD ON and GUARD OFF ........................................................................ 607
12.4.3. Guarded Methods ........................................................................................... 607
12.4.4. Additional Examples ....................................................................................... 608

13. The Security Manager 614
   13.1. Calls to the Security Manager ....................................................................... 614
       13.1.1. Example .................................................................................................. 616

14. Input and Output Streams 619
   14.1. The Input and Output Model ......................................................................... 619
       14.1.1. Input Streams ...................................................................................... 620
       14.1.2. Output Streams ................................................................................... 620
       14.1.3. External Data Queue .......................................................................... 621
       14.1.4. Default Stream Names ........................................................................ 623
       14.1.5. Line versus Character Positioning ....................................................... 624
       14.2. Implementation ......................................................................................... 625
       14.3. Operating System Specifics ....................................................................... 625
       14.4. Examples of Input and Output ................................................................. 625
       14.5. Errors during Input and Output ............................................................... 626
       14.6. Summary of Rexx I/O Instructions and Methods ...................................... 627

15. Debugging Aids 629
   15.1. Interactive Debugging of Programs .............................................................. 629
   15.2. Debugging Aids ............................................................................................ 629
       15.3. RXTRACE Variable ................................................................................. 630

16. Reserved Keywords 631

17. Special Variables 632

18. Useful Services 634
   18.1. Windows Commands .................................................................................... 634
   18.2. Linux Commands ........................................................................................ 634
   18.3. Subcommand Handler Services ................................................................... 635
       18.3.1. The RXSUBCOM Command ............................................................. 635
   18.4. The RXQUEUE Filter ................................................................................. 637
   18.5. Distributing Programs without Source ........................................................... 639

A. Using DO and LOOP 641
   A.1. Simple DO Group .......................................................................................... 641
   A.2. Repetitive Loops .......................................................................................... 641
       A.2.1. Simple Repetitive Loops ....................................................................... 641
       A.2.2. Controlled Repetitive Loops ................................................................. 641
       A.2.3. Repetitive Loops over Collections ....................................................... 643
       A.2.4. Repetitive Loops over Suppliers ........................................................... 644
   A.3. Conditional Phrases (WHILE and UNTIL) .................................................... 644
   A.4. LABEL Phrase .............................................................................................. 645
   A.5. COUNTER Phrase ......................................................................................... 645
   A.6. Conceptual Model of Loops .......................................................................... 647

B. Migration 649
   B.1. Incompatible ooRexx features ....................................................................... 649
       B.1.1. RexxUtil SysTempFileName .................................................................. 649
   B.2. Deprecated Rexx features ............................................................................ 649
       B.2.1. RexxUtil Semaphore functions ........................................................... 649
       B.2.2. RexxUtil SysLoadFuncs/SysDropFuncs ............................................. 649
       B.2.3. ::OPTIONS NOVALUE ERROR directive .......................................... 649
C. Error Numbers and Messages

C.1. Error List

C.1.1. Error 3 - Failure during initialization.
C.1.2. Error 4 - Program interrupted.
C.1.3. Error 5 - System resources exhausted.
C.1.4. Error 6 - Unmatched "/*" or quote.
C.1.5. Error 7 - WHEN or OTHERWISE expected.
C.1.6. Error 8 - Unexpected THEN or ELSE.
C.1.7. Error 9 - Unexpected WHEN or OTHERWISE.
C.1.8. Error 10 - Unexpected or unmatched END.
C.1.9. Error 11 - Control stack full.
C.1.10. Error 13 - Invalid character in program.
C.1.11. Error 14 - Incomplete DO/LOOP/SELECT/IF.
C.1.12. Error 15 - Invalid hexadecimall or binary string.
C.1.14. Error 17 - Unexpected PROCEDURE.
C.1.15. Error 18 - THEN expected.
C.1.16. Error 19 - String or symbol expected.
C.1.17. Error 20 - Symbol expected.
C.1.18. Error 21 - Invalid data on end of clause.
C.1.20. Error 23 - Invalid data string.
C.1.21. Error 24 - Invalid TRACE request.
C.1.22. Error 25 - Invalid subkeyword found.
C.1.23. Error 26 - Invalid whole number.
C.1.24. Error 27 - Invalid DO or LOOP syntax.
C.1.25. Error 28 - Invalid LEAVE or ITERATE.
C.1.27. Error 30 - Name or symbol too long.
C.1.28. Error 31 - Name starts with number or ".".
C.1.29. Error 33 - Invalid expression result.
C.1.30. Error 34 - Logical value not 0 or 1.
C.1.31. Error 35 - Invalid expression.
C.1.32. Error 36 - Unmatched "(" or "[" in expression.
C.1.33. Error 37 - Unexpected ",", ")", or "]".
C.1.34. Error 38 - Invalid template or pattern.
C.1.35. Error 39 - Evaluation stack overflow.
C.1.36. Error 40 - Incorrect call to routine.
C.1.37. Error 41 - Bad arithmetic conversion.
C.1.38. Error 42 - Arithmetic overflow/underflow.
C.1.39. Error 43 - Routine not found.
C.1.40. Error 44 - Function or message did not return data.
C.1.41. Error 45 - No data specified on function RETURN.
C.1.42. Error 46 - Invalid variable reference.
C.1.43. Error 47 - Unexpected label.
C.1.44. Error 48 - Failure in system service.
C.1.45. Error 49 - Interpretation error.
C.1.46. Error 88 - Invalid argument.
C.1.47. Error 89 - Variable or message term expected.
C.1.48. Error 90 - External name not found.
C.1.49. Error 91 - No result object.
C.1.50. Error 92 - OLE error.
C.1.51. Error 93 - Incorrect call to method. ................................................................. 683
C.1.52. Error 97 - Object method not found. ............................................................... 687
C.1.53. Error 98 - Execution error. ........................................................... 687
C.1.54. Error 99 - Translation error. ................................................................. 691
C.2. RXSUBCOM Utility Program ............................................................................. 694
C.2.1. Error 116 - The RXSUBCOM REGISTER parameters are incorrect .......... 694
C.2.2. Error 117 - The RXSUBCOM DROP parameters are incorrect ........................ 694
C.2.3. Error 118 - The RXSUBCOM LOAD parameters are incorrect ................. 694
C.2.4. Error 125 - The RXSUBCOM QUERY parameters are incorrect .......... 695
C.3. RXQUEUE Utility Program .............................................................................. 695
C.3.1. Error 119 - The REXX rxapi queuing system is not available ....................... 695
C.3.2. Error 120 - The size of the data is incorrect ................................................. 695
C.3.3. Error 121 - Storage for data queues is exhausted ......................................... 695
C.3.4. Error 122 - The name %1 is not a valid queue name .................................... 695
C.3.5. Error 123 - The queue access mode is not correct ....................................... 695
C.3.6. Error 124 - The queue %1 does not exist ...................................................... 695
C.3.7. Error 131 - The syntax of the command is incorrect .................................... 696
C.3.8. Error 132 - System error occurred while processing the command .......... 696
C.4. rexxc Utility Program .................................................................................... 696
C.4.1. Error 127 - The rexxc command parameters are incorrect ......................... 696
C.4.2. Error 128 - Output file name must be different from input file name .......... 696
C.4.4. Error 130 - Without outputfile rexxc only performs a syntax check .......... 696
D. Notices ............................................................................................................. 697
D.1. Trademarks ..................................................................................................... 697
D.2. Source Code For This Document ................................................................. 698
E. Common Public License Version 1.0 .................................................................. 699
E.1. Definitions ...................................................................................................... 699
E.2. Grant of Rights ............................................................................................... 699
E.3. Requirements ................................................................................................ 700
E.4. Commercial Distribution ............................................................................... 700
E.5. No Warranty .................................................................................................. 701
E.6. Disclaimer of Liability .................................................................................... 701
E.7. General ........................................................................................................... 701
F. Revision History ............................................................................................... 703
Index ...................................................................................................................... 704
Preface

This book describes the Open Object Rexx Interpreter, called the interpreter or language processor in the following, and the object-oriented Rexx language.

This book is intended for people who plan to develop applications using Rexx. Its users range from the novice, who might have experience in some programming language but no Rexx experience, to the experienced application developer, who might have had some experience with Open Object Rexx.

This book is a reference rather than a tutorial. It assumes you are already familiar with object-oriented programming concepts.

Descriptions include the use and syntax of the language and explain how the language processor “interprets” the language as a program is running.

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

1.1. Typographic Conventions

Typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

Mono-spaced Bold is used to highlight literal strings, class names, or inline code examples. For example:

```
The Class class comparison methods return .true or .false, the result of performing the comparison operation.

This method is exactly equivalent to subWord(n, 1).
```

Mono-spaced Normal denotes method names or source code in program listings set off as separate examples.

```
This method has no effect on the action of any hasEntry, hasIndex, items, remove, or supplier message sent to the collection.
```

Proportional Italic is used for method and function variables and arguments.

```
A supplier loop specifies one or two control variables, index, and item, which receive a different value on each repetition of the loop.

Returns a string of length length with string centered in it and with pad characters added as necessary to make up length.
```

1.2. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.
2. How to Read the Syntax Diagrams

Throughout this book, syntax is described using the structure defined below.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.
  
  The ▶️ symbol indicates the beginning of a statement.
  
  The ----- symbol indicates that the statement syntax is continued on the next line.
  
  The ---- symbol indicates that a statement is continued from the previous line.
  
  The ➞ symbol indicates the end of a statement.

- Required items appear on the horizontal line (the main path).

- Optional items appear below the main path.

- If you can choose from two or more items, they appear vertically, in a stack. If you must choose one of the items, one item of the stack appears on the main path.

- If choosing one of the items is optional, the entire stack appears below the main path.
3. Getting Help and Submitting Feedback

The Open Object Rexx Project has a number of methods to obtain help and submit feedback for ooRexx and the extension packages that are part of ooRexx. These methods, in no particular order of preference, are listed below.

3.1. The Open Object Rexx SourceForge Site

Open Object Rexx utilizes SourceForge to house its source repositories, mailing lists and other project features at https://sourceforge.net/projects/oorexx. ooRexx uses the Developer and User mailing lists at https://sourceforge.net/p/oorexx/mailman for discussions concerning ooRexx. The ooRexx user is most likely to get timely replies from one of these mailing lists.
Here is a list of some of the most useful facilities provided by SourceForge.

The Developer Mailing List

The Users Mailing List
Subscribe to the oorexx-users mailing list at https://lists.sourceforge.net/lists/listinfo/oorexx-users to discuss how to use ooRexx. It also supports a historical archive of past messages.

The Announcements Mailing List
Subscribe to the oorexx-announce mailing list at https://lists.sourceforge.net/lists/listinfo/oorexx-announce to receive announcements of significant ooRexx project events.

The Bug Mailing List
Subscribe to the oorexx-bugs mailing list at https://lists.sourceforge.net/lists/listinfo/oorexx-bugs to monitor changes in the ooRexx bug tracking system.

Bug Reports
You can view ooRexx bug reports at https://sourceforge.net/p/oorexx/bugs. To be able to create new bug reports, you will need to first register for a SourceForge userid at https://sourceforge.net/user/registration. When reporting a bug, please try to provide as much information as possible to help developers determine the cause of the issue. Sample program code that can reproduce your problem will make it easier to debug reported problems.

Documentation Feedback
You can submit feedback for, or report errors in, the documentation at https://sourceforge.net/p/oorexx/documentation. Please try to provide as much information in a documentation report as possible. In addition to listing the document and section the report concerns, direct quotes of the text will help the developers locate the text in the source code for the document. (Section numbers are generated when the document is produced and are not available in the source code itself.) Suggestions as to how to reword or fix the existing text should also be included.

Request For Enhancement
You can new suggest ooRexx features or enhancements at https://sourceforge.net/p/oorexx/feature-requests.

Patch Reports
If you create an enhancement patch for ooRexx please post the patch at https://sourceforge.net/p/oorexx/patches. Please provide as much information in the patch report as possible so that the developers can evaluate the enhancement as quickly as possible.

Please do not post bug fix patches here, instead you should open a bug report at https://sourceforge.net/p/oorexx/bugs and attach the patch to it.

The ooRexx Forums
The ooRexx project maintains a set of forums that anyone may contribute to or monitor. They are located at https://sourceforge.net/p/oorexx/discussion. There are currently three forums available: Help, Developers and Open Discussion. In addition, you can monitor the forums via email.

3.2. The Rexx Language Association Mailing List
3.3. comp.lang.rexx Newsgroup
The comp.lang.rexx newsgroup at https://groups.google.com/forum/#!forum/comp.lang.rexx is a good place to obtain help from many individuals within the Rexx community. You can obtain help on Open Object Rexx and other Rexx interpreters and tools.

4. Related Information
See also Open Object Rexx: Programmer Guide and Open Object Rexx: Application Programming Interfaces.
Open Object Rexx General Concepts

The Rexx language is particularly suitable for:

• Application scripting
• Command procedures
• Application front ends
• User-defined macros (such as editor subcommands)
• Prototyping
• Personal computing

As an object-oriented language, Rexx provides data encapsulation, polymorphism, an object class hierarchy, class-based inheritance of methods, and concurrency. It includes a number of useful base classes and allows you create new object classes of your own.

Open Object Rexx is compatible with earlier Rexx versions, both non-object based Rexx and IBM's Object Rexx. It has the usual structured-programming instructions, for example IF, SELECT, DO WHILE, and LEAVE, and a number of useful built-in functions.

The language imposes few restrictions on the program format. There can be more than one clause on a line, or a single clause can occupy more than one line. Any indentation scheme is allowed. You can, therefore, code programs in a format that emphasizes their structure, making them easier to read.

There is no limit to the size of variable values, as long as all values fit into the storage available. There are no restrictions on the types of data that variables can contain.

A language processor (interpreter) runs Rexx programs. That is, the program runs line by line and word by word, without first being translated (compiled) to machine language. One of the advantages of this is that you can fix the error and rerun the program faster than when using a compiler.

**Note:** Open Object Rexx also supplies the `rexxc` program that can be used to translate Rexx programs into a sourceless executable file. Translating a program is not the same as compiling a program to machine language. A translated Rexx program will still be interpreted line by line, though it will typically start faster as the initial parsing has already been done.

1.1. What Is Object-Oriented Programming?

Object-oriented programming is a way to write computer programs by focusing not on the instructions and operations a program uses to manipulate data, but on the data itself. First, the program simulates, or models, objects in the physical world as closely as possible. Then the objects interact with each other to produce the desired result.

Real-world objects, such as a company's employees, money in a bank account, or a report, are stored as data so the computer can act upon it. For example, when you print a report, print is the action and report is the object acted upon. Essentially, the objects are the "nouns", while the actions are the "verbs".

1.2. Modularizing Data
Modularizing Data

In conventional, structured programming, actions like print are often isolated from the data by placing them in subroutines or modules. A module typically contains an operation for implementing one simple action. You might have a PRINT module, a SEND module, an ERASE module. The data these modules operate on must be constructed by the programmer and passed to the modules to perform an action.

```
PROGRAM ...
   ------------------------
   PRINT ----------------
   SEND ..............
   ERASE ..............
```

But with object-oriented programming, it is the data that is modularized. And each data module includes its own operations for performing actions directly related to its data. The programmer that uses the objects need only be aware of the operations an object performs and not how the data is organized internally.

```
   PRINT
   Report
   SEND
   FILE
   ERASE
```

Figure 1.1. Modular Data—a Report Object

In the case of report, the report object would contain its own built-in PRINT, SEND, ERASE, and FILE operations.

Object-oriented programming lets you model real-world objects—even very complex ones—precisely and elegantly. As a result, object manipulation becomes easier and computer instructions become simpler and can be modified later with minimal effort.
Object-oriented programming *hides* any information that is not important for acting on an object, thereby concealing the object's complexities. Complex tasks can then be initiated simply, at a very high level.

### 1.3. Modeling Objects

In object-oriented programming, objects are modeled to real-world objects. A real-world object has actions related to it and characteristics of its own.

Take a ball, for example. A ball can be acted on—rolled, tossed, thrown, bounced, caught. But it also has its own physical characteristics—size, shape, composition, weight, color, speed, position. An accurate data model of a real ball would define not only the physical characteristics but *all* related actions and characteristics in one package:

![Figure 1.2. A Ball Object](image)

In object-oriented programming, objects are the basic building blocks—the fundamental units of data.

There are many kinds of objects; for example, character strings, collections, and input and output streams. An object—such as a character string—always consists of two parts: the possible actions or operations related to it, and its characteristics or variables. A variable has a *name*, and an associated data value that can change over time. The variables represent the internal state of the object, and can be directly accessed only by the code that implements the object's actions.

![Figure 1.3. Ball Object with Variable Names and Values](image)

To access an object's data, you must always specify an action. For example, suppose the object is the number 5. Its actions might include addition, subtraction, multiplication, and division. Each of these
actions is an interface to the object's data. The data is said to be encapsulated because the only way to access it is through one of these surrounding actions. The encapsulated internal characteristics of an object are its variables. The variables are associated with an object and exist for the lifetime of that object:

![Figure 1.4. Encapsulated 5 Object](image)

### 1.4. How Objects Interact

The actions defined by an object are its only interface to other objects. Actions form a kind of "wall" that encapsulates the object, and shields its internal information from outside objects. This shielding is called information hiding. Information hiding protects an object's data from corruption by outside objects, and also protects outside objects from relying on another object's private data, which can change without warning.

One object can act upon another (or cause it to act) only by calling that object's actions, namely by sending messages. Objects respond to these messages by performing an action, returning data, or both. A message to an object must specify:

- A receiving object
- The “message send” symbol, ~, which is called the twiddle
- The action and, optionally in parentheses, any parameters required by the action

So the message format looks like this:

\[
\text{object~action(parameters)}
\]

Assume that the object is the string \!iH. Sending it a message to use its REVERSE action:

\[
"!iH"\text{~reverse}
\]

returns the string object Hi!.

### 1.5. Methods

Sending a message to an object results in performing some action; that is, it executes some underlying code. The action-generating code is called a method. When you send a message to an object, the message is the name of the target method. Method names are character strings like
Data Abstraction

**reverse**. In the preceding example, sending the `reverse` message to the `!iH` object causes it to run the `reverse` method. Most objects are capable of more than one action, and so have a number of available methods.

The classes Rexx provides include their own predefined methods. The Message class, for example, has completed, init, notify, result, send, and start methods. When you create your own classes, you can write new methods for them in Rexx code. Much of the object programming in Rexx is writing the code for the methods you create.

Rexx lets you send the same message to objects that are different:

<table>
<thead>
<tr>
<th>Example 1.1. Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;!iH&quot;-reverse         -- Reverses the characters &quot;!iH&quot; to form &quot;Hi!&quot;</td>
</tr>
<tr>
<td>pen-reverse           -- Reverses the direction of a plotter pen</td>
</tr>
<tr>
<td>ball-reverse          -- Reverses the direction of a moving ball</td>
</tr>
</tbody>
</table>

As long as each object has its own `reverse` method, `reverse` runs even if the programming implementation is different for each object. Each object knows only its own version of `reverse`. And even though the objects are different, each reverses itself as dictated by its own code.

Although the `!iH` object's `reverse` code is different from the plotter pen's, the method name can be the same because Rexx keeps track of the methods each object owns. You do not need to have several message names like `reverse_string`, `reverse_pen`, `reverse_ball`. This keeps method-naming schemes simple and makes complex programs easy to follow and modify.

### 1.6. Data Abstraction

The ability to create new, high-level data types and organize them into a meaningful class structure is called **data abstraction**. Data abstraction is at the core of object-oriented programming. Once you model objects with real-world properties from the basic data types, you can continue creating, assembling, and combining them into increasingly complex objects. Then you can use these objects as if they were part of the original programming language.

### 1.7. Classes and Instances

In Rexx, objects are organized into **classes**. Classes are like templates; they define the methods and variables that a group of similar objects have in common and store them in one place.

If you write a program to manipulate some screen icons, for example, you might create an `Icon` class. In that `Icon` class you can include all the icon objects with similar actions and characteristics:
All the icon objects might use common methods like DRAW or ERASE. They might contain common variables like position, color, or size. What makes each icon object different from one another is the data assigned to its variables. For the Windows system icon, it might be position="20,20", while for the shredder it is "20,30" and for information it is "20,40":

Objects that belong to a class are called instances of that class. As instances of the Icon class, the Windows system icon, shredder icon, and information icon acquire the methods and variables of that class. Instances behave as if they each had their own methods and variables of the same name. All instances, however, have their own unique properties—the data associated with the variables. Everything else can be stored at the class level.
If you must update or change a particular method, you only have to change it at one place, at the class level. This single update is then acquired by every new instance that uses the method.

A class that can create instances of an object is called an object class. The Icon class is an object class you can use to create other objects with similar properties, such as an application icon or a drives icon.

An object class is like a factory for producing instances of the objects.

### 1.8. Polymorphism

Polymorphism gives you a single interface to objects of different types. This example shows instances of classes inheriting from the Collection class, all sharing a common method named put, but with a different implementation each.

#### Example 1.2. Polymorphism

```plaintext
stem~put("value", "tail")         -- sets a Stem tail to "value"
stringTable~put("value", "index") -- sets a StringTable "index" to "value"
array-put("value", 1)             -- sets Array index 1 to "value"
set-put("value")                  -- makes "value" a member of the Set
```

The ability to hide the various implementations of a method while leaving the interface the same illustrates polymorphism. On a higher level, polymorphism permits extensive code reuse.

### 1.9. Subclasses, Superclasses, and Inheritance

When you write your first object-oriented program, you do not have to begin your real-world modeling from scratch. Rexx provides predefined classes and methods. From there you can create additional classes and methods of your own, according to your needs.

Rexx classes are hierarchical. Any subclass (a class below another class in the hierarchy) inherits the methods and variables of one or more superclasses (classes above a class in the hierarchy):
You can add a class to an existing superclass. For example, you might add the Icon class to the Screen-Object superclass:

In this way, the subclass inherits additional methods from the superclass. A class can have more than one superclass, for example, subclass Bitmap might have the superclasses Screen-Object and Art-Object. Acquiring methods and variables from more than one superclass is known as multiple inheritance:

A Rexx program is built from a series of clauses that are composed of:

- Zero or more whitespace characters (blank or horizontal tabs) (which are ignored)
- A sequence of tokens (see Section 1.10.4, "Tokens")
- Zero or more whitespace characters (again ignored)
- A semicolon (;) delimiter that the line end, certain keywords, or the colon (:) implies.

Conceptually, each clause is scanned from left to right before processing, and the tokens composing it are identified. Instruction keywords are recognized at this stage, comments are removed, and sequences of whitespace characters (except within literal strings) are converted to single blanks. Whitespace characters adjacent to operator characters and special characters are also removed.

### 1.10.1. Characters

A character is a member of a defined set of elements that is used for the control or representation of data. You can usually enter a character with a single keystroke. The coded representation of a character is its representation in digital form. A character, the letter A, for example, differs from its coded representation or encoding. Various coded character sets (such as ASCII and EBCDIC) use different encodings for the letter A (decimal values 65 and 193, respectively). This book uses
Characters to convey meanings and not to imply a specific character code, except where otherwise stated. The exceptions are certain built-in functions that convert between characters and their representations. The functions C2D, C2X, D2C, X2C, and XRANGE depend on the character set used.

A code page specifies the encodings for each character in a set. Be aware that:
- Some code pages do not contain all characters that Rexx defines as valid (for example, the logical NOT character).
- Some characters that Rexx defines as valid have different encodings in different code pages, for example the exclamation mark (!).

### 1.10.2. Whitespace

A whitespace character is one that the interpreter recognizes as a “blank” or “space” character. There are two characters used by Rexx as whitespace that can be used interchangeably:

- **(blank)**
  A “blank” or “space” character. This is represented by ‘20’X in ASCII implementations.

- **(horizontal tab)**
  A “tab”. This is represented by ‘09’X in ASCII implementations.

Horizontal tabs encountered in Rexx program source are converted into blanks, allowing tab characters and blanks to be use interchangeably in source. Additionally, Rexx operations such as the PARSE instruction or the SUBWORD() built-in function will also accept either blank or tab characters as word delimiters.

### 1.10.3. Comments

A comment is a sequence of characters delimited by specific characters. It is ignored by the program but acts as a separator. For example, a token containing one comment is treated as two tokens.

The interpreter recognizes the following types of comments:

- A line comment, where the comment is limited to one line
- The standard Rexx comment, where the comment can cover several lines

A **line comment** is started by two subsequent minus signs (--) and ends at the end of a line. Example:

```
"Fred"
"Don't Panic!"
'You shouldn't'
  -- Same as "You shouldn't"
```

In this example, the language processor processes the statements from 'Fred' to 'You shouldn't', ignores the words following the line comment, and continues to process the statement ""

A **standard comment** is a sequence of characters (on one or more lines) delimited by /* and */. Within these delimiters any characters are allowed. Standard comments can contain other standard
Comments

comments, as long as each begins and ends with the necessary delimiters. They are called nested comments. Standard comments can be anywhere and of any length.

/* This is an example of a valid Rexx comment */

Take special care when commenting out lines of code containing /* or */ as part of a literal string. Consider the following program segment:

Example 1.3. Comments

```rexx
01 parse pull input
02 if substr(input,1,5) = "/*123"
03 then call process
04 dept = substr(input,32,5)
```

To comment out lines 2 and 3, the following change would be incorrect:

```rexx
01 parse pull input
02 /* if substr(input,1,5) = "/*123"
03 then call process
04 */ dept = substr(input,32,5)
```

This is incorrect because the language processor would interpret the /* that is part of the literal string /*123 as the start of a nested standard comment. It would not process the rest of the program because it would be looking for a matching standard comment end (*/).

You can avoid this type of problem by using concatenation for literal strings containing /* or */; line 2 would be:

```rexx
if substr(input,1,5) = "/" || "*123"
```

You could comment out lines 2 and 3 correctly as follows:

Example 1.4. Comments

```rexx
01 parse pull input
02 /* if substr(input,1,5) = "/" || "*123"
03 then call process
04 */ dept = substr(input,32,5)
```

Both types of comments can be mixed and nested. However, when you nest the two types, the type of comment that comes first takes precedence over the one nested. Here is an example:

Example 1.5. Comments

```rexx
"Fred"
"Don't Panic!"
'You shouldn''t'        /* Same as "You shouldn't"
"
```

In this example, the language processor ignores everything after 'You shouldn''t' up to the end of the last line. In this case, the standard comment has precedence over the line comment.
When nesting the two comment types, make sure that the start delimiter of the standard comment */ is not in the line commented out with the line comment signs.

Example 1.6. Comments

```
"Fred"
"Don't Panic!"
'You shouldn't' -- Same as /* "You shouldn't"
"
The null string */
```

This example produces an error because the language processor ignores the start delimiter of the standard comment, which is commented out using the line comment.

1.10.4. Tokens

A token is the unit of low-level syntax from which clauses are built. Programs written in Rexx are composed of tokens. Tokens can be of any length, up to an implementation-restricted maximum. They are separated by whitespace or comments, or by the nature of the tokens themselves. The classes of tokens are:

- Literal strings
- Hexadecimal strings
- Binary strings
- Symbols
- Numbers
- Operator characters
- Special characters

1.10.4.1. Literal Strings

A literal string is a sequence including any characters except line-end and end-of-file characters, and delimited by a single quotation mark (') or a double quotation mark ("). You use two consecutive double quotation marks (""") to represent one double quotation mark (") within a literal string delimited by double quotation marks. Similarly, you use two consecutive single quotation marks (' ') to represent one single quotation mark (') within a string delimited by single quotation marks. A literal string is a constant and its contents are never modified when it is processed. Literal strings must be complete on a single line. This means that unmatched quotation marks can be detected on the line where they occur.

A literal string with no characters (that is, a string of length 0) is called a null string.

These are valid strings:

Example 1.7. Valid strings

```
"Fred"
"Don't Panic!"
```
A literal string has no upper bound on the number of characters, limited only by available memory.

Note that a string immediately followed by a left parenthesis is considered to be the name of a function. If immediately followed by the symbol X or x, it is considered to be a hexadecimal string. If followed immediately by the symbol B or b, it is considered to be a binary string.

### 1.10.4.2. Hexadecimal Strings

A hexadecimal string is a literal string, expressed using a hexadecimal notation of its encoding. It is any sequence of zero or more hexadecimal digits (0-9, a-f, A-F), grouped in pairs. A single leading 0 is assumed, if necessary, at the beginning of the string to make an even number of hexadecimal digits. The groups of digits are optionally separated by one or more whitespace characters, and the whole sequence is delimited by single or double quotation marks and immediately followed by the symbol x or X. Neither x nor X can be part of a longer symbol. The whitespace characters, which can only be on byte boundaries (and not at the beginning or end of the string), are to improve readability. The language processor ignores them.

A hexadecimal string is a literal string formed by packing the hexadecimal digits given. Packing the hexadecimal digits removes whitespace and converts each pair of hexadecimal digits into its equivalent character, for example, '41'x to A.

Hexadecimal strings let you include characters in a program even if you cannot directly enter the characters themselves. These are valid hexadecimal strings:

**Example 1.8. Valid hexadecimal strings**

```
"ABCD"x
"1d ec f8"X
'1 d8'x
```

**Note**

A hexadecimal string is not a representation of a number. It is an escape mechanism that lets a user describe a character in terms of its encoding (and, therefore, is machine-dependent). In ASCII, '20'x is the encoding for a blank. In every case, a string of the form '.....'x is an alternative to a straightforward string. In ASCII '41'x and 'A' are identical, as are '20'x and a blank, and must be treated identically.

The packed length of a hexadecimal string (the string with whitespace removed) is unlimited.

### 1.10.4.3. Binary Strings

A binary string is a literal string, expressed using a binary representation of its encoding. It is any sequence of zero or more binary digits (0 or 1) in groups of 8 (bytes) or 4 (nibbles). The first group can have less than four digits; in this case, up to three 0 digits are assumed to the left of the first digit,
making a total of four digits. The groups of digits are optionally separated by one or more whitespace characters, and the whole sequence is delimited by matching single or double quotation marks and immediately followed by the symbol \texttt{b} or \texttt{B}. Neither \texttt{b} nor \texttt{B} can be part of a longer symbol. The whitespace characters, which can only be byte or nibble boundaries (and not at the beginning or end of the string), are to improve readability. The language processor ignores them.

A binary string is a literal string formed by packing the binary digits given. If the number of binary digits is not a multiple of 8, leading zeros are added on the left to make a multiple of 8 before packing. Binary strings allow you to specify characters explicitly, bit by bit. These are valid binary strings:

\begin{example}
\begin{verbatim}
"11110000"b /* == "f0"x */
"101 1101"b /* == "5d"x */
'1'b /* == '0000001'b or '01'x */
"10000 10101010"b /* == "0001 0000 1010 1010"b */
""b /* == "" */
\end{verbatim}
\end{example}

The packed length of a binary-literal string is unlimited.

### 1.10.4.4. Symbols

Symbols are groups of characters, selected from the:

- English alphabetic characters (\texttt{A-Z} and \texttt{a-z}).
- Numeric characters (\texttt{0-9}).
- Characters \texttt{.} ! ? and underscore (_).

Any lowercase alphabetic character in a symbol is translated to uppercase (that is, lowercase \texttt{a-z} to uppercase \texttt{A-Z}) before use.

These are valid symbols:

\begin{example}

Fred
Albert.Hall
WHERE?
\end{example}

If a symbol does not begin with a digit or a period, you can use it as a variable and can assign it a value. If you have not assigned a value to it, its value is the characters of the symbol itself, translated to uppercase (that is, lowercase \texttt{a-z} to uppercase \texttt{A-Z}). Symbols that begin with a number or a period are \textit{constant symbols} and cannot directly be assigned a value.

One other form of symbol is allowed to support the representation of numbers in exponential format. The symbol starts with a digit (\texttt{0-9}) or a period, and it can end with the sequence \texttt{E} or \texttt{e}, followed immediately by an optional sign (- or +), followed immediately by one or more digits (which cannot be followed by any other symbol characters). The character sequence to the left of the \texttt{"E"} or \texttt{"e"} must be a valid simple number, consisting only of digits or \texttt{\'\.'}. There must be at least one digit and at most one \texttt{\'\.'}. The sign in this context is part of the symbol and is not an operator.
These are valid numbers in exponential notation:

**Example 1.11. Valid exponential numbers**

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.3E-12</td>
<td></td>
</tr>
<tr>
<td>.03e+9</td>
<td></td>
</tr>
</tbody>
</table>

These are not valid numbers in exponential notation, but rather multiple tokens with an operator between:

**Example 1.12. Invalid exponential numbers**

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.E-12</td>
<td>-- no digits</td>
</tr>
<tr>
<td>3ae+6</td>
<td>-- non-digit character</td>
</tr>
<tr>
<td>3..0e+9</td>
<td>-- more than one '.'</td>
</tr>
</tbody>
</table>

### 1.10.4.5. Numbers

Numbers are character strings consisting of one or more decimal digits, with an optional prefix of a plus (+) or minus (-) sign, and optionally including a single period (.) that represents a decimal point. A number can also have a power of 10 suffixed in conventional exponential notation: an E (uppercase or lowercase), followed optionally by a plus or minus sign, then followed by one or more decimal digits defining the power of 10. Whenever a character string is used as a number, rounding can occur to a precision specified by the NUMERIC DIGITS instruction (the default is nine digits). See Chapter 10, *Numbers and Arithmetic* for a full definition of numbers.

Numbers can have leading whitespace (before and after the sign) and trailing whitespace. Whitespace characters cannot be embedded among the digits of a number or in the exponential part. Note that a symbol or a literal string can be a number. A number cannot be the name of a variable.

These are valid numbers:

**Example 1.13. Valid numbers**

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>&quot;-17.9&quot;</td>
<td></td>
</tr>
<tr>
<td>127.8650</td>
<td></td>
</tr>
<tr>
<td>73e+128</td>
<td></td>
</tr>
<tr>
<td>&quot; + 7.9E5 &quot;</td>
<td></td>
</tr>
</tbody>
</table>

You can specify numbers with or without quotation marks around them. Note that the sequence `-17.9` (without quotation marks) in an expression is not simply a number. It is a minus operator (which can be prefix minus if no term is to the left of it) followed by a positive number. The result of the operation is a number, which might be rounded or reformatted into exponential form depending on the size of the number and the current NUMERIC DIGITS setting.

A *whole number* is a number that has a no decimal part and that the language processor would not usually express in exponential notation. That is, it has no more digits before the decimal point than the current setting of NUMERIC DIGITS.

**Implementation maximum:** The exponent of a number expressed in exponential notation can have up to nine digits.
1.10.4.6. Operator Characters

The characters + - \ / % * | & = ¬ > < and the sequences >= <= \> <\ >\ <= >= indicate operations (see Section 1.11.2, “Operators”). A few of these are also used in parsing templates, and the equal sign and the sequences += -= *= /= %= //= ||= &= |= &&= **= are also used to indicate assignment. Whitespace characters and standard Rexx comments adjacent to operator characters and within operator character sequences are ignored. Therefore, the following are identical in meaning:

Example 1.14. White space and numbers

<table>
<thead>
<tr>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>345&gt;=123</td>
<td>345 &gt;= 123</td>
</tr>
<tr>
<td>345 &gt;=123</td>
<td>345 &gt;= 123</td>
</tr>
<tr>
<td>345 = 123</td>
<td>345 &gt;/&quot;not smaller&quot;/ = 123</td>
</tr>
</tbody>
</table>

Some of these characters (and some special characters—see the next section) might not be available in all character sets. In this case, appropriate translations can be used. In particular, the vertical bar (|) is often shown as a split vertical bar (¦).

Note

The Rexx interpreter uses ASCII character 124 in the concatenation operator and as the logical OR operator. Depending on the code page or keyboard for your particular country, ASCII 124 can be shown as a solid vertical bar (|) or a split vertical bar (¦). The character on the screen might not match the character engraved on the key. If you receive error 13, Invalid character in program, on an instruction including a vertical bar character, make sure this character is ASCII 124.

Throughout the language, the NOT (¬) character is synonymous with the backslash (\). You can use the two characters interchangeably according to availability and personal preference.

The Rexx interpreter recognizes both ASCII character 170 ('AA'x) and ASCII character 172 ('AC'x) for the logical NOT operator. Depending on your country, the ¬ might not appear on your keyboard. If the character is not available, you can use the backslash (\) in place of ¬.

1.10.4.7. Special Characters

The following characters, together with the operator characters, have special significance when found outside of literal strings:

| , ; : ( ) [ ] ~ |

These characters constitute the set of special characters. They all act as token delimiters, and whitespace characters (blank or horizontal tab) adjacent to any of these are removed. There is an exception: a whitespace character adjacent to the outside of a parenthesis or bracket is deleted only
if it is also adjacent to another special character (unless the character is a parenthesis or bracket and the whitespace character is outside it, too). For example, the language processor does not remove the blank in \texttt{A (Z)}. This is a concatenation that is not equivalent to \texttt{A(Z)}, a function call. The language processor removes the blanks in \texttt{(A) + (Z)} because this is equivalent to \texttt{(A)+(Z)}.

\textbf{1.10.4.8. Example}

The following example shows how a clause is composed of tokens:

\begin{example}

\begin{verbatim}
"REPEAT" A + 3;
\end{verbatim}

\end{example}

This example is composed of six tokens—a literal string ("\texttt{REPEAT}"), a blank operator, a symbol (\texttt{A}, which can have an assigned value), an operator (+), a second symbol (\texttt{3}, which is a number and a symbol), and the clause delimiter (;). The blanks between the \texttt{A} and the + and between the + and the \texttt{3} are removed. However, one of the blanks between the "\texttt{REPEAT}" and the \texttt{A} remains as an operator. Thus, this clause is treated as though written:

\begin{verbatim}
"REPEAT" A+3;
\end{verbatim}

\textbf{1.10.5. Implied Semicolons}

The last element in a clause is the semicolon (;) delimiter. The language processor implies the semicolon at a line end, after certain keywords, and after a colon if it follows a single symbol. This means that you need to include semicolons only when there is more than one clause on a line or to end an instruction whose last character is a comma.

A line end usually marks the end of a clause and, thus, Rexx implies a semicolon at most end of lines. However, there are the following exceptions:

\begin{itemize}
  \item The line ends in the middle of a multi-line (/* ... */) comment. The clause continues on to the next line.
  \item The last token was the continuation character (a comma or a minus sign) and the line does not end in the middle of a comment. (Note that a comment is not a token.)
\end{itemize}

Rexx automatically implies semicolons after colons (when following a single symbol or literal string, a label) and after certain keywords when they are in the correct context. The keywords that have this effect are ELSE, OTHERWISE, and THEN. These special cases reduce typographical errors significantly.

\textbf{Note}

The two characters forming the comment delimiters, /* and */, must not be split by a line end (that is, / and * should not appear on different lines) because they could not then be recognized correctly; an implied semicolon would be added.

\textbf{1.10.6. Continuations}
One way to continue a clause on the next line is to use the comma or the minus sign (-), which is referred to as the **continuation character**. The continuation character is functionally replaced by a blank, and, thus, no semicolon is implied. One or more comments can follow the continuation character before the end of the line.

The following example shows how to use the continuation character to continue a clause:

**Example 1.16. Continuations**

```
say "You can use a comma",       -- this line is continued
    "to continue this clause."
```

or

```
say "You can use a minus"-,       -- this line is continued
    "to continue this clause."
```

---

### 1.11. Terms, Expressions, and Operators

Expressions in Rexx are a general mechanism for combining one or more pieces of data in various ways to produce a result, usually different from the original data. All expressions evaluate to objects.

Everything in Rexx is an object. Rexx provides some objects, which are described in later sections. You can also define and create objects that are useful in particular applications—for example, a menu object for user interaction. See Section 1.3, “Modeling Objects” for more information.

#### 1.11.1. Terms and Expressions

**Terms** are literal strings, symbols, message terms and sequences, Array terms, Variable Reference terms, function calls, or subexpressions interspersed with zero or more operators that denote operations to be carried out on terms.

**Literal strings**, which are delimited by quotation marks, are constants.

**Symbols** (no quotation marks) are translated to uppercase. A symbol that does not begin with a digit or a period can be the name of a variable; in this case the value of that variable is used. A symbol that begins with a period can identify an object that the current environment provides; in this case, that object is used. Otherwise a symbol is treated as a constant string. A symbol can also be **compound**.

**Message terms** are described in Section 1.11.4, “Message Terms”; **Message sequences** are described in Section 1.11.5, “Message Sequences”.

**Array terms** are described in Section 1.11.6, “Array Term”.

**Variable Reference terms** are described in Section 1.11.7, “Variable Reference Term”.

**Function calls** (see Chapter 7, Functions), which are of the following form:

![Function Call Diagram]

The **symbolorstring** is a symbol or literal string.
An expression consists of one or more terms. A subexpression is a term in an expression surrounded with a left and a right parenthesis.

Evaluation of an expression is left to right, modified by parentheses and operator precedence in the usual algebraic manner (see Section 1.11.3, “Parentheses and Operator Precedence”). Expressions are wholly evaluated, unless an error occurs during evaluation.

As each term is used in an expression, it is evaluated as appropriate. The result is an object. Consequently, the result of evaluating any expression is itself an object (such as a character string).

1.11.2. Operators

An operator is a representation of an operation, such as an addition, to be carried out on one or two terms. Each operator, except for the prefix operators, acts on two terms, which can be symbols, strings, function calls, message terms, intermediate results, or subexpressions. Each prefix operator acts on the term or subexpression that follows it. Whitespace characters (and comments) adjacent to operator characters have no effect on the operator; thus, operators constructed from more than one character can have embedded whitespace and comments. In addition, one or more whitespace characters, if they occur in expressions but are not adjacent to another operator, also act as an operator. The language processor functionally translates operators into message terms. For dyadic operators, which operate on two terms, the language processor sends the operator as a message to the term on the left, passing the term on the right as an argument. For example, the sequence

```
say 1+2
```

is functionally equivalent to:

```
say 1~"+"(2)
```

The blank concatenation operator sends the message " " (a single blank), and the abuttal concatenation operator sends the "" message (a null string). When the ~ character is used in an operator, it is changed to a \. That is, the operators ~= and \= both send the message \= to the target object.

For an operator that works on a single term (for example, the prefix - and prefix + operators), Rexx sends a message to the term, with no arguments. This means -z has the same effect as z~"-".

See Section 5.1.4.2, “Comparison Methods” for comparison operator methods of the Object class and Section 5.1.7.18, “Arithmetic Methods” for arithmetic operator methods of the String class.

There are four types of operators:

- Concatenation
- Arithmetic
- Comparison
- Logical

1.11.2.1. String Concatenation
The concatenation operators combine two strings to form one string by appending the second string to the right-hand end of the first string. The concatenation may occur with or without an intervening blank. The concatenation operators are:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(blank)</td>
<td>Concatenate terms with one blank in between</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(abuttal)</td>
<td>Concatenate without an intervening blank</td>
</tr>
</tbody>
</table>

You can force concatenation without a blank by using the || operator.

The abuttal operator is assumed between two terms that are not separated by another operator. This can occur when two terms are syntactically distinct, such as a literal string and a symbol, or when they are only separated by a comment.

**Examples:**

An example of syntactically distinct terms is: if Fred has the value 37.4, then Fred"%" evaluates to 37.4%.

If the variable PETER has the value 1, then (Fred)(Peter) evaluates to 37.41.

The two adjoining strings, one hexadecimal and one literal, "4a 4b"x"LMN" evaluate to JKLNM.

In the case of

Fred/* The NOT operator precedes Peter. */¬Peter

there is no abuttal operator implied, and the expression is not valid. However,

(Fred)/" The NOT operator precedes Peter. */(¬Peter)

results in an abuttal, and evaluates to 37.40.

**1.11.2.2. Arithmetic**

You can combine character strings that are valid numbers (see Section 1.10.4.5, "Numbers") using the following arithmetic operators:

- **Add**
- **Subtract**
- **Multiply**
- **Divide**
- **Integer divide (divide and return the integer part of the result)**
- **Remainder (divide and return the remainder—not modulo, because the result can be negative)**
- **Power (raise a number to a whole-number power)**
- Prefix - Same as the subtraction: 0 - number
- Prefix + Same as the addition: 0 + number
See Chapter 10, Numbers and Arithmetic for details about precision, the format of valid numbers, and the operation rules for arithmetic. Note that if an arithmetic result is shown in exponential notation, it is likely that rounding has occurred.

### 1.11.2.3. Comparison

The comparison operators compare two terms and return the value 1 if the result of the comparison is true, or 0 otherwise.

The strict comparison operators all have one of the characters defining the operator doubled. The `==`, `\==`, and `\==` operators test for an exact match between two strings. The two strings must be identical (character by character) and of the same length to be considered strictly equal. Similarly, the strict comparison operators such as `>>` or `<<` carry out a simple character-by-character comparison, with no padding of either of the strings being compared. The comparison of the two strings is from left to right. If one string is shorter than the other and is a leading substring of another, then it is smaller than (less than) the other. The strict comparison operators also do not attempt to perform a numeric comparison on the two operands.

For all other comparison operators, if both terms involved are numeric, a numeric comparison (see Section 10.4, "Numeric Comparisons") is effected. Otherwise, both terms are treated as character strings, leading and trailing whitespace characters are ignored, and the shorter string is padded with blanks on the right.

Character comparison and strict comparison operations are both case-sensitive, and the exact collating order might depend on the character set used for the implementation. In an ASCII environment, such as Windows and Unix-like systems, the ASCII character value of digits is lower than that of the alphabetic characters, and that of lowercase alphabetic characters is higher than that of uppercase alphabetic characters.

The comparison operators and operations are:

- `=`  True if the terms are equal (numerically or when padded)
- `\=`, `\==`, `\==` True if the terms are not equal (inverse of `=`)
- `>`  Greater than
- `<`  Less than
- `><` Greater than or less than (same as not equal)
- `<>` Greater than or less than (same as not equal)
- `>=` Greater than or equal to
- `\<`, `\<<` Not less than
- `<=` Less than or equal to
- `\>`, `\>>` Not greater than
- `==` True if terms are strictly equal (identical)
- `\==`, `\==` True if the terms are not strictly equal (inverse of `==`)
- `>>` Strictly greater than
- `<<` Strictly less than
- `\>=`, `\>=` Strictly greater than or equal to
- `\<\<<`, `\排除` Strictly not less than
- `\<=\<=` Strictly less than or equal to
- `\>>\>>` Strictly not greater than
Parentheses and Operator Precedence

Note

Throughout the language, the NOT (¬) character is synonymous with the backslash(\). You can use the two characters interchangeably, according to availability and personal preference. The backslash can appear in the following operators: \ (prefix not), \=, \==, \<, \>, \<<, and \>>.

1.11.2.4. Logical (Boolean)

A character string has the value false if it is 0, and true if it is 1. The logical operators take one or two such values and return 0 or 1 as appropriate. Values other than 0 or 1 are not permitted.

&       AND — returns 1 if both terms are true.
|       Inclusive OR — returns 1 if either term or both terms are true.
&&      Exclusive OR — returns 1 if either term, but not both terms, is true.
Prefix \, ¬   Logical NOT— negates; 1 becomes 0, and 0 becomes 1.

1.11.3. Parentheses and Operator Precedence

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

• When parentheses are encountered—other than those that identify the arguments on messages (see Section 1.11.4, “Message Terms”) and function calls—the entire subexpression between the parentheses is evaluated immediately when the term is required.

• When the sequence

```
  term1 operator1 term2 operator2 term3
```

is encountered, and operator2 has precedence over operator1, the subexpression (term2 operator2 term3) is evaluated first.

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

For example, * (multiply) has a higher priority than + (add), so 3+2*5 evaluates to 13 (rather than the 25 that would result if a strict left-to-right evaluation occurred). To force the addition to occur before the multiplication, you could rewrite the expression as (3+2)*5. Adding the parentheses makes the first three tokens a subexpression. Similarly, the expression -3**2 evaluates to 9 (instead of -9) because the prefix minus operator has a higher priority than the power operator.

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

```
~   ~~          (message send)
+   -   \        (prefix operators)
**          (power)
*   /   %   //   (multiply and divide)
+   -        (add and subtract)
```
Parentheses and Operator Precedence

(±blank) || (abuttal) (concatenation with or without blank)

= > < (comparison operators, all with equal precedence)

<= = >=

॥ = <= =>

> =>

< =>

< <=

< == =>

<< = <<

>>& && (and)

|| & & (or, exclusive or)

Suppose the symbol A is a variable whose value is 3, DAY is a variable whose value is Monday, and other variables are uninitialized. Then:

Example 1.17. Arithmetic

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + 5</td>
<td>&quot;8&quot;</td>
</tr>
<tr>
<td>A - 4 * 2</td>
<td>&quot;-5&quot;</td>
</tr>
<tr>
<td>A / 2</td>
<td>&quot;1.5&quot;</td>
</tr>
<tr>
<td>0.5 ** 2</td>
<td>&quot;0.25&quot;</td>
</tr>
</tbody>
</table>
| (A + 1) > 7 | "0"    | /* that is, .false */
| " " = "="  | "1"    | /* that is, .true */
| " \= ""    | "0"    | /* that is, .false */
| " & " "="  | "1"    | /* that is, .true */

Today is Day  | "TODAY IS Monday"
"If it is" day | "If it is Monday"
Substr(Day, 2, 3) | "ond" /* Substr is a function */
"!" xxx"!"     | "!XXX!"
Note

The Rexx order of precedence usually causes no difficulty because it is the same as in
conventional algebra and other computer languages. There are two differences from common
notations:
• The prefix minus operator always has a higher priority than the power operator.
• Power operators (like other operators) are evaluated from left to right.

For example:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3**2</td>
<td><strong>9</strong></td>
<td>/* not -9 */</td>
</tr>
<tr>
<td>-(2+1)**2</td>
<td>9</td>
<td>/* not -9 */</td>
</tr>
<tr>
<td>2<strong>2</strong>3</td>
<td>64</td>
<td>/* not 256 */</td>
</tr>
</tbody>
</table>

1.11.4. Message Terms

You can include messages to objects in an expression wherever a term, such as a literal string, is
valid. A message can be sent to an object to perform an action, obtain a result, or both.

A message term can have one of the following forms:

```
receiver ~ messagename (expression, symbol) [expression]
```

The receiver is a term (see Section 1.11.1, “Terms and Expressions” for a definition of term). It
receives the message. The ~ or ~~ indicates sending a message. The messagename is a literal
string or a symbol that is taken as a constant. The expressions (separated by commas) between
the parentheses or brackets are the arguments for the message. The receiver and the argument
expressions can themselves include message terms. If the message has no arguments, you can omit
the parentheses.

The left parenthesis, if present, must immediately follow a token (messagename or symbol) with no
blank in between them. Otherwise, only the first part of the construct is recognized as a message term.
(A blank operator would be assumed at that point.) Only a comment (which has no effect) can appear
between a token and the left parenthesis.

You can use any number of expressions, separated by commas. The expressions are evaluated from
left to right and form the arguments during the execution of the called method. Any ARG, PARSE
ARG, or USE ARG instruction or ARG() built-in function in the called method accesses these objects
while the called method is running. You can omit expressions, if appropriate, by including extra
commas.

The receiver object is evaluated, followed by one or more expression arguments. The message name
(in uppercase) and the resulting argument objects are then sent to the receiver object. The receiver
object selects a method to be run based on the message name, and runs the selected method with the specified argument objects. The receiver eventually returns, allowing processing to continue.

If the message term uses ~, the receiver method must return a result object. This object is included in the original expression as if the entire message term had been replaced by the name of a variable whose value is the returned object.

For example, the message POS is valid for strings, and you could code:

```clike
c="escape"
a="Position of 'e' is:" c~pos("e",3)
/* would set A to "Position of 'e' is: 6" */
```

If the message term uses ~~, the receiver method need not return a result object. Any result object is discarded, and the receiver object is included in the original expression in place of the message term.

For example, the messages INHERIT and SUBCLASS are valid for classes (see Section 5.1.1, “Class Class (Metaclass)”) and, assuming the existence of the Persistent class, you could code:

```clike
account = .object~subclass("Account")~~inherit(.persistent)
/* would set ACCOUNT to the object returned by SUBCLASS, */
/* after sending that object the message INHERIT */
```

If the message term uses brackets, the message [] is sent to the receiver object. (The expressions within the brackets are available to the receiver object as arguments.) The effect is the same as for the corresponding ~ form of the message term. Thus, `a[b]` is the same as `a~"[\]()"(b)`.

For example, the message [] is valid for arrays (see Section 5.3.6, “Array Class”) and you could code:

```clike
Example 1.18. Arrays
```

```clike
a = .array-of(10,20)
say "Second item is" a[2]  /* Same as: a-at(2) */
/* or a~"[\]()"(2) */
/* Produces: "Second item is 20" */
```

A message can have a variable number of arguments. You need to specify only those required. For example, "ESCAPE"~POS("E") returns 1.

A colon (:) and symbol can follow the message name. In this case, the symbol must be the name of a variable (usually the special variable `SUPER`) or an environment symbol. The resulting value changes the usual method selection. For more information, see Section 4.2.7, “Changing the Search Order for Methods”.

### 1.11.5. Message Sequences

The ~ and ~~~ forms of message terms differ only in their treatment of the result object. Using ~ returns the result of the method. Using ~~~ returns the object that received the message. Here is an example:

```clike
Example 1.19. Messages
```

```clike
/* Two ways to use the INSERT method to add items to a list */
```
Thus, you would use \texttt{~} when you want the returned result to be the receiver of the next message in the sequence.

### 1.11.6. Array Term

As a convenience, Rexx allows a shortened syntax for creating arrays, called \textit{Array term}, of the following form:

\begin{center}
\begin{tikzpicture}
  \node [input, align=left, font=ootnotesize] (term) {expression, expression};
  \node [output, font=ootnotesize] at (term.east) (output) {expression};
  \draw [arrow] (term) -- (output);
\end{tikzpicture}
\end{center}

Except for trailing commas, an Array term returns a newly created array, as if it had been created with the message term \texttt{rexx: Array-term(...)}. (An Array term will always return an instance of the Rexx-defined \textit{Array} class, even if a user-defined class named \textit{Array} exists.)

Here are some examples:

\begin{verbatim}
week = "mon", "tue", "wed", "thu", "fri", "sat", "sun"
say week~items "days" -- 7 days
say ("here", "we", "go")~makeString(, " ") -- here we go
do list over .environment, .local
  say list~items -- 65
end -- 10
say 0~sendWith("MAX", (2, 3, 5, 7, 11, 13)) -- 13
sparse = ,,0
say "size" sparse~size, "items" sparse~items -- size 4, 1 items
\end{verbatim}

If the array term has trailing commas, the returned array has a bigger size than what \texttt{.Array-term(...)} would have returned:

\begin{verbatim}
say (1, , 3, ,)-size .Array-of(1, , 3, ,)-size -- 5 3
say (,)-size .Array-of(,)-size -- 2 0
\end{verbatim}

In a context, where commas already have a different meaning, it may be necessary to put an Array term between brackets. For example:
A variable reference term represents a reference to a variable. Variable reference terms can be used as arguments to subroutines, functions, methods, or routines, thus allowing the original variable to be modified or created by the called Rexx code.

While subroutines, functions, methods, or routines can modify argument objects when accessed with the USE ARG instruction without using a variable reference, changing the value of arguments to new objects—like setting them to a new string, a new Array, or to .nil—can only be done using a variable reference term.

Variable reference terms start with either of the two reference operators, > or <, followed by a simple variable name or a stem variable name. Variable references to compound variables are not allowed.

Here is an example:

```
call dir >files, ":.txt"
say files~items "files with extension .txt"
::routine dir
   use strict arg >array, extension = ""
   array = .Array~new
   do file over .File~new(".")~list
      if file~caselessEndsWith(extension) then
         array~append(file)
   end
```

Variable reference arguments and USE ARG names must match. They must either be both simple variable references, or both stem variable names. USE ARG variable references can never be optional, a default value is not allowed.

### 1.12. Clauses and Instructions

Clauses can be subdivided into the following types:

- Null Clauses,
- Directives,
- Labels,
- Instructions,
- Assignments,
- Message Instructions,
- Keyword Instructions, and
- Commands.

#### 1.12.1. Null Clauses
A clause consisting only of whitespace characters, comments, or both is a null clause. It is completely ignored.

**Note**

A null clause is not an instruction; for example, putting an extra semicolon after the THEN or ELSE in an IF instruction is not equivalent to using a dummy instruction (as it would be in the C language). The NOP instruction is provided for this purpose.

### 1.12.2. Directives

A clause that begins with two colons is a directive. Directives are nonexecutable code and can start in any column. They divide a program into separate executable units (methods and routines) and supply information about the program or its executable units. Directives perform various functions, such as creating new Rexx classes (::CLASS directive) or defining a method (::METHOD directive). See Chapter 3, Directives for more information about directives.

### 1.12.3. Labels

A clause that consists of a single symbol or string followed by a colon is a label. The colon in this context implies a semicolon (clause separator), so no semicolon is required.

The label's name is taken from the string or symbol part of the label. If the label uses a symbol for the name, the label's name is in uppercase. If a label uses a string, the name can contain mixed-case characters.

Labels identify the targets of CALL instructions, SIGNAL instructions, and internal function calls. Label searches for CALL, SIGNAL, and internal function calls are case-sensitive. Label-search targets specified as symbols cannot match labels with lowercase characters. Literal-string or computed-label searches can locate labels with lowercase characters.

Labels can be any number of successive clauses. Several labels can precede other clauses. Labels are treated as null clauses and can be traced selectively to aid debugging.

Duplicate labels are permitted, but control is only passed to the first of any duplicates in the main program (prolog), a method, or a routine. The duplicate labels occurring later can be traced but cannot be used as a target of a CALL, SIGNAL, or function invocation.

### 1.12.4. Instructions

An instruction consists of one or more clauses describing some course of action for the language processor to take. Instructions can be assignments, message instructions, keyword instructions, or commands.

### 1.12.5. Assignments

A single clause of the form `symbol=expression` is an instruction known as an assignment. An assignment gives a (new) value to a variable. See Section 1.13, “Assignments and Symbols”.
1.12.5.1. Extended Assignments

The character sequences `+=`, `-=` , `*=` , `/=` , `%=`, `//=`, `||=`, `&&=`, and `**=` can be used to create extended assignments. These sequences combine an operation with the assignment. See Section 1.13.1, “Extended Assignments” for more details.

1.12.6. Message Instructions

A message instruction is a single clause in the form of a message term (see Section 1.11.4, “Message Terms”) or in the form `messageterm=expression`. A message is sent to an object, which responds by performing some action. See Section 1.15, “Message Instructions”.

1.12.7. Keyword Instructions

A keyword instruction is one or more clauses, the first of which starts with a keyword that identifies the instruction. Keyword instructions control, for example, the external interfaces and the flow of control. Some keyword instructions can include nested instructions. In the following example, the DO construct (DO, the group of instructions that follow it, and its associated END keyword) is considered a single keyword instruction.

```plaintext
DO
  instruction
  instruction
  instruction
END
```

A subkeyword is a keyword that is reserved within the context of a particular instruction, for example, the symbols TO and WHILE in the DO instruction.

1.12.8. Commands

A command is a clause consisting of an expression only. The expression is evaluated and the result is passed as a command string to an external environment.

1.13. Assignments and Symbols

A variable is an object whose value can change during the running of a Rexx program. The process of changing the value of a variable is called assigning a new value to it. The value of a variable is a single object. Note that an object can be composed of other objects, such as an array or directory object.

You can assign a new value to a variable with the ARG, PARSE, PULL, or USE instructions, or the VALUE built-in function, but the most common way of changing the value of a variable is the assignment instruction itself. Any clause in the form

```
symbol= expression;
```

is taken to be an assignment. The result of `expression` becomes the new value of the variable named by the symbol to the left of the equal sign.
Example:

/* Next line gives FRED the value "Frederic" */
Fred="Frederic"

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

You can use a symbol in an expression even if you have not assigned a value to it, because a symbol has a defined value at all times. A variable to which you have not assigned a value is uninitialized. Its value is the characters of the symbol itself, translated to uppercase (that is, lowercase a-z to uppercase A-Z). However, if it is a compound symbol (described in Section 1.13.5, “Compound Symbols”), its value is the derived name of the symbol.

Example 1.20. Derived symbol names

/* If Freda has not yet been assigned a value, */
/* then next line gives FRED the value "FREDA" */
Fred=Freda

The meaning of a symbol in Rexx varies according to its context. As a term in an expression, a symbol belongs to one of the following groups: constant symbols, simple symbols, compound symbols, environment symbols, and stems. Constant symbols cannot be assigned new values. You can use simple symbols for variables where the name corresponds to a single value. You can use compound symbols and stems for more complex collections of variables although the collection classes might be preferable in many cases. See Section 5.3.2, “Collection Class”.

Notes:

1. When the ARG, PARSE, PULL, or USE instruction, the VALUE built-in function, or the variable pool interface changes a variable, the effect is identical to an assignment.

2. Any clause that starts with a symbol and whose second token is (or starts with) an equal sign (=) is an assignment, rather than an expression (or a keyword instruction). This is not a restriction, because you can ensure that the clause is processed as a command, such as by putting a null string before the first name, or by enclosing the expression in parentheses.

If you unintentionally use a Rexx keyword as the variable name in an assignment, this should not cause confusion. For example, the following clause is an assignment, not an ADDRESS instruction:

```rexx
Address="10 Downing Street";
```

3. You can use the VAR function to test whether a symbol has been assigned a value. In addition, you can set NOVALUE to trap the use of any uninitialized variables (except when they are tails in compound variables or stem variables, which are always initialized with a Stem object when first used.)

1.13.1. Extended Assignments

The character sequences +=, -=, *=, /=, %=, //=, ||=, &=, |=, &&=, and **= can be used to create extended assignment instructions. An extended assignment combines a non-prefix operator with an assignment where the term on the left side of the assignment is also used as the left term of the operator. For example,
Constant Symbols

a += 1

is exactly equivalent to the instruction

\[ a = a + 1 \]

Extended assignments are processed identically to the longer form of the instruction.

### 1.13.2. Constant Symbols

A *constant symbol* starts with a digit (0-9) or a period.

You cannot change the value of a constant symbol. It is simply the string consisting of the characters of the symbol (that is, with any lowercase alphabetic characters translated to uppercase).

These are constant symbols:

**Example 1.21. Constants**

```
77
827.53
.12345
12e5 /* Same as 12E5 */
30
17E-3
```

Constant symbols, where the first character is a period, which have at least one other character, and which are not a valid Rexx number, are *environment symbols* and may have a value other than the symbol name.

### 1.13.3. Simple Symbols

A *simple symbol* does not contain any periods and does not start with a digit (0-9).

By default, its value is the characters of the symbol (that is, translated to uppercase). If the symbol has been assigned a value, it names a variable and its value is the value of that variable.

These are simple symbols:

**Example 1.22. Simple symbols**

```
FRED
Whatagoodidea? /* Same as WHATAGOODIDEA? */
?12
```

### 1.13.4. Stems

A *stem* is a symbol that contains a single period as the last character of the name. It cannot start with a digit.

These are stems:
The value of a stem is always a Stem object (see Section 5.3.16, “Stem Class” for details). The stem variable’s Stem object is automatically created the first time you use the stem variable or a compound variable containing the stem variable name. The Stem object’s assigned name is the name of the stem variable (with the characters translated to uppercase). If the stem variable has been assigned a value, or the Stem object has been given a default value, the assigned name overrides the default stem name. A reference to a stem variable will return the associated Stem object.

When a stem is the target of an assignment, the action taken depends on the value being assigned. If the new value is a Stem object, the new Stem object will replace the Stem object that is currently associated with the stem variable. This can result in multiple stem variables referring to the same Stem object, effectively creating a variable alias.

If the new value is not a Stem object, a new Stem object is created and assigned to the stem variable, replacing the Stem object currently associated with the stem variable.

The new value assigned to the stem variable is given to the new Stem object as a default value. Following the assignment, a reference to any compound symbol with that stem variable returns the new value until another value is assigned to the stem, the Stem object, or the individual compound variable.

Thus, you can initialize an entire collection of compound variables to the same value.

You can pass stem collections as function, subroutine, or method arguments.
Stems

Randomize: Use Arg count, stem.
do i = 1 to count
  stem.i = random(1,100)
end
return

The USE ARG instruction functions as an assignment instruction. The variable STEM. in the example above is functionally equivalent to:

stem. = arg(2)

Note

USE ARG must be used to access the stem variable as a collection. PARSE and PARSE ARG will force the stem to be a string value.

Stems can also be returned as function, subroutine, or method results. The resulting return value is the Stem object associated with the stem variable.

Example 1.27. Stems

/* RANDOMIZE(count) calls routine */
Randomize: Use Arg count
do i = 1 to count
  stem.i = random(1,100)
end
return stem.

When a stem. variable is used in an expression context, the stem variable reference returns the associated Stem object. The Stem object will forward many object messages to its default value. For example, the STRING method will return the Stem object's default value's string representation:

Example 1.28. Stems

total. = 0
say total. /* says "0" */

The [] method with no arguments will return the currently associated default value. variables can always be obtained by using the stem. However, this is not the same as using a compound variable whose derived name is the null string.

Example 1.29. Stems

total. = 0
null = ""
total.null = total.null + 5
say total.[] total.null /* says "0 5" */
You can use the *DROP*, *EXPOSE*, and *PROCEDURE* instructions to manipulate collections of variables, referred to by their stems. *DROP FRED* assigns a new Stem object to the specified stem. *EXPOSE FRED* and *PROCEDURE EXPOSE FRED* expose all possible variables with that stem.

The *DO* instruction can also iterate over all of the values assigned to a stem variable.

### 1.13.5. Compound Symbols

A *compound symbol* contains at least one period and two other characters. It cannot start with a digit or a period, and if there is only one period it cannot be the last character.

The name begins with a stem (that part of the symbol up to and including the first period) and is followed by a tail consisting of one or more name parts (delimited by periods) that are constant symbols, simple symbols, or null. Note that you cannot use constant symbols with embedded signs (for example, 12.3E+5) after a stem; in this case the whole symbol would not be valid.

These are compound symbols:

**Example 1.30. Compound symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRED.3</td>
</tr>
<tr>
<td>Array.I.J</td>
</tr>
<tr>
<td>AMESSY..One.2.</td>
</tr>
</tbody>
</table>

Before the symbol is used, that is, at the time of reference, the language processor substitutes in the compound symbol the character string values of any simple symbols in the tail (I, J, and One in the examples), thus generating a new, derived tail. The value of a compound symbol is, by default, its the name of the Stem object associated with the stem variable concatenated to the derived tail or, if it has been used as the target of an assignment, the value of Stem element named by the derived tail.

The substitution in the symbol permits arbitrary indexing (subscripting) of collections of variables that have a common stem. Note that the values substituted can contain any characters (including periods and blanks). Substitution is done only once.

More formally, the derived name of a compound variable that is referenced by the symbol

\[ s_0.s_1.s_2. \ldots .s_n \]

is given by

\[ d_0.v_1.v_2. \ldots .v_n \]

where \(d_0\) is the name of the Stem object associated with the stem variable \(s_0\) and \(v_1\) to \(v_n\) are the values of the constant or simple symbols \(s_1\) through \(s_n\). Any of the symbols \(s_1\) to \(s_n\) can be null. The values \(v_1\) to \(v_n\) can also be null and can contain any characters (including periods). Lowercase characters are not translated to uppercase, blanks are not removed, and periods have no special significance. There is no limit on the length of the evaluated name.

Some examples of simple and compound symbols follow in the form of a small extract from a Rexx program:

**Example 1.31. Compound symbols**

```rexx
a=3       /* assigns "3" to the variable A */
```
You can use compound symbols to set up arrays and lists of variables in which the subscript is not necessarily numeric, thus offering a great scope for the creative programmer. A useful application is to set up an array in which the subscripts are taken from the value of one or more variables, producing a form of associative memory (content-addressable).

1.13.5.1. Evaluated Compound Variables
The value of a stem variable is always a Stem object (see Section 5.3.16, “Stem Class” for details). A Stem object is a type of collection that supports the [] and [[]] methods used by other collection classes. The [] method provides an alternate means of accessing compound variables that also allows embedded subexpressions.

Tails for compound variables are normally specified by symbols separated by periods. An alternative is to specify the tail as a bracketed list of expressions separated by commas. The expressions are evaluated to character strings. These are concatenated with intervening periods and the resulting string is used as tail. This notation can be used in assignments to compound variables as well as when referencing them. Examples:

Example 1.32. Evaluated compound variables

```
Example 1.32. Evaluated compound variables
```

```
```

1.13.6. Environment Symbols
An environment symbol starts with a period and has at least one other character. The symbol may not be a valid Rexx number. By default the value of an environment symbol is the string consisting of the characters of the symbol (translated to uppercase). If the symbol identifies an object in the current environment, its value is the mapped object.

These are environment symbols:

Example 1.33. Environment symbols

```
Example 1.33. Environment symbols
```

```
```
When you use an environment symbol, the language processor performs a series of searches to see if the environment symbol has an assigned value. The search locations and their ordering are:

2. The list of classes declared on `::CLASS directives` within the current program package or added to the current package using the `addClass` method.
3. The list of public classes declared on `::CLASS directives` of other files included with a `::REQUIRES directive` or added to the current Package instance using the `addPackage` method.
4. The list of public Rexx supplied classes in the REXX package, like `Object`, `String`, or `Array`.
5. The package local environment directory specific to the current package. You can access the package local environment directory through the `Package local` method.
6. The local environment directory specific to the current interpreter instance. The local environment includes instance-specific objects such as the `.INPUT` and `.OUTPUT` objects. You can directly access the local environment directory by using the `.LOCAL environment symbol`.
7. The global environment directory. The global environment includes Rexx supplied objects like `.endofline` or the `.RexxInfo object`. You can directly access the global environment by using the `.ENVIRONMENT environment symbol`. Entries in the global environment directory can also be accessed via the `VALUE built-in function` by using a null string for the `selector` argument.
8. Rexx defined symbols. Other simple environment symbols are reserved for use by Rexx built-in environment objects. The currently defined built-in objects are `.RS`, `.LINE`, `.METHODS`, `.ROUTINES`, `.RESOURCES`, and `.CONTEXT`.

If an entry is not found for an environment symbol, then the default character string value is used.

### Note

You can place entries in both the `.LOCAL` and the `.ENVIRONMENT directories for programs to use. To avoid conflicts with future Rexx defined entries, it is recommended that the entries that you place in either directory include at least one period in the entry name, for example:

```rexx
/* establish settings directory */
.local-setentry("MyProgram.settings", .directory~new)
```

### 1.14. Namespaces

Namespaces can be used to differentiate between classes or routines of the same name, having been loaded through `::REQUIRES directives`.

Using `::REQUIRES` with the `NAMESPACE` option allows to tag a loaded file with a namespace name, which in turn can be used to qualify references to classes or routines within that namespace to explicitly identify a specific class or routine. A special reserved namespace "REXX" will allow to always access the Rexx-provided classes.
1.15. Message Instructions

You can send a message to an object to perform an action, obtain a result, or both. You use a message instruction if the main purpose of the message is to perform an action. You use a message term if the main purpose of the message is to obtain a result.

A message instruction is a clause of the form:

```
message term = expression
```

If there is only a message term, the message is sent in exactly the same way as for a message term. If the message yields a result object, it is assigned to the sender's special variable RESULT. If you use the ~~ form of message term, the receiver object is used as the result. If there is no result object, the variable RESULT is dropped (becomes uninitialized). A message term using ~~ is sometimes referred to as a cascading message.
Example 1.36. Message instructions

```
mytable~add("John", 123)
```

This sends the message ADD to the object MYTABLE. The ADD method need not return a result. If ADD returns a result, the result is assigned to the variable RESULT.

The equal sign (=) sets a value. If `=expression` follows the message term, a message is sent to the receiver object with an `=` concatenated to the end of the message name. The result of evaluating the expression is passed as the first argument of the message.

Example 1.37. Message instructions

```
person-age = 39   /* Same as person~"AGE"=(39) */
table[i] = 5      /* Same as table~"[i]"=(5,i) */
```

The expressions are evaluated in the order in which the arguments are passed to the method. That is, the language processor evaluates the `=expression` first. Then it evaluates the argument expressions within any `[]` pairs from left to right.

The extended assignment form may also be used with message terms.

Example 1.38. Message instructions

```
table[i] += 1        -- Same as table[i] = table[i] + 1
```

See Section 1.13.1, “Extended Assignments” for more details.

1.16. Commands to External Environments

Issuing commands to the surrounding environment is an integral part of Rexx.

1.16.1. Environment

The base system for the language processor is assumed to include at least one environment for processing commands. An environment is selected by default on entry to a Rexx program. You can change the environment by using the `ADDRESS` instruction. You can find out the name of the current environment by using the `ADDRESS` built-in function. The underlying operating system defines environments external to the Rexx program. The environments selected depend on the caller. The default environment is `CMD` on Windows and `sh` on Unix-like systems. There are three aliases for the default environment, "" (null string), `COMMAND` and `SYSTEM`. These environments execute commands through the standard command interpreter or system shell, which is `cmd.exe` on Windows and `sh` on Unix-like systems. On Unix-like systems alternate environments `bsh`, `bash`, `csh`, `ksh`, `tcsh`, and `zsh` are available, which allow execution of commands through a shell named like the environment, if such a shell is installed on the system.

An additional environment `PATH`—available on both Windows and Unix-like systems—provides an alternative to command interpreter-based shell-based environments. It executes commands directly, without the need for a command interpreter or shell by searching the environment variable `PATH`
to locate the command. Note that this environment does not provide shell or command interpreter features such as internal commands, redirection, piping, environment variable substitution or wildcard expansion.

If called from an editor that accepts subcommands from the language processor, the default environment can be that editor.

A Rexx program can issue commands—called subcommands—to other application programs. For example, a Rexx program written for a text editor can inspect a file being edited, issue subcommands to make changes, test return codes to check that the subcommands have been processed as expected, and display messages to the user when appropriate.

An application that uses Rexx as a macro language must register its environment with the Rexx language processor. See the Open Object Rexx: Application Programming Interfaces for a discussion of this mechanism.

### 1.16.2. Commands

To send a command to the currently addressed environment, use a clause of the form:

```
expression;
```

The expression (which must not be an expression that forms a valid message instruction) is evaluated, resulting in a character string value (which can be the null string), which is then prepared as appropriate and submitted to the environment specified by the current ADDRESS setting.

The environment then processes the command and returns control to the language processor after setting a return code. A return code is a string, typically a number, that returns some information about the command processed. A return code usually indicates if a command was successful but can also represent other information. The language processor places this return code in the Rexx special variable RC.

In addition to setting a return code, the underlying system can also indicate to the language processor if an error or failure occurred. An error is a condition raised by a command to which a program that uses that command can respond. For example, a locate command to an editing system might report requested string not found as an error. A failure is a condition raised by a command to which a program that uses that command cannot respond, for example, a command that is not executable or cannot be found.

Errors and failures in commands can affect Rexx processing if a condition trap for ERROR or FAILURE is ON (see Chapter 11, Conditions and Condition Traps). They can also cause the command to be traced if TRACE E or TRACE F is set. TRACE Normal is the same as TRACE F and is the default—see Section 2.29, “TRACE”.

The .RS environment symbol can also be used to detect command failures and errors. When the command environment indicates that a command failure has occurred, the Rexx environment symbol .RS has the value -1. When a command error occurs, .RS has a value of 1. If the command did not have a FAILURE or ERROR condition, .RS is 0.

Here is an example of submitting a command. Where the default environment is Windows, the sequence:

```
Example 1.39. Commands
```

```
fname = "CHESHIRE"
```
would result in passing the string **TYPE CHESHIRE.CAT** to the command processor, CMD.EXE. The simpler expression:

**Example 1.40. Commands**

```
"TYPE CHESHIRE.CAT"
```

has the same effect.

On return, the return code placed in RC will have the value 0 if the file CHESHIRE.CAT were typed, or a nonzero value if the file could not be found in the current directory.

**Note**

Remember that the expression is evaluated before it is passed to the environment. Constant portions of the command should be specified as literal strings.

**Example 1.41. Commands — Windows**

```c
delete "**".lst           /* not "multiplied by" */
var.003 = anyvalue
     type "var.003"     /* not a compound symbol */
     w = any
     dir"/w"            /* not "divided by ANY" */
```

**Example 1.42. Commands — Linux**

```c
rm "**".lst               /* not "multiplied by" */
var.003 = anyvalue
     cat "var.003"       /* not a compound symbol */
     w = any
     ls "/w"             /* not "divided by ANY" */
```

Enclosing an entire message instruction in parentheses causes the message result to be used as a command. Any clause that is a message instruction is not treated as a command. Thus, for example, the clause

```c
myfile~linein
```

causes the returned line to be assigned to the variable RESULT, not to be used as a command to an external environment, while

```c
(myfile~linein)
```
Using Rexx on Windows and Unix-like systems

1.17. Using Rexx on Windows and Unix-like systems

1.17.1. Calling other Rexx scripts

Rexx programs can call other Rexx programs as external functions or subroutines with the `CALL` instruction.

If a program is called with the `CALL` instruction, the program runs in the same process as the calling program. If you call another program by a Rexx command, the program is executed in a new process and therefore does not share .environment, .local, or the Windows or Unix-like systems shell environment.

**Example 1.43. Calling other Rexx scripts**

```
call "other.REX"         /* runs in the same process */
"rexx other.REX"         /* runs in a new child process */
"start rexx other.REX"   /* runs in a new detached process */
```

When Rexx programs call other Rexx programs as commands, the return code of the command is the exit value of the called program provided that this value is a whole number in the range -32768 to 32767. Otherwise, the exit value is ignored and the called program is given a return code of 0.

1.17.2. Shebang support

Rexx supports shebangs, also called hashbangs, as the first line of a Rexx program on both Unix-like and Windows systems.

Typically, shebangs are found on Unix-like systems to identify the script language with which to run a script. On these systems, to run a script as a Rexx program, depending on where the interpreter is installed, the shebang would be something like `#!/usr/bin/rexx`.

With several Unix-like systems changing the standard installation location away from `/usr/bin` (including macOS), a new convention is used for the sample programs, which works if the Rexx interpreter is anywhere on the path. This convention uses the `env` command in the shebang, like this: `#!/usr/bin/env rexx`. This has the advantage that it is portable, and will use the first occurrence of `rexx` that is found on `$PATH`.

On a Windows system, Rexx will simply ignore the first line of a Rexx program, if it starts with the character sequence "#!".

1.17.3. Line-end characters

Windows and Unix-like systems use different characters to indicate a line-break (ending one line and starting a new one) in a text file. While Windows uses the two-character sequence `'0d 0a'x` (carriage-return, line-feed), Unix-like systems use the character `'0a'x` (line-feed) as a line-end indicator.

Rexx supports both line-end indicators on each platform, both for the Rexx program itself, and for data manipulated with e.g. built-in functions `LINES`, `LINEIN`, `Stream` methods `lines`, `lineln`, `arrayIn`,
String method `makeArray`, or `MutableBuffer` method `makeArray`. This means, when moving from one platform to another, typically there should be no line-end conversions necessary for Rexx programs or data read by these programs.

Note that you cannot include line-end characters '\0d\x' or '\0a\x' in a literal string.

1.17.4. End-of-file character
Traditionally Windows used a special character '\1a\x' (end-of-file) to signify the end of a text file. Although today use of this special end-of-file character has become less common, some editors may still append it to the end of a text file. Unix-like systems do not use a special end-of-file character in text files.

Rexx will honor the end-of-file character '\1a\x' within a Rexx program source file on both Windows and Unix-like platforms. It will not scan the source file beyond any end-of-file character it finds. This means, that a character '\1a\x' cannot be directly used within a Rexx program source file, for example within a literal string or a comment.
Keyword Instructions

A keyword instruction is one or more clauses, the first of which starts with a keyword that identifies the instruction. Some keyword instructions affect the flow of control, while others provide services to the programmer. Some keyword instructions, like DO, can include nested instructions.

In the syntax diagrams on the following pages, symbols (words) in capitals denote keywords or subkeywords. Other words, such as expression, denote a collection of tokens as defined previously. Note, however, that the keywords and subkeywords are not case-dependent. The symbols if, If, and iF all have the same effect. A clause requires a semicolon (;) as a terminating delimiter unless the end of a line implies it.

A keyword instruction is recognized only if its keyword is the first token in a clause and if the second token does not start with an equal (=) character (implying an assignment) or a colon (implying a label). The keywords ELSE, END, OTHERWISE, THEN, and WHEN are treated in the same way. Note that any clause that starts with a keyword defined by Rexx cannot be a command. Therefore,

```
arg(fred) rest
```

is an ARG keyword instruction, not a command that starts with a call to the ARG built-in function. A syntax error results if the keywords are not in their correct positions in a DO, IF, or SELECT instruction. The keyword THEN is also recognized in the body of an IF or WHEN clause. In other contexts, keywords are not reserved and can be used as labels or as the names of variables (though this is generally not recommended).

Subkeywords are reserved within the clauses of individual instructions. For example, the symbols VALUE and WITH are subkeywords in the ADDRESS and PARSE instructions, respectively. For details, see the description of each instruction.

Whitespace characters (blanks or horizontal tabs) adjacent to keywords separate the keyword from the subsequent token. One or more whitespace characters following VALUE are required to separate the expression from the subkeyword in the example following:

```
ADDRESS VALUE expression
```

However, no whitespace character is required after the VALUE subkeyword in the following example, although it would improve readability:

```
ADDRESS VALUE"ENVIR"||number
```

2.1. ADDRESS
ADDRESS permanently changes the destination or I/O redirection of commands, or temporarily changes the destination and sends a command with optional I/O redirection of standard input, standard output, and standard error from or to Rexx objects.

Commands are strings sent to an external environment. You can send commands by specifying clauses consisting of only an expression or by using the ADDRESS instruction. (See Section 1.16, "Commands to External Environments").

To send a single command to a specified environment, code an environment, a literal string or a single symbol, which is taken to be a constant, followed by an expression. The environment name is the name of an external procedure or process that can process commands. The expression is evaluated to produce a character string value, and this string is routed to the environment to be processed as a command. After execution of the command, environment is set back to its original state, thus temporarily changing the destination for a single command. The special variable RC and the environment symbol .RS are set and errors and failures in commands processed in this way are trapped or traced.

The following environments are available in ooRexx:

sh (Unix-like systems only)
This is the default environment on Unix-like systems. It uses sh as a shell program to execute the command. All shell features such as redirection or piping can be used.

bsh, bash, csh, ksh, tcsh, zsh (Unix-like systems only)
These environments use alternate Unix-like system shells bsh, bash, csh, ksh, tcsh, or zsh to execute the command. If the appropriate shell is not installed on the system, executing a command in any of these environments will raise a failure.

cmd (Windows only)
This is the default environment on Windows systems. It uses cmd.exe as a command interpreter to execute the command. Command interpreter features such as internal commands, redirection or piping can be used.

command, system, ""
These environment names are synonyms for the default environments sh on Unix-like systems and cmd on Windows.

path
This environment executes commands directly, without using an intermediate command interpreter or shell program. It searches the environment variable PATH to locate the command to execute.
No shell features such as internal commands, redirection, piping or environment variable substitution are available.

The **WITH** subkeyword sets a command's I/O redirection. STDIN input can be redirected from a Rexx object to the command, and STDOUT and STDERR output from the command can be redirected to a Rexx object.

I/O redirection is permanent when specified on an ADDRESS instruction without a command, and temporary if a command is specified. Any permanent I/O redirection is associated with the environment name and will be saved and restored across function and subroutine calls. For permanent I/O redirections, any redirection objects or expressions are not evaluated at the time the ADDRESS instruction is processed. Each time a command is sent to this environment, these objects and expressions will be evaluated in the then current variable context.

**WITH INPUT**
redirects data from a stem, a stream, or other Rexx object to the command's input.

If option **NORMAL** is specified, the command's standard input will be used.

If option **STEM** is specified, **stem** must be a stem variable, where **stem.0** specifies the number of input lines, and each **stem.i** (with **i** from 1 through **stem.0**) specifies an input line.

If option **STREAM** is specified, **stream** must be a literal string, a constant symbol, or an expression enclosed in parentheses that evaluates to a string which is used as the stream name. Input lines for the command are read from **stream** using the **lineIn** method.

If option **USING** is specified, **expr** must be a literal string, a constant symbol, or an expression enclosed in parentheses that evaluates to a String, a Stem object, an InputStream, a Monitor or a File object, or an Array object or any other object that supports a **makeArray** method (e.g. a RexxQueue object). If the resulting object is a String, the command will receive a single input line, if it is a Stem, an InputStream, a Monitor, or a File object, input is redirected as described for options **STEM** and **STREAM**. If the resulting object is an Array object, all Array items are converted to strings and are sent to the command as input lines. Empty array items are ignored.

**WITH OUTPUT** and **WITH ERROR**
redirect a command's standard output or error output to a stem, a stream, or other Rexx objects.

If option **NORMAL** is specified, the command's default output destination, or default error destination is used.

If option **STEM** is specified, **stem** must be a stem variable. If **REPLACE** is specified together with **STEM**, the number of output lines is stored in **stem.0** and individual lines are stored as **stem.i**, with **i** running from 1 through **stem.0**. If **APPEND** is specified with **STEM**, individual lines are stored as **stem.i**, with **i** starting at the value of the existing **stem.0** plus 1. The initial **stem.0** value is then incremented by the total number of output lines for the command.

If option **STREAM** is specified, **stream** must be a literal string, a constant symbol, or an expression enclosed in parentheses that evaluates to a string which is used as the stream name. Output or error lines from the command are written to the stream with the **lineOut** method. If **REPLACE** is specified with **STREAM**, **stream** is truncated to zero length before any output lines are written. **REPLACE** is the default. If **APPEND** is specified with **STREAM**, output lines are appended to **stream**.

If option **USING** is specified, **expr** must be a constant symbol, or an expression enclosed in parentheses that evaluates to a Stem object, an OutputStream, a Monitor, a RexxQueue, a File object, or an OrderedCollection object. If the resulting object is a Stem or a File object, output lines are written as described for options **STEM** and **STREAM**. If it is an OutputStream or a Monitor object, output lines are always appended to the stream; neither **REPLACE** nor **APPEND** can be
specified in this case. If the object is a RexxQueue, method queue is used for each output line. Neither REPLACE nor APPEND can be specified for a RexxQueue. If the resulting object is an OrderedCollection object and REPLACE is specified, the collection is emptied using method empty before any output lines are added to the collection using method append. REPLACE is the default. If APPEND is specified, output lines are appended to the existing collection using method append.

Notes:

1. Specifying one of the INPUT, OUTPUT, or ERROR subkeywords more than once is an error.
2. If an input source object and an output or error target object is the same, Rexx uses appropriate read and write buffering to make sure results are correct.
3. If the standard output target and the standard error target object is the same object, Rexx will send interleaved output and error lines to the target.

Example 2.1. Instructions — ADDRESS

```
ADDRESS "CMD" 'dir "\Program Files"' -- Windows
ADDRESS "sh" "ls /usr/bin" -- Unix-like system
address "%" "cat" with input using "single line" -- Unix-like system: "single line"
address "%" "ver" with output stem v.; say v.2 -- "Microsoft Windows ..."

address "%" with input using (a) output using (a)
a = 4, 2, 3, 1
"sort"
say a~toString(, " ") -- 1 2 3 4
```

If you specify only environment, a lasting change of destination occurs: all commands (see Section 1.16.2, “Commands”) that follow are routed to the specified command environment, until the next ADDRESS instruction is processed. The previously selected environment is saved.

Assume that the environment for a Windows text editor is registered by the name EDIT:

Example 2.2. Instructions — ADDRESS environments

```
address CMD
"DIR C:\AUTOEXEC.BAT"
if rc=0 then "COPY C:\AUTOEXEC.BAT C:\*.TMP"
address EDIT
```

Subsequent commands are passed to the editor until the next ADDRESS instruction.

Similarly, you can use the VALUE form to make a lasting change to the environment. Here env_expression, which can be a variable name, is evaluated, and the resulting character string value forms the name of the environment. You can omit the subkeyword VALUE if env_expression does not begin with a literal string or symbol, that is, if it starts with a special character such as an operator character or parenthesis.

Example 2.3. Instructions — ADDRESS environments

```
ADDRESS ("ENVIR"||number) /* Same as ADDRESS VALUE "ENVIR"||number */
```
With no arguments, commands are routed back to the environment that was selected before the previous change of the environment, and the current environment name is saved. After changing the environment, repeated execution of ADDRESS alone, therefore, switches the command destination between two environments. Using a null string for the environment name ("") is the same as using the default environment.

The two environment names are automatically saved across internal and external subroutine and function calls. See the CALL instruction for more details.

The address setting is the currently selected environment name. You can retrieve the current address setting by using the ADDRESS built-in function. The Open Object Rexx: Application Programming Interfaces describes the creation of alternative subcommand environments.

### 2.2. ARG

ARG retrieves the argument strings provided to a program, internal routine, or method and assigns them to variables. It is a short form of the instruction:

```plaintext
PARSE UPPER ARG template_list
```

The `template_list` can be a single template or list of templates separated by commas. Each template consists of one or more symbols separated by whitespace characters, patterns, or both.

The objects passed to the program, routine, or method are converted to string values and parsed into variables according to the rules described in Chapter 9, Parsing.

The language processor converts the objects to strings and translates the strings to uppercase (that is, lowercase a-z to uppercase A-Z) before processing them. Use the PARSE ARG instruction if you do not want uppercase translation.

You can use the ARG and PARSE ARG instructions repeatedly on the same source objects (typically with different templates). The source objects do not change.

**Example 2.4. Instructions — ARG**

```plaintext
/* String passed is "Easy Rider" */
Arg adjective noun .

/* Now: ADJECTIVE contains "EASY" */
/* NOUN contains "RIDER" */
```

If you expect more than one object to be available to the program or routine, you can use a comma in the parsing `template_list` so each template is selected in turn.

**Example 2.5. Instructions — ARG**

```plaintext
/* Function is called by FRED("data X",1,5) */
```
Fred: Arg string, num1, num2

/* Now: STRING contains "DATA X" */
/* NUM1 contains "1" */
/* NUM2 contains "5" */

Notes:

1. The **ARG** built-in function can also retrieve or check the arguments.

2. The **USE ARG** instruction is an alternative way of retrieving arguments. **USE ARG** performs a direct, one-to-one assignment of argument objects to Rexx variables. You should use this when your program needs a direct reference to the argument object, without string conversion or parsing. **ARG** and **PARSE ARG** produce string values from the argument objects, and the language processor then parses the string values.

### 2.3. CALL

CALL calls a routine (if you specify `name`) or controls the trapping of certain conditions (if you specify `ON` or `OFF`).

To control trapping, you specify `OFF` or `ON` and the condition you want to trap. `OFF` turns off the specified condition trap. `ON` turns on the specified condition trap. The `usercondition` is a single symbol that is taken as a constant. The `trapname` is a symbol or string taken as a constant. All information on condition traps is contained in *Chapter 11, Conditions and Condition Traps*. 47
To call a routine, specify *name*, which must be a literal string or symbol that is taken as a constant. The routine called can be:

**An internal routine**
A subroutine that is in the same program as the CALL instruction or function call that calls it. Internal routines are located using label instructions.

**A built-in routine**
A function that is defined as part of the Rexx language.

**An external routine**
A subroutine that is neither built-in nor a label within the same same program as the CALL instruction call that invokes it. See Section 7.2.1, “Search Order” for details on the different types of external routines.

If *name* is a literal string (that is, specified in quotation marks), the search for internal routines is bypassed, and only a built-in function or an external routine is called. Note that built-in function names are in uppercase. Therefore, a literal string call to a built-in function must also use uppercase characters.

You can also specify (*expr*), any valid expression enclosed in parentheses. The expression is evaluated before any of the argument expressions, and the value is the target of the CALL instruction. The language processor does not translate the expression value into uppercase, so the evaluated name must exactly match any label name or built-in function name. See Section 1.12.3, “Labels” for a description of label names.

The called routine can optionally return a result. In this case, the CALL instruction is functionally identical with the clause:

```
result = name(expression)
```

You can use any number of *expressions*, separated by commas. The expressions are evaluated from left to right and form the arguments during execution of the routine. Any ARG, PARSE ARG, or USE ARG instruction or ARG built-in function in the called routine accesses these objects while the called routine is running. You can omit expressions, if appropriate, by including extra commas.

The CALL then branches to the routine called *name*, using exactly the same mechanism as function calls. See Chapter 7, Functions. The search order is as follows:

**Internal routines**
These are sequences of instructions inside the same program, starting at the label that matches *name* in the CALL instruction. If you specify the routine name in quotation marks, then an internal routine is not considered for that search order. The RETURN instruction completes the execution of an internal routine.

**Built-in routines**
These are routines built into the language processor for providing various functions. They always return an object that is the result of the routine. See Section 7.4, “Built-in Functions”.
Note

You can call any built-in function as a subroutine. Any result is stored in RESULT. Simply specify CALL, the function name (with no parenthesis) and any arguments, for example:

```rxml
call length "string"   /* Same as length("string") */
say result             /* Produces: 6               */
```

External routines

Users can write or use routines that are external to the language processor and the calling program. You can code an external routine in Rexx or in any language that supports the system-dependent interfaces. If the CALL instruction calls an external routine written in Rexx as a subroutine, you can retrieve any argument strings with the ARG, PARSE ARG, or USE ARG instructions or the ARG built-in function.

For more information on the search order, see Section 7.2.1, “Search Order”.

During execution of an internal routine, all variables previously known are generally accessible. However, the PROCEDURE instruction can set up a local variables environment to protect the subroutine and caller from each other. The EXPOSE option on the PROCEDURE instruction can expose selected variables to a routine.

Calling an external program or routine defined with a ::ROUTINE directive is similar to calling an internal routine. The external routine, however, is an implicit PROCEDURE in that all the caller's variables are always hidden. The status of internal values, for example NUMERIC settings, start with their defaults (rather than inheriting those of the caller). In addition, you can use EXIT to return from the routine.

When control reaches an internal routine, the line number of the CALL instruction is available in the variable SIGL (in the caller's variable environment). This can be used as a debug aid because it is possible to find out how control reached a routine. Note that if the internal routine uses the PROCEDURE instruction, it needs to EXPOSE SIGL to get access to the line number of the CALL.

After the subroutine processed the RETURN instruction, control returns to the clause following the original CALL. If the RETURN instruction specified an expression, the variable RESULT is set to the value of that expression. Otherwise, the variable RESULT is dropped (becomes uninitialized).

An internal routine can include calls to other internal routines, as well as recursive calls to itself.

Example 2.6. Instructions — CALL

```rxml
/* Recursive subroutine execution... */
arg z
call factorial z
say z"! =" result
exit
factorial: procedure   /* Calculate factorial by */
arg n                /* recursive invocation. */
if n=0 then return 1
call factorial n-1
return result * n
```
During internal subroutine (and function) execution, all important pieces of information are automatically saved and then restored upon return from the routine. These are:

- **The status of loops and other structures:** Executing a SIGNAL within a subroutine is safe because loops and other structures that were active when the subroutine was called are not ended. However, those currently active within the subroutine are ended.

- **Trace action:** After a subroutine is debugged, you can insert a TRACE Off at the beginning of it without affecting the tracing of the caller. If you want to debug a subroutine, you can insert a TRACE Results at the start and tracing is automatically restored to the conditions at entry (for example, Off) upon return. Similarly, ? (interactive debug) is saved across routines.

- **NUMERIC settings:** The DIGITS, FUZZ, and FORM of arithmetic operations are saved and then restored on return. A subroutine can, therefore, set the precision, for example, that it needs to use without affecting the caller.

- **ADDRESS settings:** The current and previous destinations for commands, including any associated I/O redirection configurations, are saved and then restored on return.

- **Condition traps:** CALL ON and SIGNAL ON are saved and then restored on return. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions the caller set up.

- **Condition information:** This information describes the state and origin of the current trapped condition. The CONDITION built-in function returns this information.

- **.RS value:** The value of the .RS environment symbol.

- **Elapsed-time clocks:** A subroutine inherits the elapsed-time clock (see Section 7.4.65, “TIME”) from its caller, but because the time clock is saved across routine calls, a subroutine or internal function can independently restart and use the clock without affecting its caller. For the same reason, a clock started within an internal routine is not available to the caller.

### 2.4. DO

![Diagram of DO loop]

**REPETITOR:**
DO groups instructions and optionally processes them repetitively. During repetitive execution, control variables (control1, control2, index, or item) can be stepped through some range of values.

LOOP groups instructions and processes them repetitively. LOOP behaves identically to DO, except for the simple LOOP ... END case, which is equivalent to DO FOREVER ... END.

Notes:

1. The LABEL and COUNTER phrases can be in any order, if used. They must precede any repetitor or conditional fragment.

2. The COUNTER phrase is only valid with a repetitive or conditional DO, it cannot be used on a simple DO group.

3. control1, control2, index, item, and ctr must be symbols that are valid variable names.

4. The exprr, expri, exprb, exprt, and exprf options, if present, are any expressions that evaluate to a number. The exprr and exprf options are further restricted to result in a positive whole number or zero. If necessary, the numbers are rounded according to the setting of NUMERIC DIGITS.

5. The exprw or expru options, if present, can be any expression that evaluates to 1 or 0. This includes the list form of a conditional expression, which is a list of expressions separated by ",". The list of expressions is evaluated left-to-right. Each subexpression must evaluate to either 0 or 1. Evaluation will stop with the first 0 result and 0 will be returned as the condition result. If all of the subexpressions evaluate to 1, then the condition result is also 1.

6. The TO, BY, and FOR phrases can be in any order, if used, and are evaluated in the order in which they are written.

7. The INDEX and ITEM phrases can be in any order, if used. They cannot be used more than once.
8. The instruction can be any instruction, including assignments, commands, message instructions, and keyword instructions (including any of the more complex constructs such as IF, SELECT, and the DO or LOOP instruction itself).

9. The subkeywords WHILE, UNTIL, OVER, WITH, INDEX, and ITEM are reserved within a DO or LOOP instruction in that they act as expression terminators for other keywords. Thus they cannot be used as symbols in any of the expressions. Similarly, TO, BY, and FOR cannot be used in expri, exprt, exprb, or exprf. FOREVER is also reserved, but only if it immediately follows the keyword DO or LOOP and is not followed by an equal sign. However, parentheses around or within an expression can prevent these keywords from terminating an expression. For example,

Example 2.7. Instructions — DO variable without parenthesis

```plaintext
do i = 1 while i < until
  say i
end
```

is considered a syntax error because of the variable named UNTIL. Using parentheses around the expression allows the variable UNTIL to be used:

Example 2.8. Instructions — DO variable with parenthesis

```plaintext
do i = 1 while (i < until)
  say i
end
```

10. The exprb option defaults to 1, if relevant.

11. The collection can be any expression that evaluates to an object that supports a makeArray method. Array and List items return an array with the items in the appropriate order, as do Streams. Tables, Stems, Directories, etc. are not ordered so the items get placed in the array in no particular order.

12. The supplier can be any expression that evaluates to an object that supports a supplier method.

For more information, refer to Appendix A, Using DO and LOOP.

2.5. DROP

DROP "unassigns" variables, that is, restores them to their original uninitialized state. If name is not enclosed in parentheses, it identifies a variable you want to drop and must be a symbol that is a valid variable name, separated from any other name by one or more whitespace characters or comments.

If parentheses enclose a single name, then its value is used as a subsidiary list of variables to drop. Whitespace characters are not necessary inside or outside the parentheses, but you can add them if desired. This subsidiary list must follow the same rules as the original list, that is, be valid character
strings separated by whitespace, except that no parentheses are allowed. The list need not contain any names—that is, it can be empty.

Variables are dropped from left to right. It is not an error to specify a name more than once or to drop a variable that is not known. If an exposed variable is named (see Section 2.7, “EXPOSE” and Section 2.19, “PROCEDURE”), then the original variable is dropped.

Example 2.9. Instructions — DROP

```plaintext
j=4
Drop a z.3 z.j
/* Drops the variables: A, Z.3, and Z.4 */
/* so that reference to them returns their names. */
```

Here, a variable name in parentheses is used as a subsidiary list.

Example 2.10. Instructions — DROP

```plaintext
mylist="c d e"
drop (mylist) f
/* Drops the variables C, D, E, and F */
/* Does not drop MYLIST */
```

Specifying a stem (that is, a symbol that contains only one period as the last character) assigns the stem variable to a new, empty stem object.

Example:

```plaintext
Drop z.
/* Assigns stem variable z. to a new empty stem object */
```

2.6. EXIT

EXIT leaves a program unconditionally. Optionally, EXIT returns a result object to the caller. The program is stopped immediately, even if an internal routine is being run. If no internal routine is active, RETURN and EXIT are identical in their effect on the program running.

If you specify expression, it is evaluated and the object resulting from the evaluation is passed back to the caller when the program stops.

Example 2.11. Instructions — EXIT

```plaintext
j=3
Exit j*4
/* Would exit with the string "12" */
```
If you do not specify expression, no data is passed back to the caller. If the program was called as a function, this is detected as an error.

You can also use EXIT within a method. The method is stopped immediately, and the result object, if specified, is returned to the sender. If the method has previously issued a \textit{REPLY} instruction, the EXIT instruction must not include a result expression.

\textbf{Notes:}

1. If the program was called through a command interface, an attempt is made to convert the returned value to a return code acceptable by the underlying operating system. The returned string must be a whole number in the range \(-32768\) to \(32767\). If the conversion fails, no error is raised, and a return code of 0 is returned.

2. If you do not specify EXIT, EXIT is implied at the end of the program, but no result value is returned.

3. On Unix-like systems the returned value is limited to a numerical value between 0 and 255.

\textbf{2.7. EXPOSE}

\texttt{EXPOSE name}\bigbreak

EXPOSE causes the object variables identified in \texttt{name} to be exposed to a method. References to exposed variables, including assigning and dropping, access variables in the current object's variable pool. (An object variable pool is a collection of variables that is associated with an object rather than with any individual method.) Therefore, the values of existing variables are accessible, and any changes are persistent even after RETURN or EXIT from the method.

Any changes a method makes to an object variable pool are immediately visible to any other methods that share the same object variable scope. All other variables that a method uses are local to the method and are dropped on RETURN or EXIT. If an EXPOSE instruction is included, it must be the first instruction of the method.

If parentheses enclose a single \texttt{name}, then, after the variable \texttt{name} is exposed, the character string value of \texttt{name} is immediately used as a subsidiary list of variables. Whitespace characters are not necessary inside or outside the parentheses, but you can add them if desired. This subsidiary list must follow the same rules as the original list, that is, valid variable names separated by whitespace characters, except that no parentheses are allowed.

Variables are exposed in sequence from left to right. It is not an error to specify a name more than once, or to specify a name that has not been used as a variable.

\textbf{Example 2.12. Instructions — EXPOSE}

\begin{verbatim}
/* Example of exposing object variables */
myobj = .myclass~new
myobj-c     /* Would display "Z is: 120" */
::class myclass /* The ::CLASS directive */
::method c    /* The ::METHOD directive */
::method d    /* Would display "Z is: 120" */
::class myclass /* The ::CLASS directive */
::method c    /* The ::METHOD directive */
\end{verbatim}
You can expose an entire collection of compound variables by specifying their stem in the variable list or a subsidiary list. The variables are exposed for all operations.

**Example 2.13. Instructions — EXPOSE**

```plaintext
expose j k c. d. /* This exposes "J", "K", and all variables whose */ /* name starts with "C." or "D." */
  c.1="7." /* This sets "C.1" in the object */ /* variable pool, even if it did not */ /* previously exist. */
```

### 2.8. FORWARD

FORWARD forwards the message that caused the currently active method to begin running. The FORWARD instruction can change parts of the forwarded message, such as the target object, the message name, the arguments, and the superclass override.

If you specify the TO option, the language processor evaluates `exprt` to produce a new target object for the forwarded message. The `exprt` is a literal string, a constant symbol, or an expression enclosed in parentheses. If you do not specify the TO option, the initial value of the Rexx special variable SELF is used.

If you specify the ARGUMENTS option, the language processor evaluates `expra` to produce an array object that supplies the set of arguments for the forwarded message. The `expra` can be a literal string,

**Note**

You can specify the options in any order.
a constant symbol, or an expression enclosed in parentheses. The **ARGUMENTS** value must evaluate to a Rexx array object.

If you specify the **ARRAY** option, each `expri` is an expression (use commas to separate the expressions). The language processor evaluates the expression list to produce a set of arguments for the forwarded message. It is an error to use both the **ARRAY** and the **ARGUMENTS** options on the same FORWARD instruction.

If you specify neither **ARGUMENTS** nor **ARRAY**, the language processor uses the same arguments specified on the original method call.

If you specify the **MESSAGE** option, the `exprm` is a literal string, a constant symbol, or an expression enclosed in parentheses. If you specify an expression enclosed in parentheses, the language processor evaluates the expression to obtain its value. The uppercase character string value of the **MESSAGE** option is the name of the message that the FORWARD instruction issues.

If you do not specify **MESSAGE**, FORWARD uses the message name used to call the currently active method.

If you specify the **CLASS** option, the `exprs` is a literal string, a constant symbol, or an expression enclosed in parentheses. This is the class object used as a superclass specifier on the forwarded message.

If you do not specify **CLASS**, the message is forwarded without a superclass override.

If you do not specify the **CONTINUE** option, the language processor immediately exits the current method before forwarding the message. Results returned from the forwarded message are the return value from the original message that called the active method (the caller of the method that issued the FORWARD instruction). Any conditions the forwarded message raises are raised in the calling program (without raising a condition in the method issuing the FORWARD instruction).

If you specify the **CONTINUE** option, the current method does not exit and continues with the next instruction when the forwarded message completes. If the forwarded message returns a result, the language processor assigns it to the special variable RESULT. If the message does not return a result, the language processor drops (uninitializes) the variable RESULT.

The FORWARD instruction passes all or part of an existing message invocation to another method. For example, the FORWARD instruction can forward a message to a different target object, using the same message name and arguments.

**Example 2.14. Instructions — FORWARD**

```rxml
::method substr
  forward to (self~string) /* Forward to the string value */
```

You can use FORWARD in an UNKNOWN method to reissue to another object the message that the UNKNOWN method traps.

**Example 2.15. Instructions — FORWARD**

```rxml
::method unknown
  use arg msg, args
  /* Forward to the string value */
  /* passing along the arguments */
  forward to (self~string) message (msg) arguments (args)
```
You can use FORWARD in a method to forward a message to a superclass's methods, passing the same arguments. This is very common usage in object INIT methods.

Example 2.16. Instructions — FORWARD

```
::class savings subclass account
::method init
  expose type penalty
  forward class (super) continue /* Send to the superclass */
  type = "Savings" /* Now complete initialization */
  penalty = "1% for balance under 500"
```

In the preceding example, the **CONTINUE** option causes the FORWARD message to continue with the next instruction, rather than exiting the Savings class INIT method.

### 2.9. GUARD

GUARD controls a method's exclusive access to an object.

GUARD ON acquires for an active method exclusive use of its object variable pool. This prevents other methods that also require exclusive use of the same variable pool from running on the same object. If another method has already acquired exclusive access, the GUARD instruction causes the issuing method to wait until the variable pool is available.

GUARD OFF releases exclusive use of the object variable pool. Other methods that require exclusive use of the same variable pool can begin running.

If you specify WHEN, the method delays running until the *expression* evaluates to `.true`. If the *expression* evaluates to `.false`, GUARD waits until another method assigns or drops an object variable (that is, a variable named on an EXPOSE instruction) used in the WHEN expression. When an object variable changes, GUARD reevaluates the WHEN expression. If the *expression* evaluates to `.true`, the method resumes running. If the *expression* evaluates to `.false`, GUARD resumes waiting.

The condition *expression* after a WHEN can also be a list of *expressions* which is evaluated left-to-right. Each *expression* must evaluate to either `.false` or `.true`. Evaluation will stop with the first `.false` result and `.false` will be returned as the condition result. If all of the *expressions* evaluate to `.true`, then the condition result is also `.true`.

Example 2.17. Instructions — GUARD

```
::method c
  expose y
  if y>0 then
    return 1
  else
    return 0
::method d
```
IF conditionally processes an instruction or group of instructions depending on the evaluation of the expression. The expression is evaluated and must result in 0 or 1.

The instruction after the THEN is processed only if the result is .true. If you specify an ELSE, the instruction after ELSE is processed only if the result of the evaluation is .false.

Example:

```
if answer="YES" then say "OK!"
else say "Why not?"
```

Remember that if the ELSE clause is on the same line as the last clause of the THEN part, you need a semicolon before ELSE.

Example:

```
if answer="YES" then say "OK!"; else say "Why not?"
```

ELSE binds to the nearest IF at the same level. You can use the NOP instruction to eliminate errors and possible confusion when IF constructs are nested, as in the following example.
Example 2.18. Instructions — IF

```
If answer = "YES" Then
  If name = "FRED" Then
    say "OK, Fred."
  Else
    nop
  Else
    say "Why not?"
```

The *expression* may also be a list of expressions separated by ",". The list of expressions is evaluated left-to-right. Each subexpression must evaluate to either 0 or 1. Evaluation will stop with the first 0 result and 0 will be returned as the condition result. If all of the subexpressions evaluate to 1, then the condition result is also 1.

Example 2.19. Instructions — IF

```
If answer~datatype('w'), answer//2 = 0 Then
  say answer "is even"
Else
  say answer "is odd"
```

The example above is not the same as using the following

Example 2.20. Instructions — IF

```
If answer~datatype('w') & answer//2 = 0 Then
  say answer "is even"
Else
  say answer "is odd"
```

The logical & operator will evaluate both terms of the operation, so the term "answer//2" will result in a syntax error if answer is a non-numeric value. With the list conditional form, evaluation will stop with the first .false result, so the "answer//2" term will not be evaluated if the datatype test returns 0.

Notes:

1. The *instruction* can be any assignment, message instruction, command, or keyword instruction, including any of the more complex constructs such as DO, LOOP, SELECT, or the IF instruction itself. A null clause is not an instruction, so putting an extra semicolon (or label) after THEN or ELSE is not equivalent to putting a dummy instruction (as it would be in C). The NOP instruction is provided for this purpose.

2. Except when within a bracketed subexpression, the symbol THEN cannot be used within *expression*, because the keyword THEN is treated differently in that it need not start a clause. This allows the expression on the IF clause to be ended by THEN, without a semicolon (;) being required.

2.11. INTERPRET

```
INTERPRET expression
```
INTERPRET processes instructions that have been built dynamically by evaluating expression.

The expression is evaluated to produce a character string, and is then processed (interpreted) just as though the resulting string were a line inserted into the program and bracketed by a DO; and an END;.

Any instructions (including INTERPRET instructions) are allowed, but note that constructions such as DO...END and SELECT...END must be complete. For example, a string of instructions being interpreted cannot contain a LEAVE or ITERATE instruction (valid only within a repetitive loop) unless it also contains the whole repetitive DO...END or LOOP...END construct.

A semicolon is implied at the end of the expression during execution, if one was not supplied.

```
Example 2.21. Instructions — INTERPRET

    /* INTERPRET example */
    data="FRED"
    interpret data ";= 4"
    /* Builds the string "FRED = 4" and */
    /* Processes: FRED = 4; */
    /* Thus the variable FRED is set to "4" */

    /* Another INTERPRET example */
    data="do 3; say "Hello there!"; end"
    interpret data        /* Displays: */
                          /*   Hello there! */
                          /*   Hello there! */
                          /*   Hello there! */
```

Notes:

1. Labels within the interpreted string are not permanent and are, therefore, an error.
2. Executing the INTERPRET instruction with TRACE R or TRACE I can be helpful in interpreting the results you get.

```
Example 2.22. Instructions — INTERPRET

    /* Here is a small Rexx program. */
    Trace Int
    name="Kitty"
    indirect="name"
    interpret 'say "Hello"' indirect '"!"'
```

When this is run, you get the following trace:

```
3 *-* name="Kitty"
   >L>  "Kitty"
   >>>  "Kitty"
4 *-* indirect="name"
   >L>  "name"
   >>>  "name"
5 *-* interpret 'say "Hello"' indirect '"!"
   >L>  "say "Hello"
   >V>  INDIRECT => "name"
   >D>  " " => "say "Hello" name"
   >L>  ""!"
   >D>  "" => "say "Hello" name!"
   >>>  "say "Hello" name!"
```

60
5  **-** say "Hello" name"!"
    >L>   "Hello"
    >V>   NAME => "Kitty"
    >O>   " " => "Hello Kitty"
    >L>   "!"
    >O>   "" => "Hello Kitty!"
    >>>   "Hello Kitty!"
Hello Kitty!

Lines 3 and 4 set the variables used in line 5. Execution of line 5 then proceeds in two stages. First the string to be interpreted is built up, using a literal string, a variable (INDIRECT), and another literal string. The resulting pure character string is then interpreted, just as though it were actually part of the original program. Because it is a new clause, it is traced as such (the second **-** trace flag under line 5) and is then processed. Again a literal string is concatenated to the value of a variable (NAME) and another literal, and the final result (Hello Kitty!) is then displayed.

3. For many purposes, you can use the VALUE function instead of the INTERPRET instruction. The following line could, therefore, have replaced line 5 in the previous example:

   Example 2.23. Instructions — INTERPRET

   say "Hello" value(indirect)"!"

   INTERPRET is usually required only in special cases, such as when two or more statements are to be interpreted together, or when an expression is to be evaluated dynamically.

4. You cannot use a directive within an INTERPRET instruction.

2.12. ITERATE

ITERATE alters the flow within a repetitive loop (that is, any DO construct other than that with a simple DO or a LOOP instruction).

Execution of the group of instructions stops, and control is passed to the DO or LOOP instruction just as though the END clause had been encountered. The control variable, if any, is incremented and tested, as usual, and the group of instructions is processed again, unless the DO or LOOP instruction ends the loop.

The name is a symbol, taken as a constant. If name is not specified, ITERATE continues with the current repetitive loop. If name is specified, it must be the name of the control variable or the LABEL name of a currently active loop, which can be the innermost, and this is the loop that is stepped. Any active loops inside the one selected for iteration are ended (as though by a LEAVE instruction).

   Example 2.24. Instructions — ITERATE

   loop label MyLabelName i=1 to 4 /* label set to 'MYLABELNAME' */
   if i=2 then iterate
   say i
Notes:

1. If specified, *name* must match the symbol naming the control variable or LABEL name in the DO or LOOP clause in all respects except the case. No substitution for compound variables is carried out when the comparison is made.

2. A loop is active if it is currently being processed. If a subroutine is called, or an INTERPRET instruction is processed, during the execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. ITERATE cannot be used to continue with an inactive loop.

3. If more than one active loop uses the same name, ITERATE selects the innermost loop.

2.13. LEAVE

**LEAVE**

**name**

**Notes:**

**Example 2.25. Instructions — LEAVE**

```plaintext
max=5
do label myDoBlock /* define a label 'MYDOBLOCK' */
  loop i=1 to max /* label defaults to control variable 'I' */
  if i = 2 then iterate i
  if i = 4 then leave myDoBlock
  say i
  end i
end myDoBlock
say 'after looping' i 'times'
/* Displays the following
  1
  3
  after looping 4 times
*/
```

/* Displays the numbers:
1
3
4
*/
Notes:

1. If specified, name must match the symbol naming the control variable or LABEL name in the DO, LOOP, or SELECT clause in all respects except the case. No substitution for compound variables is carried out when the comparison is made.

2. A loop is active if it is currently being processed. If a subroutine is called, or an INTERPRET instruction is processed, during execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. LEAVE cannot be used to end an inactive block.

3. If more than one active block uses the same control variable, LEAVE selects the innermost block.

2.14. LOOP

```
DO
  LABEL name
  COUNTER ctr
  REPETITOR - fragment
    instruction
    END name
    REPETITOR:
      control1 = expr
      TO expr
      BY exprb
      FOR exprf
      control2 OVER collection
      FOR exprf
      WITH INDEX index
      OVER supplier
      FOREVER exprf
    CONDITIONAL:
```
LOOP groups instructions and processes them repetitively.

LOOP behaves identically to DO, except for the simple LOOP ... END case, which is equivalent to DO FOREVER ... END.

For details refer to Section 2.4, "DO".

2.15. NOP

NOP is a dummy instruction that has no effect. It can be useful as the target of a THEN or ELSE clause.

Example 2.26. Instructions — NOP

Select
  when a=c then nop /* Do nothing */
  when a>c then say "A > C"
  otherwise say "A < C"
end

Note

Putting an extra semicolon instead of the NOP would merely insert a null clause, which would be ignored. The second WHEN clause would be seen as the first instruction expected after the THEN, and would, therefore, be treated as a syntax error. NOP is a true instruction, however, and is, therefore, a valid target for the THEN clause.

2.16. NUMERIC
NUMERIC changes the way in which a program carries out arithmetic operations. The options of this instruction are described in detail in Chapter 10, Numbers and Arithmetic.

**NUMERIC DIGITS**
controls the precision to which arithmetic operations and built-in functions are evaluated. If you omit expression1, the precision defaults to 9 digits, but can be overridden on a source-file basis using the ::OPTIONS directive. Otherwise, the character string value result of expression1 must evaluate to a positive whole number and must be larger than the current NUMERIC FUZZ setting.

There is no limit to the value for DIGITS (except the amount of storage available), but high precisions are likely to require a great amount of processing time. It is recommended that you use the default value whenever possible.

You can retrieve the current NUMERIC DIGITS setting with the DIGITS built-in function.

**NUMERIC FORM**
controls the form of exponential notation for the result of arithmetic operations and built-in functions. This can be either SCIENTIFIC (in which case only one, nonzero digit appears before the decimal point) or ENGINEERING (in which case the power of 10 is always a multiple of 3). The default is SCIENTIFIC, but can be overridden on a source-file basis using the ::OPTIONS directive. The subkeywords SCIENTIFIC or ENGINEERING set the FORM directly, or it is taken from the character string result of evaluating the expression (expression2) that follows VALUE. The result in this case must be either SCIENTIFIC or ENGINEERING. You can omit the subkeyword VALUE if expression2 does not begin with a symbol or a literal string, that is, if it starts with a special character, such as an operator character or parenthesis.

You can retrieve the current NUMERIC FORM setting with the FORM built-in function.

**NUMERIC FUZZ**
controls how many digits, at full precision, are ignored during a numeric comparison operation. If you omit expression3, the default is 0 digits, but can be overridden on a source-file basis using the ::OPTIONS directive. Otherwise, the character string value result of expression3 must evaluate to 0 or a positive whole number rounded, if necessary, according to the current NUMERIC DIGITS setting, and must be smaller than the current NUMERIC DIGITS setting.

NUMERIC FUZZ temporarily reduces the value of NUMERIC DIGITS by the NUMERIC FUZZ value during every numeric comparison. The numbers are subtracted under a precision of DIGITS minus FUZZ digits during the comparison and are then compared with 0.

You can retrieve the current NUMERIC FUZZ setting with the FUZZ built-in function.
Note

The three numeric settings are automatically saved across internal subroutine and function calls. See the CALL instruction for more details.

2.17. OPTIONS

The OPTIONS instruction is used to pass special requests to the language processor.

The expression is evaluated, and individual words in the result that are meaningful to the language processor will be obeyed. Options might control how the interpreter optimizes code, enforces standards, enables implementation-dependent features, etc. Unrecognized words in the result are ignored, since they are assumed to be instructions for a different language processor.

Open Object Rexx does not recognize any option keywords.

2.18. PARSE

Note

You can specify UPPER and CASELESS or LOWER and CASELESS in either order.
PARSE assigns data from various sources to one or more variables according to the rules of parsing. (See Chapter 9, Parsing.)

If you specify UPPER, the strings to be parsed are translated to uppercase before parsing. If you specify LOWER, the strings are translated to lowercase. Otherwise no translation takes place.

If you specify CASELESS, character string matches during parsing are made independent of the case. This means a letter in uppercase is equal to the same letter in lowercase.

The template_list can be a single template or list of templates separated by commas. Each template consists of one or more symbols separated by whitespace, patterns, or both.

Each template is applied to a single source string. Specifying several templates is not a syntax error, but only the PARSE ARG variant can supply more than one non-null source string. See Section 9.8.1, "Parsing Several Strings" for information on parsing several source strings.

If you do not specify a template, no variables are set but the data is prepared for parsing, if necessary. Thus for PARSE PULL, a data string is removed from the current data queue, for PARSE LINEIN (and PARSE PULL if the queue is empty), a line is taken from the default input stream, and for PARSE VALUE, expression is evaluated. For PARSE VAR, the specified variable is accessed. If it does not have a value, the NOVALUE condition is raised, if it is enabled.

The following list describes the data for each variant of the PARSE instruction.

**PARSE ARG**

parses the strings passed to a program, routine, or method as input arguments. See the ARG instruction for details and examples.

*Note*

Parsing uses the string values of the argument objects. The USE ARG instruction provides direct access to argument objects. You can also retrieve or check the argument objects to a Rexx program, routine, or method with the ARG built-in function.

**PARSE LINEIN**

parses the next line of the default input stream. (See Chapter 14, Input and Output Streams for a discussion of Rexx input and output.) PARSE LINEIN is a shorter form of the following instruction:

```plaintext
PARSE VALUE LINEIN() WITH template_list
```

If no line is available, program execution usually pauses until a line is complete. Use PARSE LINEIN only when direct access to the character input stream is necessary. Use the PULL or PARSE PULL instructions for the usual line-by-line dialog with the user to maintain generality. PARSE LINEIN will not pull lines from the external data queue.

To check if any lines are available in the default input stream, use the LINES built-in function.

**PARSE PULL**

parses the next string of the external data queue. If the external data queue is empty, PARSE PULL reads a line of the default input stream (the user’s terminal), and the program pauses, if
necessary, until a line is complete. You can add data to the head or tail of the queue by using the **PUSH** and **QUEUE** instructions, respectively. You can find the number of lines currently in the queue with the **QUEUED** built-in function. The queue remains active as long as the language processor is active. Other programs in the system can alter the queue and use it to communicate with programs written in Rexx. See also the **PULL** instruction.

**Note**

PULL and PARSE PULL read the current data queue. If the queue is empty, they read the default input stream, .INPUT (typically, the keyboard).

**PARSE SOURCE**

parses data describing the source of the program, routine or method running.

The first two tokens of source string are the operating system name, followed by either **COMMAND**, **FUNCTION**, **SUBROUTINE**, **METHOD**, or **REQUIRES**, depending on whether the program was called as a host command, or from a function call in an expression, or using the CALL instruction, or as a method of an object, or from a ::REQUIRES directive to run the prolog code. These two tokens are followed by the path specification of the program, or the name given to a Package, Routine or Method instance, or the string **INSTORE** when run via rexx -e.

The string might look like one of the following:

- **WindowsNT COMMAND** C:\Program Files\ooRexx\rextry.rex
- **LINUX COMMAND** /usr/local/bin/rextry.rex

**PARSE VALUE**

parses the data, a character string, that is the result of evaluating expression. If you specify no expression, the null string is used. Note that WITH is a subkeyword in this context and cannot be used as a symbol within expression.

Thus, for example:

```plaintext
PARSE VALUE time() WITH hours ":" mins ":" secs
```

gets the current time and splits it into its constituent parts.

**PARSE VAR name**

parses the character string value of the variable name. The name must be a symbol that is valid as a variable name, which means it cannot start with a period or a digit. Note that the variable name is not changed unless it appears in the template, so that, for example:

```plaintext
PARSE VAR string word1 string
```

removes the first word from string, puts it in the variable word1, and assigns the remainder back to string.

```plaintext
PARSE UPPER VAR string word1 string
```

also translates the data from string to uppercase before it is parsed.
PARSE VERSION
parses information describing the language level and the date of the language processor. This information consists of five blank-delimited words:

- The string `REXX-ooRexx_5.0.0(MT)_64-bit`, if using the ooRexx interpreter at version 5, release 0, modification 0, and compiled for 64-bit addressing mode.
- The language level description, for example `6.05` for ooRexx 5.0, or `6.04` for ooRexx 4.2.
- Three tokens that describe the language processor release date in the same format as the default for the `DATE` built-in function, for example, "1 Sep 2016".

2.19. PROCEDURE

PROCEDURE, within an internal routine (subroutine or function), protects the caller’s variables by making them unknown to the instructions that follow it. After a RETURN instruction is processed, the original variable environment is restored and any variables used in the routine (that were not exposed) are dropped. (An exposed variable is one belonging to the caller of a routine that the PROCEDURE instruction has exposed. When the routine refers to, or alters, the variable, the original (caller’s) copy of the variable is used.) An internal routine need not include a PROCEDURE instruction. In this case the variables it is manipulating are those the caller owns. If the PROCEDURE instruction is used, it must be the first instruction processed after the CALL or function invocation; that is, it must be the first instruction following the label.

If you use the EXPOSE option, any variable specified by the `name` is exposed. Any reference to it (including setting and dropping) is made to the variables environment the caller owns. Hence, the values of existing variables are accessible, and any changes are persistent even on RETURN from the routine. If the `name` is not enclosed in parentheses, it identifies a variable you want to expose and must be a symbol that is a valid variable name, separated from any other `name` with one or more whitespace characters.

If parentheses enclose a single `name`, then, after the variable `name` is exposed, the character string value of `name` is immediately used as a subsidiary list of variables. Whitespace characters are not necessary inside or outside the parentheses, but you can add them if desired. This subsidiary list must follow the same rules as the original list, that is, valid variable names separated by whitespace characters, except that no parentheses are allowed.

Variables are exposed from left to right. It is not an error to specify a name more than once, or to specify a name that the caller has not used as a variable.

Any variables in the main program that are not exposed are still protected. Therefore, some of the caller’s variables can be made accessible and can be changed, or new variables can be created. All these changes are visible to the caller upon RETURN from the routine.

Example 2.27. Instructions — PROCEDURE

/* This is the main Rexx program */
j=1; z.1="a"
call toft
Note that if Z.J in the EXPOSE list is placed before J, the caller's value of J is not visible, so Z.1 is not exposed.

The variables in a subsidiary list are also exposed from left to right.

Example 2.28. Instructions — PROCEDURE

```rexx
/* This is the main Rexx program */
j=1;k=6;m=9
a ="j k m"
call test
exit

/* This is a subroutine */
test: procedure expose (a) /* Exposes A, J, K, and M */
say a j k m                /* Displays "j k m 1 6 9" */
return
```

You can use subsidiary lists to more easily expose a number of variables at a time or, with the VALUE built-in function, to manipulate dynamically named variables.

Example 2.29. Instructions — PROCEDURE

```rexx
/* This is the main Rexx program */
c=11; d=12; e=13
Showlist="c d"     /* but not E */
call Playvars
say c d e f        /* Displays "11 New 13 9" */
exit

/* This is a subroutine */
Playvars: procedure expose (showlist) f
say word(showlist,2) /* Displays "d" */
say value(word(showlist,2),"New") /* Displays "12" and sets new value */
say value(word(showlist,2)) /* Displays "New" */
```

Specifying a stem as name exposes this stem and all possible compound variables whose names begin with that stem.

Example 2.30. Instructions — PROCEDURE

```rexx
/* This is the main Rexx program */
a.=11; i=13; j=15
i = i + 1
C.5 = "FRED"
call lucky7
```
say a. a.1 i j c. c.5
say "You should see 11 14 15 C. FRED"
exit

lucky7:Procedure Expose i j a. c.
/* This exposes I, J, and all variables whose   */
/* names start with A. or C.                   */
A.1=7  /* This sets A.1 in the caller-'s        */
      /* environment, even if it did not       */
      /* previously exist.                     */
return

Note
Variables can be exposed through several generations of routines if they are included in all intermediate PROCEDURE instructions.

See the CALL instruction and Chapter 7, Functions for details and examples of how routines are called.

2.20. PULL

PULL reads a string from the head of the external data queue or, if the external data queue is empty, from the standard input stream (typically the keyboard). (See Chapter 14, Input and Output Streams for a discussion of Rexx input and output.) It is a short form of the following instruction:

The current head of the queue is read as one string. Without a template_list specified, no further action is taken and the string is thus effectively discarded. The template_list can be a single template or list of templates separated by commas, but PULL parses only one source string. Each template consists of one or more symbols separated by whitespace, patterns, or both.

If you specify several comma-separated templates, variables in templates other than the first one are assigned the null string. The string is translated to uppercase (that is, lowercase a-z to uppercase A-Z) and then parsed into variables according to the rules described in Chapter 9, Parsing. Use the PARSE PULL instruction if you do not desire uppercase translation.

Note
If the current data queue is empty, PULL reads from the standard input (typically, the keyboard). If there is a PULL from the standard input, the program waits for keyboard input with no prompt.
Example:

Say "Do you want to erase the file? Answer Yes or No:"
Pull answer.
if answer="NO" then say "The file will not be erased."

Here the dummy placeholder, a period (.), is used in the template to isolate the first word the user enters.

If the external data queue is empty, a line is read from the default input stream and the program pauses, if necessary, until a line is complete. (This is as though PARSE UPPER LINEIN had been processed. For details see PARSE LINEIN)

The QUEUED built-in function returns the number of lines currently in the external data queue.

2.21. PUSH

PUSH stacks the string resulting from the evaluation of expression LIFO (Last In, First Out) into the external data queue. (See Chapter 14, Input and Output Streams for a discussion of Rexx input and output.)

If you do not specify expression, a null string is stacked.

Example 2.31. Instructions — PUSH

```
a="Fred"
push       /* Puts a null line onto the queue */
push a 2   /* Puts "Fred 2" onto the queue */
```

The QUEUED built-in function returns the number of lines currently in the external data queue.

2.22. QUEUE

QUEUE appends the string resulting from expression to the tail of the external data queue. That is, it is added FIFO (First In, First Out). (See Chapter 14, Input and Output Streams for a discussion of Rexx input and output.)

If you do not specify expression, a null string is queued.

Example 2.32. Instructions — QUEUE

```
a="Toft"
queue a 2   /* Enqueues "Toft 2" */
queue      /* Enqueues a null line behind the last */
```
The *QUEUED* built-in function returns the number of lines currently in the external data queue.

### 2.23. RAISE

![Diagram of RAISE function]

**OPTIONS:**

- **ADDITIONAL**
- **ARRAY**
- **DESCRIPTION**
- **EXIT**
- **RETURN**

**Note**

You can specify the options **ADDITIONAL, ARRAY, DESCRIPTION, RETURN, and EXIT** in any order. However, if you specify **EXIT** without **expre** or **RETURN** without **exprr**, it must appear last.

RAISE returns or exits from the currently running routine or method and raises a condition in the caller (for a routine) or sender (for a method). See *Chapter 11, Conditions and Condition Traps* for details of
the actions taken when conditions are raised. The RAISE instruction can raise all conditions that can be trapped.

If the ERROR or FAILURE condition is raised, you must supply the associated return code as errorcode or failurecode, respectively. These can be literal strings, constant symbols, or expressions enclosed in parentheses. If you specify an expression enclosed in parentheses, a subexpression, the language processor evaluates the expression to obtain its character string value.

If the SYNTAX condition is raised, you must supply the associated Rexx error number as number. This error number can be either a Rexx major error code or a Rexx detailed error code in the form nn.nnn. The number can be a literal string, a constant symbol, or an expression enclosed in parentheses. If you specify an expression enclosed in parentheses, the language processor evaluates the expression to obtain its character string value.

If a USER condition is raised, you must supply the associated user condition name as usercondition, which must be a symbol that is taken as a constant.

If you specify the ADDITIONAL option, the language processor evaluates expra to produce an object that supplies additional object information associated with the condition. The expra can be a literal string, a constant symbol, or an expression enclosed in parentheses. The ADDITIONAL entry of the condition object and the "A" option of the CONDITION built-in function return this additional object information. For SYNTAX conditions, the ADDITIONAL value must evaluate to a single-dimensional Array.

If you specify the ARRAY option, each expri is an expression (use commas to separate the expressions). The language processor evaluates the expression list to produce an array object that supplies additional object information associated with the condition. The ADDITIONAL entry of the condition object and the "A" option of the CONDITION built-in function return this additional object information as an array of values. It is an error to use both the ARRAY option and the ADDITIONAL option on the same RAISE instruction.

The content of expra or expri is used as the contents of the secondary error message produced for a condition.

If you specify neither ADDITIONAL nor ARRAY, there is no additional object information associated with the condition.

If you specify the DESCRIPTION option, the exprd can be a literal string, a constant symbol, or an expression enclosed in parentheses. If you specify an expression enclosed in parentheses, the language processor evaluates the expression to obtain its character string value. This is the description associated with the condition. The "D" option of the CONDITION built-in function and the DESCRIPTION entry of the condition object return this string.

If you do not specify DESCRIPTION, the language processor uses a null string as the descriptive string.

If you specify the RETURN or EXIT option, the language processor evaluates the expression exprr or expre, respectively, to produce a result object that is passed back to the caller or sender as if it were a RETURN or EXIT result. The expre or exprr is a literal string, a constant symbol, or an expression enclosed in parentheses. If you specify an expression enclosed in parentheses, the language processor evaluates the expression to obtain its character string value. If you do not specify exprr or expre, no result is passed back to the caller or sender. In either case, the effect is the same as that of the RETURN or EXIT instruction. The EXIT option is the default. Following the return or exit, the appropriate action is taken in the caller or sender (see Section 11.1, “Action Taken when a Condition Is Not Trapped”). If specified, the result value can be obtained from the RESULT entry of the condition object.
Example 2.33. Instructions — RAISE

```plaintext
raise syntax 40                       /* Raises syntax error 40                */
raise syntax 40.12 array (1, number)  /* Raises syntax error 40, subcode 12    */
/* Passing two substitution values       */
raise syntax (errnum)                 /* Uses the value of the variable ERRNUM */
/* as the syntax error number           */
raise user badvalue                   /* Raises user condition BADVALUE        */
```

If you specify PROPAGATE, and there is a currently trapped condition, this condition is raised again in the caller (for a routine) or sender (for a method). Any ADDITIONAL, DESCRIPTION, ARRAY, RETURN, or EXIT information specified on the RAISE instruction replaces the corresponding values for the currently trapped condition. A SYNTAX error occurs if no condition is currently trapped.

Example 2.34. Instructions — RAISE

```plaintext
signal on syntax
a = "xyz"
c = a+2                                            /* Raises the SYNTAX condition */
.
.
exit
syntax:
raise propagate                                    /* Propagates SYNTAX information to caller */
```

2.24. REPLY

REPLY sends an early reply from a method to its caller. The method issuing REPLY returns control, and possibly a result, to its caller to the point from which the message was sent; meanwhile, the method issuing REPLY continues running on a newly created thread.

If you specify `expression`, it is evaluated and the object resulting from the evaluation is passed back. If you omit `expression`, no object is passed back.

Unlike RETURN or EXIT, the method issuing REPLY continues to run after the REPLY until it issues an EXIT or RETURN instruction. The EXIT or RETURN must not specify a result expression.

Example 2.35. Instructions — REPLY

```plaintext
reply 42                                  /* Returns control and a result */
call tidyup                               /* Can run in parallel with sender */
return
```

Notes:

1. You can use REPLY only in a method.
2. A method can execute only one REPLY instruction.
3. When the method issuing the REPLY instruction is the only active method on the current thread with exclusive access to the object’s variable pool, the method retains exclusive access on the new thread. When other methods on the thread also have access, the method issuing the REPLY releases its access and reacquires the access on the new thread. This might force the method to wait until the original activity has released its access.

See Chapter 12, Concurrency for a complete description of concurrency.

### 2.25. RETURN

```
RETURN expression
```

RETURN returns control, and possibly a result, from a Rexx program, method, or routine to the point of its invocation.

If no internal routine (subroutine or function) is active, RETURN and EXIT are identical in their effect on the program that is run.

If called as a routine, expression (if any) is evaluated, control is passed back to the caller, and the Rexx special variable RESULT is set to the value of expression. If you omit expression, the special variable RESULT is dropped (becomes uninitialized). The various settings saved at the time of the CALL (for example, tracing and addresses) are also restored.

If a function call is active, the action taken is identical, except that expression must be specified on the RETURN instruction. The result of expression is then used in the original expression at the point where the function was called. See Chapter 7, Functions for more details.

If a method is processed, the language processor evaluates expression (if any) and returns control to the point from which the method’s activating message was sent. If called as a term of an expression, expression is required. If called as a message instruction, expression is optional and is assigned to the Rexx special variable RESULT if a return expression is specified. If the method has previously issued a REPLY instruction, the RETURN instruction must not include a result expression.

If a PROCEDURE instruction was processed within an internal subroutine or internal function, all variables of the current generation are dropped (and those of the previous generation are exposed) after expression is evaluated and before the result is used or assigned to RESULT.

---

**Note**

If the RETURN statement causes the program to return to the operating system on a Unix-like systems the value returned is limited to a numerical value between 0 and 255 (an unsigned byte). If no expression is supplied then the default value returned to the operating system is zero.

### 2.26. SAY

```
SAY expression
```

SAY returns control, and possibly a result, from a Rexx program, method, or routine to the point of its invocation.
SAY writes a line to the default output stream, which displays it to the user. However, the output destination can depend on the implementation. See Chapter 14, Input and Output Streams for a discussion of Rexx input and output. The string value of the expression result is written to the default character output stream. The resulting string can be of any length. If you omit expression, the null string is written.

The SAY instruction is a shorter form of the following instruction:

```
CALL LINEOUT, expression
```

except that:

1. SAY does not affect the special variable RESULT.
2. If you use SAY and omit expression, a null string is used.
3. CALL LINEOUT can raise NOTREADY; SAY will not.

Example 2.36. Instructions — SAY

```
data=100
Say data "divided by 4 =>" data/4
/* Displays: "100 divided by 4 => 25" */
```

Notes:

1. Data from the SAY instruction is sent to the default output stream (.OUTPUT). However, the standard rules for redirecting output apply to the SAY output.
2. The SAY instruction does not format data; the operating system and the hardware handle line wrapping. However, formatting is accomplished, the output data remains a single logical line.

2.27. SELECT
SELECT conditionally calls one of several alternative instructions.

Evaluation of the expression list after a WHEN is as follows:

**SELECT without CASE**

The list of expressions after a WHEN is evaluated left-to-right. Each expression must evaluate to either `.false` or `.true`. Evaluation will stop with the first `.false` result and `.false` will be returned as the condition result.

If all of the expressions evaluate to `.true`, then the condition result is also `.true`.

**SELECT CASE**

The case_expression is evaluated only once, before the first WHEN instruction is processed. The list of expressions after a WHEN is evaluated left-to-right. Each expression is compared to the result of case_expression using `==`. Evaluation will stop with the first `.true` result and `.true` will be returned as the condition result.

If all comparisons evaluate to `.false`, then the condition result is also `.false`.

If the result from above is `.true`, the instruction following the associated THEN (which can be a complex instruction such as IF, DO, LOOP, or SELECT) is processed and control is then passed to the END. If the result is `.false`, control is passed to the next WHEN clause.

If none of the WHEN results are `.true`, control is passed to the instructions, if any, after OTHERWISE. In this situation, the absence of an OTHERWISE produces an error, however, you can omit the instruction list that follows OTHERWISE.

Example 2.37. Instructions — SELECT

```plaintext
balance=100
check=50
balance = balance - check
Select
    when balance > 0 then
        say "Congratulations! You still have" balance "dollars left."
    when balance = 0 then do
        say "Warning, Balance is now zero! STOP all spending."
        say "You cut it close this month! Hope you do not have any"
        say "checks left outstanding."
    end
Otherwise do
    say "You have just overdrawn your account."
    say "Your balance now shows" balance "dollars."
    say "Oops! Hope the bank does not close your account."
end
end  /* Select */
```

Example 2.38. Instructions — SELECT

```plaintext
select
    when answer-datatype('w'), answer//2 = 0 Then
        say answer "is even"
    when answer-datatype('w'), answer//2 = 1 Then
        say answer "is odd"
```
The example above is not the same as using the following:

```plaintext
select
  when answer~datatype('w') & answer//2 = 0 Then
    say answer "is even"
  when answer~datatype('w') & answer//2 = 1 Then
    say answer "is odd"
  otherwise
    say answer "is not a number"
end
```

The logical "&" operator will evaluate both terms of the operation, so the term "answer//2" will result in a syntax error if answer is a non-numeric value. With the list conditional form, evaluation will stop with the first false result, so the "answer//2" term will not be evaluated if the datatype test returns false.

**Example 2.39. Instructions — SELECT CASE**

```plaintext
select case random(6)
  when 1 then say "bad luck!"
  when 5, 6 then say "great!"
  otherwise say "try again"
end
```

**Notes:**

1. The instruction can be any assignment, command, message instruction, or keyword instruction, including any of the more complex constructs, such as DO, LOOP, IF, or the SELECT instruction itself.

2. A null clause is not an instruction, so putting an extra semicolon (or label) after a THEN clause is not equivalent to putting a dummy instruction. The NOP instruction is provided for this purpose.

3. Except when within a bracketed subexpression, the symbol THEN cannot be used within expression, because the keyword THEN is treated differently in that it need not start a clause. This allows the expression on the WHEN clause to be ended by the THEN without a semicolon (;).

### 2.28. SIGNAL

```plaintext
SIGNAL
  labelname
  VALUE
  expression
```
SIGNAL causes an unusual change in the flow of control (if you specify `labelname` or `VALUE expression`), or controls the trapping of certain conditions (if you specify ON or OFF).

To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap. All information on condition traps is contained in Chapter 11, Conditions and Condition Traps.

To change the flow of control, a label name is derived from `labelname` or taken from the character string result of evaluating the `expression` after `VALUE`. The `labelname` you specify must be a literal string or symbol that is taken as a constant. If you specify a symbol for `labelname`, the search looks for a label with uppercase characters. If you specify a literal string, the search uses the literal string directly. You can locate label names with lowercase letters only if you specify the label as a literal string with the same case. Similarly, for SIGNAL VALUE, the lettercase of `labelname` must match
exactly. You can omit the subkeyword VALUE if expression does not begin with a symbol or literal string, that is, if it starts with a special character, such as an operator character or parenthesis. All active pending DO, IF, SELECT, and INTERPRET instructions in the current routine are then ended and cannot be resumed. Control is then passed to the first label in the program that matches the given name, as though the search had started at the beginning of the program.

The labelname and usercondition are single symbols, which are taken as constants. The trapname is a string or symbol taken as a constant.

Example 2.40. Instructions — SIGNAL

```plaintext
Signal fred; /* Transfer control to label FRED below */
....
....
Fred: say "Hi!"
```

If there are duplicates, control is always passed to the first occurrence of the label in the program.

When control reaches the specified label, the line number of the SIGNAL instruction is assigned to the special variable SIGL. This can aid debugging because you can use SIGL to determine the source of a transfer of control to a label.

2.29. TRACE

TRACE controls the tracing action (that is, how much is displayed to the user) during the processing of a Rexx program. Tracing describes some or all of the clauses in a program, producing descriptions
of clauses as they are processed. TRACE is mainly used for debugging. Its syntax is more concise than that of other Rexx instructions because TRACE is usually entered manually during interactive debugging. (This is a form of tracing in which the user can interact with the language processor while the program is running.)

Note

TRACE cannot be used in the Rexx macro space.

If specified, the number must be a whole number.

The string or expression evaluates to:

- A numeric option
- One of the valid prefix or alphabetic character (word) options
- Null

The symbol is taken as a constant and is therefore:

- A numeric option
- One of the valid prefix or alphabetic character (word) options

The option that follows TRACE or the character string that is the result of evaluating expression determines the tracing action. You can omit the subkeyword VALUE if expression does not begin with a symbol or a literal string, that is, if it starts with a special character, such as an operator or parenthesis.

2.29.1. Trace Alphabetic Character (Word) Options

Although you can enter the word in full, only the first capitalized letter is needed; all following characters are ignored. That is why these are referred to as alphabetic character options.

TRACE actions correspond to the alphabetic character options as follows:

**All**
Traces (that is, displays) all clauses before execution.

**Commands**
Traces all commands before execution. If the command results in an error or failure, tracing also displays the return code from the command.

**Error**
Traces any command resulting in an error or failure after execution, together with the return code from the command.

**Failure**
Traces any command resulting in a failure after execution, together with the return code from the command. This is the same as the Normal option.
Prefix Option

Intermediates
Traces all clauses before execution. Also traces intermediate results during the evaluation of expressions and substituted names.

Labels
Traces method and routine invocations, internal subroutine calls, transfers of control because of the SIGNAL instruction, and labels passed during program execution. This is especially useful with debug mode, when the language processor pauses after each invocation or call.

Normal
Traces any failing command after execution, together with the return code from the command. This is the default setting, if not overridden using the ::OPTIONS directive.

For the default Windows command processor, an attempt to enter an unknown command raises a FAILURE condition. The CMD return code for an unknown command is 1. An attempt to enter a command in an unknown command environment also raises a FAILURE condition; in such a case, the variable RC is set to 30.

Off
Traces nothing and resets the special prefix option (described later) to OFF.

Results
Traces all clauses before execution. Displays the final results (in contrast with Intermediates option) of the expression evaluation. Also displays values assigned during PULL, ARG, PARSE, and USE instructions. This setting is recommended for general debugging.

2.29.2. Prefix Option

The prefix ? is valid alone or with one of the alphabetic character options. You can specify the prefix more than once, if desired. Each occurrence of a prefix on an instruction reverses the action of the previous prefix. The prefix must immediately precede the option (no intervening whitespace).

The prefix ? controls interactive debugging. During normal execution, a TRACE option with a prefix of ? causes interactive debugging to be switched on. (See Chapter 15, Debugging Aids for full details of this facility.) When interactive debugging is on, interpretation pauses after most clauses that are traced. For example, the instruction TRACE ?E makes the language processor pause for input after executing any command that returns an error, that is, a nonzero return code or explicit setting of the error condition by the command handler.

Any TRACE instructions in the program being traced are ignored to ensure that you are not taken out of interactive debugging unexpectedly.

You can switch off interactive debugging in several ways:

- Entering TRACE 0 turns off all tracing.

- Entering TRACE with no options restores the defaults—it turns off interactive debugging but continues tracing with TRACE Normal (which traces any failing command after execution).

- Entering TRACE ? turns off interactive debugging and continues tracing with the current option.

- Entering a TRACE instruction with a ? prefix before the option turns off interactive debugging and continues tracing with the new option.
Using the `?` prefix, therefore, switches you in or out of interactive debugging. Because the language processor ignores any further TRACE statements in your program after you are in interactive debug mode, use `CALL TRACE "?"` to turn off interactive debugging.

### 2.29.3. Numeric Options

If interactive debugging is active and the option specified is a positive whole number (or an expression that evaluates to a positive whole number), that number indicates the number of debug pauses to be skipped. (See Chapter 15, Debugging Aids for further information.) However, if the option is a negative whole number (or an expression that evaluates to a negative whole number), all tracing, including debug pauses, is temporarily inhibited for the specified number of clauses. For example, `TRACE -100` means that the next 100 clauses that would usually be traced are not displayed. After that, tracing resumes as before.

### 2.29.4. Tracing Tips

- When a loop is traced, the DO clause itself is traced on every iteration of the loop.
- You can retrieve the trace actions currently in effect by using the `TRACE` built-in function.
- The trace output of commands traced before execution always contains the final value of the command, that is, the string passed to the environment, and the clause generating it.
- Trace actions are automatically saved across subroutine, function, and method calls. See Section 2.3, “CALL” for more details.

One of the most common traces you will use is:

**Example 2.41. Instructions — TRACE**

```
TRACE ?R
/* Interactive debugging is switched on if it was off, */
/* and tracing results of expressions begins. */
```

### 2.29.5. The Format of Trace Output

Every clause traced appears with automatic formatting (indentation) according to its logical depth of nesting, for example. Results, if requested, are indented by two extra spaces and are enclosed in double quotation marks so that leading and trailing whitespace characters are apparent. Any control codes in the data encoding (ASCII values less than “20”x) are replaced by a question mark (?) to avoid screen interference. Results other than strings appear in the string representation obtained by sending them a `STRING` message. The resulting string is enclosed in parentheses. The line number in the program precedes the first clause traced on any line. All lines displayed during tracing have a three-character prefix to identify the type of data being traced. These can be:

```
_*_*
```

Identifies the source of a single clause, that is, the data actually in the program.
+++
Identifies a trace message. This can be the nonzero return code from a command, the prompt message when interactive debugging is entered, an indication of a syntax error when in interactive debugging.

>!
Identifies an entry to a routine or method. This trace entry will only appear if tracing is enabled using the ::OPTIONS directive using TRACE A, TRACE R, TRACE I or TRACE L.

>!
Identifies the result of a subkeyword value in a keyword instruction, like the TO subkeyword of a DO or LOOP instruction, or the DIGITS subkeyword in a NUMERIC instruction.

>>>
Identifies the result of an expression (for TRACE R) or the value returned from a subroutine call, or a value evaluated by execution of a DO loop.

>>
Identifies a variable assignment or a message assignment result. The trace message includes both the name of the assignment target and the assigned value. Assignment trace lines are displayed by assignment instructions, variable assigned via PARSE, ARG, PULL, or USE ARG, as well as control variable updates for DO and LOOP instructions.

>.>
Identifies the value assigned to a placeholder during parsing (see Section 9.1.2, “The Period as a Placeholder”).

The following prefixes are used only if TRACE Intermediates is in effect:

>A>
Identifies a value used as a function, subroutine, or message argument.

>C>
The data traced is the original name of the compound variable and the name of a compound variable, after the name has been replaced by the value of the variable but before the variable is used. If no value was assigned to the variable, the trace shows the variable in uppercase characters.

>E>
The data traced is the name and value of an environment symbol.

>F>
The data traced is the name and result of a function call.

>L>
The data traced is a literal (string, uninitialized variable, or constant symbol).

>M>
The data traced is the name and result of an object message.

>N>
The data traced is the name and result of a namespace-prefixed symbol.

>O>
The data traced is the name and result of an operation on two terms.
The data traced is the name and result of a prefix operation.

The data traced is the name of an argument variable and the name of the referenced variable.

The data traced is the name and contents of a variable.

Note

The characters => indicate the value of a variable or the result of an operation.

The characters <= indicate a value assignment. The name to the left of the marker is the assignment topic. The data to the right of the marker is the assigned value.

The character ? could indicate a non-printable character in the output.

If no option is specified on a TRACE instruction, or if the result of evaluating the expression is null, the default tracing actions are restored. The defaults are TRACE N and interactive debugging (?) off.

Following a syntax error that SIGNAL ON SYNTAX does not trap, the clause in error is always traced.

2.30. USE

The USE instruction can be used to
• retrieve the argument objects by using USE ARG or USE STRICT ARG, or to
• define local variables in a method by using USE LOCAL.

2.30.1. USE ARG, USE STRICT ARG

USE ARG retrieves the argument objects provided in a program, routine, function, or method and assigns them to variables, variable references, or message term assignments.

Each name must be a valid variable name, a variable reference term, or an assignment message term. The names are assigned from left to right. For each name you specify, the language processor assigns it a corresponding argument from the program, routine, function, or method call. If there is no corresponding argument, name is assigned the value of expr. If = expr is not specified for the given argument, the variable name is dropped. If the assignment target is a variable reference term, the corresponding argument must never be omitted. If the assignment target is a message term, no action is taken for omitted arguments.
A USE ARG instruction can be processed repeatedly and it always accesses the same current argument data.

If $expr$ is specified for an argument, the expression is evaluated to provide a default value for an argument when the corresponding argument does not exist. The default $expr$ must be a literal string, a constant symbol, or an expression enclosed in parentheses. No default value is allowed for variable reference terms.

The STRICT option imposes additional constraints on argument processing. The number of arguments must match the number of names, otherwise an error is raised. An argument is considered optional if $expr$ has been specified for the argument.

An ellipsis (...), can be specified after the last variable in a USE STRICT ARG statement to indicate that more arguments may follow. This allows defining a minimum number of arguments that must be supplied or for which there are default values defined, which may optionally be followed by any additional arguments.

Example 2.42. Instructions — USE

```
/* USE Example                       */
/* FRED("Ogof X",1,5) calls function */
Fred: use arg string, num1, num2

/* Now: STRING contains "Ogof X" */
/* NUM1 contains "1" */
/* NUM2 contains "5" */
```

```
/* Another example, shows how to pass non-string arguments with USE ARG */
/* Pass a stem and an array to a routine to modify one element of each */
stem.1 = "Value"
array = .array-of("Item")
say "Before subroutine:" stem.1 array[1] /* Shows "Value Item" */
Call Change_First stem. , array
say "After subroutine:" stem.1 array[1] /* Shows "NewValue NewItem" */
Exit

Change_First: Procedure
Use Arg substem., subarray
substem.1 = "NewValue"
subarray[1] = "NewItem"
Return
```

```
/* USE STRICT Example              */
/* FRED("Ogof X",1) calls function */
Fred: use strict arg string, num1, num2=4

/* Now: STRING contains "Ogof X" */
/* NUM1 contains "1" */
/* NUM2 contains "4" */
```

In the above example, a call to the function FRED may have either 2 or 3 arguments. The STRICT keyword on the USE instruction will raise a syntax error for any other combination of arguments.

Example 2.43. Instructions — USE

```
call test "one"
call test "one", "two"
call test "one", "two", "three"
```
call test "one", "three", "four", "five"
exit

test: procedure /* a minimum of one argument must be supplied */
use strict arg v1, v2="zwei", ...
say "There are ["arg()"] argument(s); v1,v2=["v1","v2"]"
do i=3 to arg()
say " arg #" i"=["arg(i)"]"
end
say ".--"
return

Output:

There are [1] argument(s); v1,v2=[one,zwei]
--
There are [2] argument(s); v1,v2=[one, two]
--
There are [3] argument(s); v1,v2=[one, two]
  arg # 3=[three]
--
There are [5] argument(s); v1,v2=[one, zwei]
  arg # 3=[three]
  arg # 4=[four]
  arg # 5=[five]
--

The assignment targets may be any term that can be on the left side of an assignment statement.

Example 2.44. Instructions — USE

expose myArray myDirectory
use arg myArray[1], myDirectory~name

would be equivalent to

myArray[1] = arg(1)
myDirectory~name = arg(2)

You can retrieve or check the arguments by using the ARG built-in function. The ARG and PARSE ARG instructions are alternative ways of retrieving arguments. ARG and PARSE ARG access the string values of arguments. USE ARG performs a direct, one-to-one assignment of arguments. This is preferable when you need direct access to an argument, without translation or parsing. USE ARG also allows access to both string and non-string argument objects; ARG and PARSE ARG convert the arguments to string values before parsing.

2.30.2. USE LOCAL

USE LOCAL defines local variables in a method.

Generally object variables must be specified in a method using EXPOSE, while all other variables used in the method will become local variables. In contrast to this, USE LOCAL allows to explicitly
declare local variables, while all other variables not listed on the USE LOCAL instruction will automatically become object variables.

Each name must be a valid variable name. If no name is specified, all variables will become object variables.

If a USE LOCAL instruction is present, it must be the first instruction of the method.

Note that USE LOCAL will always keep Rexx special variables RC, RESULT, SIGL, SELF, and SUPER as local variables.

Example 2.45. Instructions — USE LOCAL

```plaintext
::method init
USE LOCAL x y z   -- only x, y, and z are local
                -- all other become object variables
::method init
use local       -- any variable is an object variable
```
Directives

A Rexx program contains one or more executable code units. Directive instructions separate these executable units. A directive begins with a double colon (::) and is a nonexecutable instruction. For example, it cannot appear in a string for the INTERPRET instruction to be interpreted. The first directive instruction in a program marks the end of the main executable section of the program.

For a program containing directives, all directives are processed first to set up the program's classes, methods, and routines. Then any program code in the main code unit (preceding the first directive) is processed. This code can use any classes, methods, and routines that the directives established.

Supported directives are ::ANNOTATE, ::ATTRIBUTE, ::CLASS, ::CONSTANT, ::METHOD, ::OPTIONS, ::REQUIRES, ::RESOURCE, and ::ROUTINE.

A directive requires a semicolon (;) as a terminating delimiter unless the end of a line implies it.

3.1. ::ANNOTATE

The ::ANNOTATE directive creates annotations to the package, its classes, methods, attributes, constants, or routines.

The ::ANNOTATE directive can add metadata information (called "annotations") in the form of name/value pairs to the current package, or any of its classes, methods, attributes, constants, and routines.

An annotation attribute, class, constant, method, and routine must be a valid class name, constant name, method name, or routine name defined with its respective directive in the same source program.

The annotation name must be a symbol that is taken as a constant; a literal string is not allowed. The annotation value must be a single literal string, a symbol, or a valid number, optionally preceded by a plus or minus sign, that is taken as a constant.

Example 3.1. ANNOTATE directive

::annotate package author "B. Fox" -- annotate current package
::class command
::attribute address
::attribute command
::method init
use strict arg command, address = ""
self-command = command
self-address = address

::annotate class command languagelevel "6.05" -- annotate class COMMAND
::annotate attribute command os "unix windows mac" -- annotate attribute COMMAND
::annotate method init version "100" maxParms 2 -- annotate method INIT

Notes:

1. Each ::ANNOTATE directive, except for ::ANNOTATE PACKAGE, must be placed after its respective ::ATTRIBUTE, ::CLASS, ::CONSTANT, ::METHOD, or ::ROUTINE directive in the sourcefile.

2. When annotating non-floating attributes, constants, or methods, the ::ANNOTATE directive must precede any following ::CLASS directive.

3. An annotation for an attribute, a method, or a routine should be placed after any attribute/method/routine code body, as ::ANNOTATE, like any other directive, will end the code body.

4. A subclass of an annotated class will not inherit any annotations of its superclass or mixinclass.

5. A copy of an annotated method, package, or routine object will keep any existing annotations.

6. The ::ANNOTATE directive cannot be used to annotate packages, classes, methods, attributes, constants, and routines which are provided by Rexx or have been made available through a ::REQUIRES directive.

7. Currently Rexx does not use or redefine any specific annotation names.

8. To query an annotation name/value pair, to get a list of all attached annotation name/value pairs, or to add new or change existing annotations, use
   • the Package methods annotation and annotations,
   • the Class methods annotation and annotations,
   • the Method methods annotation and annotations, or
   • the Routine methods annotation and annotations.

3.2. ::ATTRIBUTE

The ::ATTRIBUTE directive creates attribute methods and defines the method properties.
The ::ATTRIBUTE directive creates accessor methods for object instance variables. An accessor method allows an object instance variable to be retrieved or assigned a value. ::ATTRIBUTE can create an attribute getter method, a setter method, or the getter/setter pair.

The name is a literal string or a symbol that is taken as a constant. The name must also be a valid Rexx variable name. The ::ATTRIBUTE directive creates methods in the class specified in the most recent ::CLASS directive. If no ::CLASS directive precedes an ::ATTRIBUTE directive, the attribute methods are not associated with a class but are accessible to the main (executable) part of a program through the .METHODS built-in object. Only one ::ATTRIBUTE directive can appear for any method name not associated with a class.

If you do not specify either GET or SET, ::ATTRIBUTE will create two attribute methods with the names name and name=. These are the methods for getting and setting an attribute. These generated methods are equivalent to the following code sequences:

<table>
<thead>
<tr>
<th>Example 3.2. ATTRIBUTE directive equivalent code</th>
</tr>
</thead>
<tbody>
<tr>
<td>::method name         -- attribute get method</td>
</tr>
<tr>
<td>expose name           -- establish direct access to object variable (attribute)</td>
</tr>
<tr>
<td>use strict arg        -- enforce zero parameters</td>
</tr>
<tr>
<td>return name           -- return object variable's current value</td>
</tr>
<tr>
<td>::method &quot;NAME=&quot;      -- attribute set method</td>
</tr>
<tr>
<td>expose name           -- establish direct access to object variable (attribute)</td>
</tr>
<tr>
<td>use strict arg name   -- retrieve argument and assign it to the object variable</td>
</tr>
</tbody>
</table>

Both methods will be created with the same method properties (for example, PRIVATE, GUARDED, etc.). If GET or SET are not specified, the pair of methods will be automatically generated. In that case, there is no method code body following the directive, so another directive (or the end of the program) must follow the ::ATTRIBUTE directive.

If GET or SET is specified, only the single get or set attribute method is generated. Specifying separate GET or SET ::ATTRIBUTE directives allows the methods to be created with different properties. For example, the sequence:

<table>
<thead>
<tr>
<th>Example 3.3. ATTRIBUTE directive — get and set methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>::attribute name get</td>
</tr>
<tr>
<td>::attribute name set private</td>
</tr>
</tbody>
</table>

will create a NAME method with PUBLIC access and a NAME= method with PRIVATE access.

The GET and SET options may also be used to override the default method body generated for the attribute. This is frequently used so the SET attribute method can perform new value validation.

If you specify the CLASS option, the created methods are class methods. See Chapter 4, Objects and Classes. The attribute methods are associated with the class specified on the most recent ::CLASS directive. The ::ATTRIBUTE must be preceded by a ::CLASS directive if CLASS is specified.
If **ABSTRACT** is specified, then all created methods will be marked as ABSTRACT and will raise an error if directly invoked. For ABSTRACT methods there is no method code body following the directive, so another directive (or the end of the program) must follow the ::ATTRIBUTE directive.

If **DELEGATE** is specified, execution of get method *name* and set method *name* (depending on whether GET or SET or none of these two is specified on the ::ATTRIBUTE directive) is delegated to object *delegatename*. It is a common design pattern to delegate method execution to an embedded object. The directive for such a delegation:

```plaintext
::attribute name delegate delegatename
```

is equivalent to the following code sequence:

**Example 3.4. DELEGATE subkeyword equivalent code**

```plaintext
::method name expose delegatename forward to(delegatename)
::method "NAME=" expose delegatename forward to(delegatename)
```

If the **EXTERNAL** option is specified, then *spec* identifies a method in an external native library that will be invoked as the named method. The *spec* is a literal string containing a series of whitespace-delimited tokens defining the external method. The first token must be **LIBRARY**, which indicates the method resides in a native library of the type allowed on a ::REQUIRES directive. The second token must identify the name of the external library. The external library is located using platform-specific mechanisms for loading libraries. For Unix-like systems, the library name is case-sensitive. The third token is optional and specifies the name of the method within the library package. If not specified, the ::METHOD name is used. The target package method name is case insensitive.

If the **GET** or **SET** option is not specified with the **EXTERNAL** option, then two method objects need to be created. The target method name is appended to the string "GET" to derive the name of the getter attribute method. To generate the setter attribute method, the name is appended to the string "SET". If **GET** or **SET** is specified and the method name is not specified within *spec*, then the target library method name is generated by concatenating *name* with "GET" or "SET" as appropriate. If the method name is specified in *spec* and **GET** or **SET** is specified, the spec name will be used unchanged.

**Example 3.5. ATTRIBUTE directive — naming the get and set methods**

```plaintext
-- maps "NAME" method to "GETNAME and "NAME=" to "SETNAME"
::ATTRIBUTE name EXTERNAL "LIBRARY mylib"
-- maps "ADDRESS" method to "GETADDRESS"
::ATTRIBUTE address GET EXTERNAL "LIBRARY mylib setHomeAddress"
-- maps "ADDRESS=" method to "setHomeAddress"
::ATTRIBUTE address SET EXTERNAL "LIBRARY mylib setHomeAddress"
```

**Notes:**

1. You can specify all options in any order.
2. If you specify the **PACKAGE** option, the methods are created with a package-scope, if you specify the **PRIVATE** option, the created methods are private methods. Package-scope and private methods have restricted access rules on how they can be invoked. See Section 4.2.8, “Public, Package-Scope, and Private Methods” for details of how these methods can be used. If you omit the **PACKAGE** or **PRIVATE** option, or specify **PUBLIC**, the method is a public method that any sender can activate.

3. If you specify the **UNGUARDED** option, the methods can be called while other methods are active on the same object. If you do not specify **UNGUARDED**, the method requires exclusive use of the object variable pool; it can run only if no other method that requires exclusive use of the object variable pool is active on the same object.

4. If you specify the **PROTECTED** option, the methods are protected methods. (See Chapter 13, The Security Manager for more information.) If you omit the **PROTECTED** option or specify **UNPROTECTED**, the methods are not protected.

5. It is an error to specify **ATTRIBUTE** more than once within a class definition that creates a duplicate get or set method.

### 3.3. **::CLASS**

The **::CLASS** directive causes the interpreter to create a Rexx class.

The **::CLASS** directive creates a Rexx class named `classname`. The `classname` is a literal string or symbol that is taken as a constant. The created class is available to programs through the Rexx environment symbol `.classname`. The `classname` acquires all methods defined by subsequent **::METHOD** directives until the end of the program or another **::CLASS** directive is found. Only null clauses (comments or blank lines) can appear between a **::CLASS** directive and any following directive instruction or the end of the program. Only one **::CLASS** directive can appear for `classname` in a program.

If you specify the **METACLASS** option, the instance methods of the `metaclass` class become class methods of the `classname` class. (See Chapter 4, Objects and Classes.) The `metaclass` and `classname` are literal strings or symbols that are taken as constants. In the search order for methods, the metaclass methods precede inherited class methods and follow any class methods defined by **::METHOD** directives with the **CLASS** option.

If you specify the **PUBLIC** option, the class is visible beyond its containing Rexx program to any other program that references this program with a **::REQUIRES** directive. If you do not specify the **PUBLIC** option, the class is visible only within its containing Rexx program. All public classes defined within a program are used before PUBLIC classes created with the same name.
If you specify the **SUBCLASS** option, the class becomes a subclass of the class `sclass` for inheritance of instance and class methods. The `sclass` is a literal string or symbol that is taken as a constant.

If you specify the **MIXINCLASS** option, the class becomes a subclass of the class `mclass` for inheritance of instance and class methods. You can add the new class instance and class methods to existing classes by using the **INHERIT** option on a ::CLASS directive or by sending an INHERIT message to an existing class. If you specify neither the **SUBCLASS** nor the **MIXINCLASS** option, the class becomes a non-mixin subclass of the Object class.

If you specify the **ABSTRACT** option, the class will be marked as an abstract class. Trying to create an instance of an abstract class will raise an error. Only subclasses of abstract classes will allow to create instances from.

If you specify the **INHERIT** option, the class inherits instance methods and class methods from the classes `iclasses` in their order of appearance (leftmost first). This is equivalent to sending a series of `inherit` messages to the class object, with each `inherit` message (except the first) specifying the preceding class in `iclasses` as the `classpos` argument. As with the `inherit` message, each of the classes in `iclasses` must be a mixin class. The `iclasses` is a whitespace-separated list of literal strings or symbols that are taken as constants. If you omit the **INHERIT** option, the class inherits only from `sclass`.

**Example 3.6. CLASS directive**

```ruby
::class rectangle
  ::method area    /* defined for the RECTANGLE class */
    expose width height
    return width*height

::class triangle
  ::method area    /* defined for the TRIANGLE class */
    expose width height
    return width*height/2
```

The ::CLASS directives in a program are processed in the order in which they appear. If a ::CLASS directive has a dependency on ::CLASS directives that appear later in the program, processing of the directive is deferred until all of the class's dependencies have been processed.

**Example 3.7. CLASS directive deferred processing**

```ruby
::class savings subclass account  /* requires the ACCOUNT class */
  ::method type
    return "a Savings Account"

::class account
  ::method type
    return "an Account"
```

The Savings class in the preceding example is not created until the Account class that appears later in the program has been created.

**Notes:**

1. You can specify the options `METACLASS`, `MIXINCLASS`, `SUBCLASS`, and `PUBLIC` in any order.
2. If you specify **INHERIT**, it must be the last option.
3.4. ::CONSTANT

The ::CONSTANT directive creates methods that return constant values for a class and its instances.

A ::CONSTANT directive defines a method that returns a constant value. This is useful for creating named constants associated with a class.

The name is a literal string or a symbol that is taken as a constant. A method of the given name is created as both an instance method and a class method of the most recent ::CLASS directive. A ::CLASS directive is not required before a ::CONSTANT directive. If no ::CLASS directive precedes ::CONSTANT, a single "floating" constant method is created that is not associated with a class but is accessible through the .METHODS built-in object. Only one ::CONSTANT directive can appear for any method name not associated with a class.

The methods created by a ::CONSTANT directive are UNGUARDED and will have a return result that is specified by value. If specified, the constant value must be a single literal string, a symbol that is taken as a constant, or an expression enclosed in parentheses. Also permitted is the single character "-" or "+" followed by a symbol that is a valid number. If value is omitted, the constant name will return its value in uppercase.

Here are some examples of valid constants:

**Example 3.8. CONSTANT examples**

```
::class MathConstants public
::constant pi 3.14159265
::constant author "Isaac Asimov"
::constant absolute_zero -273.15
::constant e ( rxcalcexp(1) )
::constant eSquare (self-e ** 2)
::constant primes (2, 3, 5, 7, 11, 13)

::class Search
::constant caseless

::requires rxmath library
```

A ::CONSTANT directive is a shorthand syntax for creating constants associated with a class. The created name constant can be accessed using either the class object or an instance of the class itself.

**Example 3.9. CONSTANT access examples**

```
say "Pi is" .MathConstants~pi          -- displays "Pi is 3.14159265"
instance = .MathConstants~new
say "Pi is" instance~pi               -- also displays "Pi is 3.14159265"

say .Search~caseless                  -- "CASELESS"
```

::class MathConstants public
::constant pi 3.14159265

::class Search
::constant caseless
Notes:

1. Calculated ::CONSTANT directives (where \textit{value} is an expression enclosed in parenthesis) can reference any other constant ::CONSTANT directives (where \textit{value} is omitted, a single literal string, a symbol that is taken as a constant, or a valid number optionally preceded by "-" or "+"). For a calculated ::CONSTANT directive to reference another calculated ::CONSTANT directive, the referenced directive must be defined earlier in order of appearance. Forward references to calculated ::CONSTANT directives are not allowed.

2. For a floating ::CONSTANT directive the constant \textit{value} must be a single literal string, or a symbol that is taken as a constant. An expression enclosed in parentheses is not allowed.

3. A ::CONSTANT directive cannot have a method body.

### 3.5. ::METHOD

The ::METHOD directive creates a method object and defines the method attributes.

A ::METHOD directive creates method objects that may be associated with a class instance. The created method may be from Rexx code, mapped to method in an external native library, or automatically generated. The type of method is determined by the combination of options specified.

The \textit{methodname} is a literal string or a symbol that is taken as a constant. The method is defined as \textit{methodname} in the class specified in the most recent ::CLASS directive. Only one ::METHOD directive can appear for any \textit{methodname} in a class.

A ::CLASS directive is not required before a ::METHOD directive. If no ::CLASS directive precedes ::METHOD, the method is not associated with a class but is accessible to the main (executable) part of a program through the \textit{.METHODS} built-in object. Only one ::METHOD directive can appear for any method name not associated with a class.

If you specify the \textbf{CLASS} option, the method is a class method. See Chapter 4, Objects and Classes. The method is associated with the class specified on the most recent ::CLASS directive. The ::METHOD directive must follow a ::CLASS directive when the \textbf{CLASS} option is used.

If \textbf{ABSTRACT} is specified, then the created method will be marked as an abstract method and will raise an error if directly invoked. For abstract methods there is no method code body following the directive, so another directive (or the end of the program) must follow the ::METHOD directive.
If `ABSTRACT`, `ATTRIBUTE`, or `EXTERNAL` is not specified, the `::METHOD` directive starts a section of method code which is ended by another directive or the end of the program. The `::METHOD` is not included in the source of the created METHOD object.

Example 3.10. METHOD examples

```
r = .rectangle~new(20,10)
say "Area is" r~area       /* Produces "Area is 200" */
::class rectangle
::method area              /* defined for the RECTANGLE class */
expose width height
return width*height
::method init
expose width height
use arg width, height
::method perimeter
expose width height
return (width+height)*2
```

If you specify the `ATTRIBUTE` option, method variable accessor methods are created. In addition to generating a method named `methodname`, another method named `methodname=` is created. The first method returns the value of object instance variable that matches the method name. The second method assigns a new value to the object instance variable.

For example, the directive

```
::method name attribute
```

creates two methods, `NAME` and `NAME=`, equivalent to the following code sequences:

```
::method name         -- attribute get method
expose name         -- establish direct access to object variable (attribute)
use strict arg      -- enforce zero parameters
return name         -- return object variable's current value
::method "NAME="      -- attribute set method
expose name         -- establish direct access to object variable (attribute)
use strict arg name -- retrieve argument and assign it to the object variable
```

Using the `ATTRIBUTE` option is equivalent to using the `::ATTRIBUTE` directive.

If you specify the `ABSTRACT` option, the method creates an ABSTRACT method placeholder. ABSTRACT methods define a method that an implementing subclass is expected to provide a concrete implementation for. Any attempt to invoke an ABSTRACT method directly will raise a SYNTAX condition.

If `DELEGATE` is specified, execution of method `name` is delegated to object `delegatename`. It is a common design pattern to delegate method execution to an embedded object. The directive for such a delegation

```
::method name delegate delegatename
```

is equivalent to the following code sequence:
Example 3.11. DELEGATE subkeyword equivalent code

```plaintext
::method name
  expose delegateName
  forward to(delegateName)
```

If the **EXTERNAL** option is specified, then `spec` identifies a method in an external native library that will be invoked as the named method. The `spec` is a literal string containing a series of whitespace delimited tokens defining the external method. The first token must be **LIBRARY**, which indicates the method resides in a native library of the type allowed on a ::REQUIRES directive. The second token must identify the name of the external library. The external library is located using platform-specific mechanisms for loading libraries. For Unix-like systems, the library name is case-sensitive. The third token is optional and specifies the name of the method within the library package. If not specified, the ::METHOD name is used. The target package method name is case insensitive.

Example 3.12. METHOD EXTERNAL examples

```plaintext
-- creates method INIT from method RegExp_Init in library rxregexp
::METHOD INIT EXTERNAL "LIBRARY rxregexp RegExp_Init"

-- creates method RegExp_Parse from library rxregexp
::METHOD RegExp_Parse EXTERNAL "LIBRARY rxregexp"
```

If the **ATTRIBUTE** option is specified with the **EXTERNAL** option, then two method objects need to be created. The target method name is appended to the string "GET" to derive the name of the getter attribute method. To generate the setter attribute method, the name is appended to the string "SET".

Example 3.13. METHOD EXTERNAL examples

```plaintext
-- maps "NAME" method to "GETNAME and
-- "NAME=" to "SETNAME"
::METHOD name ATTRIBUTE EXTERNAL "LIBRARY mylib"

-- maps "ADDRESS" method to "GETMyAddress and
-- "ADDRESS=" to "SETMyAddress"
::METHOD address ATTRIBUTE EXTERNAL "LIBRARY mylib MyAddress"
```

**Notes:**

1. You can specify all options in any order.

2. If you specify the **PACKAGE** option, the method is created with a package-scope, if you specify the **PRIVATE** option, the created method is a private method. Package-scope and private methods have restricted access rules on how they can be invoked. See Section 4.2.8, “Public, Package-Scope, and Private Methods” for details of how these methods can be used. If you omit the **PACKAGE** or **PRIVATE** option, or specify **PUBLIC**, the method is a public method that any sender can activate.

3. If you specify the **UNGUARDED** option, the method can be called while other methods are active on the same object. If you do not specify **UNGUARDED**, the method requires exclusive use of the
object variable pool; it can run only if no other method that requires exclusive use of the object variable pool is active on the same object.

4. If you specify the PROTECTED option, the method is a protected method. (See Chapter 13, The Security Manager for more information.) If you omit the PROTECTED option or specify UNPROTECTED, the method is not protected.

5. If you specify ATTRIBUTE, ABSTRACT, or EXTERNAL, another directive (or the end of the program) must follow the ::METHOD directive.

6. It is an error to specify ::METHOD more than once within the same class and use the same methodname.

3.6. ::OPTIONS

The ::OPTIONS directive defines default values for numeric, trace, and other runtime settings for all Rexx code contained within a package.

Any of the options may be specified on a single ::OPTIONS directive in any order. If an option is specified more than once, the last specified value will be the one used. If more than one ::OPTIONS directive appears in a source file, the options are processed in the order they appear and the effect is accumulative. If a given option type is specified on more than one directive, the last specified will be the value used.

The specified options will override the normal default settings for all Rexx code contained in the source file. For example,
would direct that all method and routine code defined in this source package execute with an initial NUMERIC DIGITS setting of 20 digits. The ::OPTIONS directive controls only the initial setting. A method or routine may change the current setting with the NUMERIC DIGITS instruction as normal. The values specified with ::OPTIONS only apply to code that appears in the same source file. It does not apply to code in other source files that may reference or use this code. For example, a subclass of a class defined in this source package will not inherit the ::OPTIONS settings if the subclass code is located in a different source package.

The following options may be specified on an ::OPTIONS directive:

**DIGITS** controls the precision to which arithmetic operations and built-in functions are evaluated. The value *digits* must be a symbol or string that is a valid positive whole number value and must be larger than the current FUZZ ::OPTIONS setting. The package value can be retrieved using the Package class *digits* method.

There is no limit to the value for DIGITS (except the amount of storage available), but high precisions are likely to require a great amount of processing time. It is recommended that you use the default value whenever possible.

**FORM** controls the form of exponential notation for the result of arithmetic operations and built-in functions. This can be either SCIENTIFIC (in which case only one, nonzero digit appears before the decimal point) or ENGINEERING (in which case the power of 10 is always a multiple of 3). The default is SCIENTIFIC. The subkeywords SCIENTIFIC or ENGINEERING must be specified as symbols. The package value can be retrieved using the Package class *form* method.

**FUZZ** controls how many digits, at full precision, are ignored during a numeric comparison operation. The value *fuzz* must be a symbol or string that is a valid positive whole number value and must be smaller than the current DIGIT ::OPTIONS setting. The package value can be retrieved using the Package class *fuzz* method.

NUMERIC FUZZ temporarily reduces the value of NUMERIC DIGITS by the NUMERIC FUZZ value during every numeric comparison. The numbers are subtracted under a precision of DIGITS minus FUZZ digits during the comparison and are then compared with 0.

**ALL** is a shortcut for all six ::OPTIONS condition directives ERROR, FAILURE, LOSTDIGITS, NOSTRING, NOTREADY, and NOVALUE.

::OPTIONS ALL SYNTAX sets all six conditions to raise SYNTAX, while ::OPTIONS ALL CONDITION sets all of them to their default state.

**ERROR** controls whether an ERROR, FAILURE, LOSTDIGITS, NOSTRING, NOTREADY, or NOVALUE condition event raises its associated condition, or raises a SYNTAX condition.

**FAILURE**

**LOSTDIGITS**

**NOSTRING**

**NOTREADY**

**NOVALUE**

If ::OPTIONS condition CONDITION is in effect, which is the default, the condition event raises its associated condition as usual. If ::OPTIONS condition SYNTAX is in effect, a SYNTAX condition is raised instead.

To override an ::OPTIONS condition SYNTAX package default, you can use SIGNAL ON condition, SIGNAL OFF condition, SIGNAL ON ANY, or SIGNAL OFF ANY, to raise the associated condition, or raise no condition at all. For conditions ERROR, FAILURE, and NOTREADY you can also use CALL ON condition, CALL OFF condition, CALL ON ANY, or CALL OFF ANY, to raise the associated condition, or raise no condition at all.
controls whether prolog code (any code in the source program that comes before the first directive) is run when another program requires it through a ::REQUIRES directive.

If ::OPTIONS PROLOG is in effect, any prolog code is run as usual when the source program is being required using a ::REQUIRES directive. If ::OPTIONS NOPROLOG is in effect, any prolog code is not run. The default is ::OPTIONS PROLOG.

TRACE controls the tracing action (that is, how much is displayed to the user) during the processing of all Rexx code contained in the package. Tracing describes some or all of the clauses in a program, producing descriptions of clauses as they are processed. TRACE is mainly used for debugging. The value trace must be one of the prefix or alphabetic character (word) options valid for the TRACE instruction. The package value can be retrieved using the Package class trace method.

3.7. ::REQUIRES

The ::REQUIRES directive specifies that the program requires access to the classes and objects of the Rexx program programname.

If the LIBRARY option is not specified, all public classes and routines defined in the named program are made available to the executing program. The programname is a literal string or a symbol that is taken as a constant. The string or symbol programname can be any string or symbol that is valid as the target of a CALL instruction. The program programname is called as an external routine with no arguments. The program is searched for using the external program search order.

If any Rexx code precedes the first directive in programname then that code is executed at the time the ::REQUIRES is processed by the interpreter. This will be executed prior to executing the main Rexx program in the file that specifies the ::REQUIRES statement.

If the LIBRARY option is specified, programname is the name of an external native library that is required by this program. The library will be loaded using platform-specific mechanisms, which generally means the library name is case sensitive. Any routines defined in the library will be made available to all programs running in the process. If the native library cannot be loaded, the program will not be permitted to run. All LIBRARY ::REQUIRES directives will be processed before ::REQUIRES for Rexx programs, which will ensure that the native libraries are available to the initialization code of the Rexx packages.

If the NAMESPACE option is specified, namespace must be a symbol that is taken as a constant; a literal string is not allowed.

The NAMESPACE option attaches the qualifier namespace to the package loaded by the ::REQUIRES directive. To distinguish public classes or public routines of the same name in different ::REQUIRES files, use a namespace-qualified symbol of the form namespace: class or namespace: routine. For details see Section 1.14, “Namespaces”.

::REQUIRES directives can be placed anywhere after the main section of code in the package. The order of ::REQUIRES directives determines the search order for classes and routines defined in the named programs and also the load order of the referenced files. Once a program is loaded by
a ::REQUIRES statement in a program, other references to that same program by ::REQUIRES statements in other programs will resolve to the previously loaded program. The initialization code for the ::REQUIRES file will only be executed on the first reference.

The following example illustrates that two programs, ProgramA and ProgramB, can both access classes and routines that another program, ProgramC, contains. (The code at the beginning of ProgramC runs prior to the start of the main Rexx program.)

```
/* ProgramA */
.
.
::Requires 'ProgramC'

/* ProgramB */
.
.
::Requires 'ProgramC'

/* ProgramC */
.
.
```

The language processor uses local routine definitions within a program in preference to routines of the same name accessed through ::REQUIRES directives. Local class definitions within a program override classes of the same name in other programs accessed through ::REQUIRES directives.

Another directive, or the end of the program, must follow a ::REQUIRES directive. Only null clauses can appear between them.

### 3.8. ::RESOURCE

The ::RESOURCE directive allows to include associated data directly in the program code.

```
::RESOURCE name
  resource_data
::END delimiter
```

The ::RESOURCE directive allows to include lines of `resource_data` of almost arbitrary form directly within the source program.

The resource `name` must be a symbol or a literal string that is taken as a constant. The optional resource end `delimiter` must be a symbol or a literal string that is taken as a constant. If `delimiter` is not specified, it defaults to the string "::END".
Example 3.14. RESOURCE directive

```
::resource greyCat end "-
La nuit, tous les chats sont gris
---------------------------------
::resource "brown fox"
The quick brown fox jumps over the lazy dog
::END
::resource nollop end ANONYMOUS
The wicked peon quivered,
then gazed balefully at the judges
  who examined him.
ANONYMOUS TYPESETTER
```

Notes:

1. Specifying more than one ::RESOURCE directive with the same resource name is an error.
2. The terminating resource end delimiter must start in the first column and is case-sensitive. Any text following on the same line as the terminating delimiter is ignored.
3. Although resource data may include almost arbitrary data (including any Rexx code), it is not well-suited for inclusion of binary data. Including special characters like line-end or end-of-file may cause unwanted results.
4. Resource data is accessible through the .RESOURCES built-in object, and the Package class methods resource and resources.

3.9. ::ROUTINE

The ::ROUTINE directive creates named routines within a program.

```
::ROUTINE routinename
```

The routinename is a literal string or a symbol that is taken as a constant. Only one ::ROUTINE directive can appear for any routinename in a program.

If the EXTERNAL option is not specified, the ::ROUTINE directive starts a routine, which is ended by another directive or the end of the program.

If you specify the PUBLIC option, the routine is visible beyond its containing Rexxx program to any other program that references this program with a ::REQUIRES directive. If you do not specify the PUBLIC option, or specify the PRIVATE option, the routine is visible only within its containing Rexxx program.

Routines you define with the ::ROUTINE directive behave like external routines. In the search order for routines, they follow internal routines and built-in functions but precede all other external routines.
Example 3.15. ROUTINE examples

```plaintext
::class c
::method a
call r "A" /* displays "In method A" */

::method b
call r "B" /* displays "In method B" */

::routine r
use arg name
say "In method" name
```

If the EXTERNAL option is specified, then `spec` identifies a routine in an external native library that will be defined as the named routine for this program. The `spec` is a literal string containing a series of whitespace delimited tokens defining the external function. The first token identifies the type of native routine to locate:

**LIBRARY**

Identifies a routine in an external native library of the type supported by the ::REQUIRES directive. The second token must identify the name of the external library. The external library is located using platform-specific mechanisms for loading libraries. For Unix-like systems, the library name is case-sensitive. The third token is optional and specifies the name of the routine within the library package. If not specified, the ::ROUTINE name is used. The routine name is not case sensitive.

**REGISTERED**

Identifies a routine in an older-style Rexx function package. The second token must identify the name of the external library. The external library is located using platform-specific mechanisms for loading libraries. For Unix-like systems, the library name is case-sensitive. The third token is optional and specifies the name of the function within the library package. If not specified, the ::ROUTINE name is used. Loading of the function will be attempted using the name as given and as all uppercase. Using REGISTERED is the equivalent of loading an external function using the RXFUNCADD built-in function.

Example 3.16. ROUTINE EXTERNAL examples

```plaintext
-- load a function from rxmath library
::routine RxCalcPi external "LIBRARY rxmath"
-- same function, but a different internal name
::routine Pi external "LIBRARY rxmath RxCalcPi"
-- same as call rxfuncadd "SQLLoadFuncs", "rexxsql", "SQLLoadFuncs"
::routine SQLLoadFuncs EXTERNAL "REGISTERED rexxsql SQLLoadFuncs"
```

Notes:

1. It is an error to specify ::ROUTINE with the same routine name more than once in the same program. It is not an error to have a local ::ROUTINE with the same name as another ::ROUTINE in another program that the ::REQUIRES directive accesses. The language processor uses the local ::ROUTINE definition in this case.

2. Calling an external Rexx program as a function is similar to calling an internal routine. For an external routine, however, the caller's variables are hidden and the internal values (NUMERIC settings, for example) start with their defaults.
Note

If you specify the same ::ROUTINE routinename more than once in different programs, the last one is used. Using more than one ::ROUTINE routinename in the same program produces an error.
Objects and Classes

This chapter provides an overview of the Rexx class structure.

A Rexx object consists of object methods and object variables ("attributes"). Sending a message to an object causes the object to perform some action; a method whose name matches the message name defines the action that is performed. Only an object's methods can access the object variables belonging to an object. EXPOSE instructions within an object's methods specify which object variables the methods will use. Any variables not exposed are local to the method and are dropped on return from a method.

You can create an object by sending a message to a class object—typically a "new" method. An object created from a class is an instance of that class. The methods a class defines for its instances are called the instance methods of that class. These are the object methods that are available for every instance of the class. Classes can also define class methods, which are a class's own object methods.

---

Note

When referring to instance methods (for objects other than classes) or class methods (for classes), this book uses the term methods when the meaning is clear from the context. When referring to instance methods and class methods of classes, this book uses the qualified terms to avoid possible confusion.

4.1. Types of Classes

There are four kinds of classes:

- Object classes
- Mixin classes
- Abstract classes
- Metaclasses

The following sections explain these.

4.1.1. Object Classes

An object class is a factory for producing objects. An object class creates objects (instances) and provides methods that these objects can use. An object acquires the instance methods of the class to which it belongs at the time of its creation. If a class gains additional methods, objects created before the definition of these methods do not acquire the new or changed methods.

The instance variables within an object are created on demand whenever a method EXPOSEs an object variable. The class creates the object instance, defines the methods the object has, and the object instance completes the job of constructing the object.

The String class and the Array class are examples of object classes.
4.1.2. Mixin Classes

Classes can inherit from more than the single superclass from which they were created. This is called *multiple inheritance*. Classes designed to add a set of instance and class methods to other classes are called *mixin classes*, or simply mixins.

You can add mixin methods to an existing class by sending an INHERIT message or using the INHERIT option on the ::CLASS directive. In either case, the class to be inherited must be a mixin. During both class creation and multiple inheritance, subclasses inherit both class and instance methods from their superclasses.

Mixins are always associated with a *base class*, which is the mixin's first non-mixin superclass. Any subclass of the mixin's base class can (directly or indirectly) inherit a mixin; other classes cannot. For example, a mixin class created as a subclass of the Array class can only be inherited by other Array subclasses. Mixins that use the Object class as a base class can be inherited by any class.

To create a new mixin class, you send a MIXINCLASS message to an existing class or use the ::CLASS directive with the MIXINCLASS option. A mixin class is also an object class and can create instances of the class.

4.1.3. Abstract Classes

*Abstract classes* provide definitions for instance methods and class methods but are not intended to create instances. Abstract classes often define the message interfaces that subclasses should implement.

You create an abstract class by specifying the ABSTRACT subkeyword on the ::CLASS directive. Trying to create an instance from an abstract class will result in an error.

It is possible to create abstract methods or attributes on a class. Abstract methods or attributes are placeholders that subclasses are expected to override. Failing to provide a real method or attribute implementation will result in an error when the abstract version is called.

4.1.4. Metaclasses

A *metaclass* is a class you can use to create another class. The Class class is the metaclass of all the classes Rexx provides. This means that instances of .Class are themselves classes. The Class class is like a factory for producing the factories that produce objects.

To change the behavior of an object that is an instance, you generally use subclassing. For example, you can create Statarray, a subclass of the *Array class*. The statArray class can include a method for computing a total of all the numeric elements of an array.

Example 4.1. Creating an array subclass

```rexx
/* Creating an array subclass for statistics */
::class statArray subclass array public
::method init /* Initialize running total and forward to superclass */
  expose total
  total = 0
  forward class (super)
```
You can use this method on the individual array instances, so it is an instance method.

However, if you want to change the behavior of the factory producing the arrays, you need a new class method. One way to do this is to use the ::METHOD directive with the CLASS option. Another way to add a class method is to create a new metaclass that changes the behavior of the Statarray class. A new metaclass is a subclass of .class.

You can use a metaclass by specifying it in a SUBCLASS or MIXINCLASS message or on a ::CLASS directive with the METACLASS option.

If you are adding a highly specialized class method useful only for a particular class, use the ::METHOD directive with the CLASS option. However, if you are adding a class method that would be useful for many classes, such as an instance counter that counts how many instances a class creates, you use a metaclass.

The following examples add a class method that keeps a running total of instances created. The first version uses the ::METHOD directive with the CLASS option. The second version uses a metaclass.

**Version 1**

```plaintext
::method put /* Modify to increment running total */
expose total
use arg value
total = total + value /* Should verify that value is numeric!!! */
forward class (super)
::method "[]=" /* Modify to increment running total */
forward message "PUT"
::method remove /* Modify to decrement running total */
expose total
use arg index
forward message "AT" continue
total = total - result
forward class (super)
::method average /* Return the average of the array elements */
expose total
return total / self~items
::method total /* Return the running total of the array elements */
expose total
return total
```

Example 4.2. Adding a CLASS method

```plaintext
/* Adding a class method using ::METHOD */
a = .point~new(1,1) /* Create some point instances */
say "Created point instance" a
b = .point~new(2,2) /* create another point instance */
say "Created point instance" b
c = .point~new(3,3) /* create another point instance */
say "Created point instance" c
/* ask the point class how many */
/ * instances it has created */
say "The point class has created" .point~instances "instances."
```
::class point public /* create Point class */
::method init class expose instanceCount
  instanceCount = 0 /* Initialize instanceCount */
  forward class (super) /* Forward INIT to superclass */
::method new class expose instanceCount /* Creating a new instance */
  instanceCount = instanceCount + 1 /* Bump the count */
  forward class (super) /* Forward NEW to superclass */
::method instances class expose instanceCount /* Return the instance count */
  return instanceCount
::method init expose xVal yVal /* Set object variables */
  use arg xVal, yVal /* as passed on NEW */
::method string expose xVal yVal /* Use object variables */
  return "("xVal","yVal")" /* to return string value */

Version 2

/* Adding a class method using a metaclass */
a = .point~new(1,1) /* Create some point instances */
say "Created point instance" a
b = .point~new(2,2)
say "Created point instance" b
c = .point~new(3,3)
say "Created point instance" c

::class InstanceCounter subclass class /* Create a new metaclass that */
  /* will count its instances */
::method init expose instanceCount
  instanceCount = 0 /* Initialize instanceCount */
  forward class (super) /* Forward INIT to superclass */
::method new expose instanceCount /* Creating a new instance */
  instanceCount = instanceCount + 1 /* Bump the count */
  forward class (super) /* Forward NEW to superclass */
::method instances expose instanceCount /* Return the instance count */
  return instanceCount
::class point public metaclass InstanceCounter /* Create Point class */
  /* using InstanceCounter metaclass */
::method init expose xVal yVal /* Set object variables */
  use arg xVal, yVal /* as passed on NEW */
::method string expose xVal yVal /* Use object variables */
4.2. Creating and Using Classes and Methods

You can define a class using either directives or messages.

To define a class using directives, you place a ::CLASS directive after the main part of your source program:

```
::class "Account"
```

This creates an Account class that is a subclass of the Object class. Object is the default superclass if one is not specified. See Section 5.1.4, “Object Class” for details. The string "Account" is a string identifier for the new class. The string identifier is both the internal class name and the name of the environment symbol used to locate your new class instance.

Now you can use ::METHOD directive to add methods to your new class. The ::METHOD directives must immediately follow the ::CLASS directive that creates the class.

**Example 4.3. Adding a method**

```
::method type
    return "an account"
::method "name="
    expose name
    use arg name
::method name
    expose name
    return name
```

This adds the methods TYPE, NAME, and NAME= to the Account class.

You can create a subclass of the Account class and define a method for it:

**Example 4.4. Adding a method**

```
::class "Savings" subclass account
::method type
    return "a savings account"
```

Now you can create an instance of the Savings class with the new method and send TYPE, NAME, and NAME= messages to that instance:

**Example 4.5. Invoking a method**

```
asav = .savings~new
say asav~type
asav~name = "John Smith"
```

The Account class methods NAME and NAME= create a pair of access methods to the account object variable NAME. The following directive sequence creates the NAME and NAME= methods:
Example 4.6. Defining SET and GET methods

```plaintext
::method "name="
  expose name
  use arg name
::method name
  expose name
  return name
```

You can replace this with a single `::ATTRIBUTE` directive. For example, the directive

```plaintext
::attribute name
```

adds two methods, `NAME` and `NAME=` to a class. These methods perform the same function as the `NAME` and `NAME=` methods in the original example. The `NAME` method returns the current value of the object variable `NAME`; the `NAME=` method assigns a new value to the object variable `NAME`.

In addition to defining operational methods and attribute methods, you can add “constant” methods to a class using the `::CONSTANT` directive. The `::CONSTANT` directive will create both a class method and an instance method to the class definition. The constant method will always return the same constant value, and can be invoked by sending a message to either the class or an instance method. For example, you might add the following constant to your Account class:

```plaintext
::constant checkingMinimum 200
```

This value can be retrieved using either of the following methods

Example 4.7. Retrieving method values

```plaintext
say .Account~checkingMinimum    -- displays "200"
asave = .savings~new
say asave~checkingMinimum       -- also displays "200"
```

4.2.1. Using Classes

When you create a new class, it is always a subclass of an existing class. You can create new classes with the `::CLASS` directive or by sending the SUBCLASS or MIXINCLASS message to an existing class. If you specify neither the SUBCLASS nor the MIXINCLASS option on the `::CLASS` directive, the superclass for the new class is the Object class, and it is not a mixin class.

Example of creating a new class using a message:

```plaintext
persistence = .object~mixinclass("Persistence")
myarray=.array~subclass("myarray")~~inherit(persistence)
```

Example of creating a new class using the directive:

```plaintext
::class persistence mixinclass object
::class myarray subclass array inherit persistence
```
4.2.2. Scope

A scope refers to the methods and object variables defined for a single class (not including the superclasses). Only methods defined in a particular scope can access the object variables within that scope. This means that object variables in a subclass can have the same names as object variables used by a superclass, because the variables are created at different scopes.

4.2.3. Defining Instance Methods with SETMETHOD or ENHANCED

In Rexx, methods are usually associated with instances using classes, but it is also possible to add methods directly to an instance using the `setMethod` or `enhanced` method.

All subclasses of the Object class inherit SETMETHOD. You can use SETMETHOD to create one-off objects, objects that must be absolutely unique so that a class that is capable of creating other instances is not necessary. The Class class also provides an ENHANCED method that lets you create new instances of a class with additional methods. The methods and the object variables defined on an object with SETMETHOD or ENHANCED form a separate scope, like the scopes the class hierarchy defines.

4.2.4. Method Names

A method name can be any string. When an object receives a message, the language processor searches for a method whose name matches the message name in uppercase.

Note

The language processor also translates the specified name of all methods added to objects into uppercase characters.

You must surround a method name with quotation marks when it contains characters that are not allowed in a symbol (for example, the operator characters). The following example creates a new class (the Cost class), defines a new method (%), creates an instance of the Cost class (mycost), and sends a % message to mycost:

Example 4.8. Accessing a method

```rexx
cost=.object~subclass("A cost")
cost~define("%", 'expose p; say "Enter a price."; pull p; say p*1.07;')
mycost=cost~new
mycost~"%" /* Produces: Enter a price. */
/* If the user specifies a price of 100, */
/* produces: 107.00 */
```

4.2.5. Default Search Order for Method Selection

The search order for a method name matching the message is for:

1. A method the object itself defines with `setMethod` or `enhanced`. 
2. A method the object's class defines. (Note that an object acquires the instance methods of the class to which it belongs at the time of its creation. If a class gains additional methods, objects created before the definition of these methods do not acquire these methods.)

3. A method that a superclass of the object's class defines. This is also limited to methods that were available when the object was created. The order of the `inherit` messages sent to an object's class determines the search order of the superclass method definitions.

This search order places methods of a class before methods of its superclasses so that a class can supplement or override inherited methods.

If the language processor does not find a match for the message name, the language processor checks the object for a method name `UNKNOWN`. If it exists, the language processor calls the `UNKNOWN` method and returns as the message result any result the `UNKNOWN` method returns. The `UNKNOWN` method arguments are the original message name and a Rexx array containing the original message arguments.

If the object does not have an `UNKNOWN` method, the language processor raises a `NOMETHOD` condition. If there are no active traps for the `NOMETHOD` condition, a syntax error is raised.

### 4.2.6. Defining an `UNKNOWN` Method

When an object that receives a message does not have a matching message name, the language processor checks if the object has a method named `UNKNOWN`. If the object has an `UNKNOWN` method, the language processor calls `UNKNOWN`, passing two arguments. The first argument is the name of the method that was not located. The second argument is an array containing the arguments passed with the original message.

For example, the following `UNKNOWN` method will print out the name of the invoked method and then invoke the same method on another object. This can be used to track the messages that are sent to an object:

```rxml
::method unknown
expose target      -- will receive all of the messages
use arg name, arguments
say name "invoked with" arguments~toString
-- send along the message with the original args
forward to(target) message(name) arguments(arguments)
```

### 4.2.7. Changing the Search Order for Methods

You can change the usual search order for methods by specifying a colon and a class symbol after the message name. The class symbol can be a variable name or an environment symbol. It identifies the class object to be used as the starting point for the method search.

The class object must be a superclass of the class defining the active method, or, if you used `setMethod` to define the active method, the object's own class. The class symbol is usually the special variable `SUPER`, but it can be any environment symbol or variable name whose value is a valid class.

Suppose you create an Account class that is a subclass of the Object class, define a `TYPE` method for the Account class, and create the Savings class that is a subclass of Account. You could define a `TYPE` method for the Savings class as follows:
savings-define("TYPE", 'return "a savings account"')

You could change the search order by using the following line:

savings-define("TYPE", 'return self-type:super "(savings)"')

This changes the search order so that the language processor searches for the TYPE method first in the Account superclass (rather than in the Savings subclass). When you create an instance of the Savings class (asav) and send a TYPE message to asav:

say asav-type

an account (savings) is displayed. The TYPE method of the Savings class calls the TYPE method of the Account class, and adds the string (savings) to the results.

4.2.8. Public, Package-Scope, and Private Methods

A method can be public, package-scope, or private.

Any object can send a message that runs a public method.

A package-scope method can only be invoked from methods or routines defined in the same package as the package-scope method.

A private method can only be invoked from specific calling contexts. These contexts are:
1. From within a method owned by the same class as the target. This is frequently the same object, accessed via the special variable SELF. Private methods of an object can also be accessed from other instances of the same class (or subclass instances).
2. From within a method defined at the same class scope as the method. For example:

Example 4.10. Referencing methods

```plaintext
::class Savings
::method newCheckingAccount CLASS
  instance = self-new
  instance-makeChecking
  return instance

::method makeChecking private
  expose checking
  checking = .true
```

The newCheckingAccount CLASS method is able to invoke the makeChecking method because the scope of the makeChecking method is .Savings.

3. From within an instance (or subclass instance) of a class to a private class method of its class. For example:

Example 4.11. Referencing methods

```plaintext
::class Savings
::method init class
```
expose counter
counter = 0

::method allocateAccountNumber private class
expose counter
  counter = counter + 1
  return counter

::method init
  expose accountNumber
  accountNumber = self-class-allocateAccountNumber

The instance init method of the Savings class is able to invoke the allocateAccountNumber private method of the .Savings class object because it is owned by an instance of the .Savings class.

Private methods include methods at different scopes within the same object. This allows superclasses to make methods available to their subclasses while hiding those methods from other objects. A private method is like an internal subroutine. It shields the internal information of an object to outsiders, but allowing objects to share information with each other and their defining classes.

4.2.9. Initialization

Any object requiring initialization at creation time must define an INIT method. If this method is defined, the class object runs the INIT method after the object is created. If an object has more than one INIT method (for example, it is defined in several classes), each INIT method must forward the INIT message up the hierarchy to complete the object's initialization.

Example 4.12. Instance initialization

```plaintext
asav = .savings~new(1000.00, 6.25)
say asav~type
asav~name = "John Smith"

::class Account

::method INIT
  expose balance
  use arg balance

::method TYPE
  return "an account"

::method name attribute

::class Savings subclass Account

::method INIT
  expose interest_rate
  use arg balance, interest_rate
  self~init:super(balance)

::method type
  return "a savings account"
```

The NEW method of the Savings class object creates a new Savings object and calls the INIT method of the new object. The INIT method arguments are the arguments specified on the NEW method. In the Savings INIT method, the line:
Example 4.13. Instance initialization

```
self-init:super(balance)
```

calls the INIT method of the Account class, using just the balance argument specified on the NEW message.

### 4.2.10. Object Destruction and Uninitialization

Object destruction is implicit. When an object is no longer in use, Rexx automatically reclaims its storage. If the object has allocated other system resources, you must release them at this time. (Rexx cannot release these resources, because it is unaware that the object has allocated them.)

Similarly, other uninitialization processing may be needed, for example, by a message object holding an unreported error. An object requiring uninitialization should define an UNINIT method. If this method is defined, Rexx runs it before reclaiming the object's storage. If an object has more than one UNINIT method (defined in several classes), each UNINIT method is responsible for sending the UNINIT method up the object hierarchy.

### 4.2.11. Required String Values

Rexx requires a string value in a number of contexts within instructions and built-in function calls.

- DO statements containing `expr` or `exprf`
- Substituted values in compound variable names
- Commands to external environments
- Commands and environment names on ADDRESS instructions
- Strings for ARG, PARSE, and PULL instructions to be parsed
- Parenthesized targets on CALL instructions
- Subsidiary variable lists on DROP, EXPOSE, and PROCEDURE instructions
- Instruction strings on INTERPRET instructions
- DIGITS, FORM, and FUZZ values on NUMERIC instructions
- Options strings on OPTIONS instructions
- Data queue strings on PUSH and QUEUE instructions
- Label names on SIGNAL VALUE instructions
- Trace settings on TRACE VALUE instructions
- Arguments to built-in functions
- Variable references in parsing templates
- Data for PUSH and QUEUE instructions to be processed
- Data for the SAY instruction to be displayed
• Rexx dyadic operators when the receiving object (the object to the left of the operator) is a string

If you supply an object other than a string in these contexts, by default the language processor converts it to some string representation and uses this. However, the programmer can cause the language processor to raise the NOSTRING condition when the supplied object does not have an equivalent string value.

To obtain a string value, the language processor sends a `request("STRING")` message to the object. Strings and other objects that have string values return the appropriate string value for Rexx to use. (This happens automatically for strings and for subclasses of the String class because they inherit a suitable makeString method from the String class.) For this mechanism to work correctly, you must provide a makeString method for any other objects with string values.

For other objects without string values (that is, without a makeString method), the action taken depends on the setting of the NOSTRING condition trap. If the NOSTRING condition is being trapped (see Chapter 11, Conditions and Condition Traps), the language processor raises the NOSTRING condition. If the NOSTRING condition is not being trapped, the language processor sends a `string` message to the object to obtain its readable string representation and uses this string.

Example 4.14. Comparing to the .nil object

```rxml
d = .directory-new
say substr(d,5,7) /* Produces "rectory" from "a Directory" */
signal on nostring
say substr(d,5,7) /* Raises the NOSTRING condition */
say substr(d~string,3,6) /* Displays "Direct" */
```

For arguments to Rexx object methods, different rules apply.

For String arithmetic, comparison, and concatenation methods:

These methods always require a string argument, so first a `request("STRING")` message is sent to the argument object. If `request` returns `.nil` because the argument object does not have a makeString method, and the NOSTRING condition is not being trapped, a `string` message is sent to the object to obtain its string representation.

For all other methods:

When a method expects a string as an argument, the argument object is sent the `request("STRING")` message. If `request` returns `.nil`, the method raises an error.

4.2.12. Concurrency

Rexx supports concurrency, multiple methods running simultaneously on a single object. See Chapter 12, Concurrency for a full description of concurrency.

4.3. Overview of Classes Provided by Rexx

This section gives a brief overview of the classes and methods Rexx defines.

4.3.1. The Class Hierarchy

Rexx provides the following classes belonging to the `Object class`.

The classes are in a class hierarchy with an inheriting class indented below its superclass or mixin class. Classes inheriting from multiple mixin classes are only listed below one of these mixin classes.

`Alarm class`
The Class Hierarchy

**AlarmNotification class**
**Buffer class**
**Class class**
**Collection class**
  **MapCollection class**
    **Bag class**
    **Directory class**
      **Properties class**
    **IdentityTable class**
    **Relation class**
    **Set class**
    **Stem class**
    **StringTable class**
    **Table class**
  **OrderedCollection class**
    **Array class**
    **List class**
    **Queue class**
      **CircularQueue class**
  **SetCollection class**
**Comparable class**
**DateTime class**
**File class**
**String class**
**TimeSpan class**
**Comparator class**
  **CaselessColumnComparator class**
  **CaselessComparator class**
  **CaselessDescendingComparator class**
  **ColumnComparator class**
  **DescendingComparator class**
  **InvertingComparator class**
  **NumericComparator class**
**InputStream class**
  **InputOutputStream class**
    **Stream class**
**MessageNotification class**
  **Message class**
**Method class**
**Monitor class**
**MutableBuffer class**
**Object class**
**Orderable class**
**OutputStream class**
**Package class**
**Pointer class**
**RexxContext class**
**RexxInfo class**
**RexxQueue class**
**Routine class**
**StackTrace class**
**Supplier class**
  **StreamSupplier class**
**Ticker class**
Validate class
VariableReference class
WeakReference class

Note that there might also be other classes available, depending on the operating system. Additional classes may be accessed by using an appropriate `::requires` directive to load the class definitions.

4.3.2. Class Library Notes
The chapters that follow describe the classes and other objects that Rexx provides and their available methods. Rexx provides the objects listed in these sections and they are generally available through environment symbols.

Notes:
1. In the method descriptions in the chapters that follow, methods that return a result begin with the word "returns".
2. For `[]` and `[]=` methods, the syntax diagrams include the index or indexes within the brackets. These diagrams are intended to show how you can use these methods. For example, to set the element `(2, 3)` of a multi-dimensional Array named `matrix` to `0`, you would typically use the syntax:

   ```
   matrix[2, 3] = 0
   ```

   rather than:

   ```
   matrix~"[]="(0, 2, 3)
   ```

   even though the latter is valid and equivalent. For more information, see Section 1.11.4, "Message Terms" and Section 1.12.6, "Message Instructions".
3. When the argument of a method must be a specific kind of object (such as array, class, method, or string) the variable you specify must be of the same class as the required object or be able to produce an object of the required kind in response to a conversion message. In particular, subclasses are acceptable in place of superclasses (unless overridden in a way that changes superclass behavior), because they inherit a suitable conversion method from their Rexx superclass.

   The `isA` method of the Object class can perform this validation.
Built-in Classes

This chapter describes all of the Rexx built-in classes.

Fundamental Classes
These classes are the fundamental building blocks for all other classes.
- **Object** and **Class** class,
- **String Class**,
- **Method**, **Routine**, and **Package** class, and
- **Message** class.

Stream Classes
This set of classes implements Rexx data streams. They consist of **InputStream**, **OutputStream**, **InputOutputStream**, and **Stream** class.

Collection Classes
This set of classes implements object collections. It includes
- Ordered collections **Array**, **List**, **Queue**, and **CircularQueue** class,
- Map collections **Directory**, **StringTable**, **Stem**, **Table**, **IdentityTable**, **Relation**, and **Properties** class, and
- Map/Set collections **Bag** and **Set** class.

Utility Classes
This set of classes consists of
- **MutableBuffer Class**,
- **File Class**,
- Date-, time-, and timing-related classes **DateTime**, **TimeSpan**, **Alarm**, **Ticker**, and the notification classes **AlarmNotification** and **MessageNotification**,
- the synchronization classes **EventSempahore** and **MutexSempahore**,
- **Comparable** class and **Orderable** class,
- eight **Comparator** classes used for sorting **(Caseless)Comparator**, **(Caseless)ColumnComparator**, **(Caseless)DescendingComparator**, **InvertingComparator**, and **NumericComparator**, and

5.1. Fundamental Classes

This section describes the Rexx fundamental classes.
- **Object** and **Class** class,
- **String Class**,
- **Method**, **Routine**, and **Package** class, and
- **Message** class.

5.1.1. Class Class (Metaclass)

The Class class is like a factory that produces the factories that produce objects. It is a subclass of the **Object class**. The instance methods of the Class class are also the class methods of all classes.

Note that the copy method is forbidden for Class and all other class objects, and will result in an error.
Table 5.1. Class Class

<table>
<thead>
<tr>
<th>Methods inherited from the Object class</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison Methods</td>
<td>= == &lt;&gt; &gt;&lt; = ==</td>
</tr>
</tbody>
</table>

activate  id  new
annotation  inherit  package
annotations  isAbstract  queryMixinClass
baseClass  isMetaclass  subclass
defaultName  isSubclassOf  subclasses
define  metaClass  superClass
defineMethods  method  superClasses
delete  methods  uninherit
ehanced  mixinClass

5.1.1.1. Comparison Methods

Returns `.true` or `.false`, the result of performing a specified comparison operation.

For the Class class, if `argument` is the same class as the receiver class, the result is `.true`, otherwise `.false` is returned.

The comparison operators you can use in a message are:

`=, ==
  `.true` if the terms are the same class.

`\=, >, <, \<, \>\<, \>=`
  `.true` if the terms are not the same class (inverse of =).

5.1.1.2. activate

Completes initialization of a class object created from a `::CLASS` directive. The activate method is called after all classes in a package have been created and made available, but before the main portion of the program starts to execute. Activate is called for each class in the package in their construction order. The class object is fully constructed and capable of creating new instances of the class. All other classes in the same package are also available, although other classes might not have been activated yet. Because the INIT method is called early in the class construction process, only limited class initialization is possible at that time. The activate method is the preferred method for initializing a class object.

5.1.1.3. annotation

Completes initialization of a class object created from a `::CLASS` directive. The activate method is called after all classes in a package have been created and made available, but before the main portion of the program starts to execute. Activate is called for each class in the package in their construction order. The class object is fully constructed and capable of creating new instances of the class. All other classes in the same package are also available, although other classes might not have been activated yet. Because the INIT method is called early in the class construction process, only limited class initialization is possible at that time. The activate method is the preferred method for initializing a class object.
Returns the value of the annotation named name for this class. If no such annotation exists, .nil is returned.

See also
• method annotations and
• ::ANNOTATE directive.

5.1.1.4. annotations

Returns a StringTable of all annotation name/value pairs for this class.

See also
• method annotation and
• ::ANNOTATE directive.

5.1.1.5. baseClass

Returns the base class associated with the class. If the class is a mixin class, the base class is the first superclass that is not also a mixin class. If the class is not a mixin class, the base class is the class receiving the baseClass message.

5.1.1.6. defaultName

Returns a short human-readable string representation of the class. The string returned is of the form

The id class

where id is the identifier assigned to the class when it was created.

Example 5.1. Class class — defaultName method

```ruby
say .array-defaultName  /* Displays "The Array class" */
say .account-defaultName /* Displays "The ACCOUNT class" */
say .savings-defaultName /* Displays "The Savings class" */
::class account        /* Name is all upper case */
::class "Savings"      /* String name is mixed case */
```

5.1.1.7. define
Incorporates the method object *method* in the receiver class's collection of instance methods. The method name *methodname* is translated to uppercase. Using the *define* method replaces any existing definition for *methodname* in the receiver class.

If you omit *method*, the method name *methodname* is made unavailable for the receiver class. Sending a message of that name to an instance of the class causes the unknown method (if any) to be run.

The *method* argument can be a string containing a method source line instead of a method object. Alternatively, you can pass an array of strings containing individual method lines. Either way, *define* creates an equivalent method object.

**Notes:**

1. The classes Rexx provides do not permit changes or additions to their method definitions.
2. The *define* method is a protected method.

**Example 5.2. Class class — define method**

```plaintext
bank_account=.object~subclass("Account")
bank_account~define("TYPE","return "a bank account"")
```

**5.1.1.8. defineMethods**

Incorporates all methods *methods* in the receiver class's collection of instance methods.

The *methods* is a collection whose indexes are the names of methods and whose items are method objects (or strings or arrays of strings containing method code). The method names are translated to uppercase. Any existing methods with the same method name in the receiver class are replaced.

If, for any item in *methods*, the method name is `.nil`, then a method of this name in the receiver class is made unavailable.

See also method *define*.

**Example 5.3. Class class — defineMethods method**

```plaintext
before = .c~new
say before-a -- a-original
.c~defineMethods(.methods)
after = .c~new
say after-a after-b -- a-define b-define
::method a
return "a-define"
::method b
return "b-define"
::class c
::method a
```
5.1.1.9. delete

```
delete(methodname)
```

Removes the receiver class's definition for the method name `methodname`. If the receiver class defined `methodname` as unavailable with the `define` method, this definition is nullified. If the receiver class had no definition for `methodname`, no action is taken.

Notes:

1. The classes Rexx provides do not permit changes or additions to their method definitions.
2. delete deletes only methods the target class defines. You cannot delete inherited methods the target's superclasses define.
3. The delete method is a protected method.

Example 5.4. Class class — delete method

```
myclass=.object~subclass("Myclass")        /* After creating a class */
myclass~define("TYPE","return "my class"") /* and defining a method */
myclass~delete("TYPE")                     /* this deletes the method */
```

5.1.1.10. enhanced

```
enhanced(methods, argument)
```

Returns an enhanced new instance of the receiver class, with object methods that are the instance methods of the class, enhanced by the methods in the collection `methods`. The collection indexes are the names of the enhancing methods, and the items are the method objects (or strings or arrays of strings containing method code). You can use any collection that supports a `supplier` method.

enhanced sends an `init` message to the created object, passing the `arguments` specified on the enhanced method.

See also method `define`.

Example 5.5. Class class — enhanced method

```
/* Set up rclass with class method or methods you want in your */
/* remote class */
rclassmeths = .directory~new
rclassmeths["DISPATCH"] = d_source       /* d_source must have code for a */
                           /* DISPATCH method. */
/* The following sends init("Remote Class") to a new instance */
5.1.11. id

Returns the class identity (instance) string. (This is the string that is an argument on the subclass and mixinClass methods.) The string representations of the class and its instances contain the class identity.

Example 5.6. Class class — id method

```
myobject=.object~subclass("my object") /* Creates a subclass */
say myobject~id /* Produces: "my object" */
```

5.1.12. inherit

Causes the receiver class to inherit the instance and class methods of the class object classobj. The classpos is a class object that specifies the position of the new superclass in the list of superclasses. (You can use the superClasses method to return the immediate superclasses.)

The new superclass is inserted in the search order after the specified class. If the classpos class is not found in the set of superclasses, an error is raised. If you do not specify classpos, the new superclass is added to the end of the superclasses list.

Inherited methods can take precedence only over methods defined at or above the base class of the classobj in the class hierarchy. Any subsequent change to the instance methods of classobj takes immediate effect for all the classes that inherit from it.

The new superclass classobj must be created with the MIXINCLASS option of the ::CLASS directive or the mixinClass method and the base class of the classobj must be a direct superclass of the receiver object. The receiver must not already descend from classobj in the class hierarchy and vice versa.

The method search order of the receiver class after inherit is the same as before inherit, with the addition of classobj and its superclasses (if not already present).

Notes:
1. You cannot change the classes that Rexx provides by sending inherit messages.
2. The inherit method is a protected method.

Example 5.7. Class class — inherit method

```
room-inherit(.location)
```
5.1.13. isAbstract

Returns `.true.` if the receiving class is an abstract class, otherwise returns `.false.`. See also ABSTRACT option of the `::CLASS` directive.

Example 5.8. Class class — isAbstract method

```rxml
say .abs~isAbstract                    -- 1
::class abs abstract
```

5.1.14. isMetaclass

Returns `.true.` if the receiving class is a metaclass, otherwise returns `.false.`. The Class class is the only metaclass that Rexx provides; any subclasses of the Class class are also metaclasses.

Example 5.9. Class class — isMetaclass method

```rxml
do class
over .RexxInfo~package~classes~allItems~appendAll(.context~package~classes~allItems)
if class~isMetaclass then
  say class~string": metaclass"      -- The Class class: metaclass
  say class~string": metaclass"      -- The META class: metaclass
end
::class meta subclass class
::class other
```

5.1.15. isSubclassOf

Returns `.true.` if the object is a subclass of the specified `class`. Returns `.false.` if the object is not a subclass of the specified `class`. A class is a subclass of a class if the target class is the same as `class` or if `class` is in the object's direct or mixin class inheritance chain. For example:

Example 5.10. Class class — isSubclassOf method

```rxml
.String~isSubclassOf(.object)             ->  1
.String~isSubclassOf(.mutablebuffer)      ->  0
```

5.1.16. metaClass

```rxml
```
Returns the receiver class's default metaclass. This is the class used to create subclasses of this class when you send `subclass` or `mixinClass` messages (with no metaclass arguments). The instance methods of the default metaclass are the class methods of the receiver class.

For more information about class methods, see Section 4.1.1, "Object Classes". See also the description of method `subclass`.

**5.1.1.17. method**

Returns the method object for the receiver class's definition for the method name `methodname`. If the receiver class defined `methodname` as unavailable, this method returns `.nil`. If the receiver class did not define `methodname`, an error is raised.

**Example 5.11. Class class — method method**

```scheme
/* Create and retrieve the method definition of a class */
myclass=.object~subclass("My class")   /* Create a class */
mymethod=.method~new(" ","Say arg(1)") /* Create a method object */
myclass~define("ECHO",mymethod)  /* Define it in the class */
method_source = myclass~method("ECHO")~source /* Extract it */
say method_source /* Says "an Array" */
say method_source[1] /* Shows the method source code */
```

**5.1.1.18. methods**

Returns a Supplier object for all the instance methods of the receiving class and its superclasses, if no argument is specified. In this case, the supplier's indexes may contain duplicate entries, if classes override methods in superclasses.

If `class_object` is `.nil`, methods returns a Supplier object for only the instance methods of the receiving class. If a `class_object` is specified, this method returns a Supplier object containing only the instance methods that `class_object` defines.

The returned supplier's indexes are the method names and the supplier's items are their associated Method objects. The Supplier enumerates all the names and methods existing at the time of the supplier's creation.

**Note**

Methods that have been hidden with a `setMethod` or `define method` are included with the other methods that methods returns. The hidden methods have `.nil` for their associated method.
5.1.19. mixinClass

Returns a new mixin subclass of the receiver class. You can use this method to create a new mixin class that is a subclass of the superclass to which you send the message. The classid is a string that identifies the new mixin subclass. You can use the id method to retrieve this string.

The metaclass is a class object. If you specify metaclass, the new subclass is an instance of metaclass. (A metaclass is a class that you can use to create a class, that is, a class whose instances are classes. The Class class and its subclasses are metaclasses.)

If you do not specify a metaclass, the new mixin subclass is an instance of the default metaclass of the receiver class. For subclasses of the Object class, the default metaclass is the Class class.

The methods is a collection whose indexes are the names of methods and whose items are method objects (or strings or arrays of strings containing method code). If you specify methods, the new class is enhanced with class methods from this collection. (The metaclass of the new class is not affected.)

The metaclass method returns the metaclass of a class.

The method search order of the new subclass is the same as that of the receiver class, with the addition of the new subclass at the start of the order.

5.1.20. new

Returns a new instance of the receiver class, whose object methods are the instance methods of the class. This method initializes a new instance by running its init methods (see Section 4.2.9, “Initialization”). new also sends an init message. If you specify args, new passes these arguments on the init message.
Example 5.14. Class class — new method

```verbatim
/* new method example */
a = .account~new /* -> Object variable balance=0 */
y = .account~new(340.78) /* -> Object variable balance=340.78 */
/* plus free toaster oven */
::class account subclass object
::method init /* Report time each account created */
/* plus free toaster when more than $100 */
Expose balance
Arg opening_balance
Say "Creating" self~ObjectName "at time" time()
If datatype(opening_balance, "N") then balance = opening_balance
else balance = 0
If balance > 100 then Say " You win a free toaster oven"
```

5.1.1.21. package

Returns the Package class instance that defined the receiving class. The package instance controls and defines the search order for classes and routines referenced by the receiving class.

See also Package class.

Example 5.15. Class class — package method

```verbatim
say .Class~package~name                -- REXX
say .other~package~name                -- C:\ExampleClassPackage.rex
::class other
```

5.1.1.22. queryMixinClass

Returns .true if the class is a mixin class, or .false.

5.1.1.23. subclass

Returns a new subclass of the receiver class. You can use this method to create a new class that is a subclass of the superclass to which you send the message. The classid is a string that identifies the subclass. (You can use the id method to retrieve this string.)

The metaclass is a class object. If you specify metaclass, the new subclass is an instance of metaclass. (A metaclass is a class that you can use to create a class, that is, a class whose instances are classes. The Class class and its subclasses are metaclasses.)
If you do not specify a metaclass, the new subclass is an instance of the default metaclass of the receiver class. For subclasses of the Object class, the default metaclass is the Class class.

The methods is a collection whose indexes are the names of methods and whose items are method objects (or strings or arrays of strings containing method code). If you specify methods, the new class is enhanced with class methods from this collection. (The metaclass of the new class is not affected.)

The metaclass method returns the metaclass of a class.

The method search order of the new subclass is the same as that of the receiver class, with the addition of the new subclass at the start of the order.

Example 5.16. Class class — subclass method

```
room=.object~subclass("Room") /* Superclass is .object */
/* Subclass is room */
/* Subclass identity is Room */
```

5.1.1.24. subclasses

Returns the immediate subclasses of the receiver class in the form of a single-dimensional Array of the required size, in an unspecified order. (The program should not rely on any order.)

5.1.1.25. superClass

Returns the immediate superclass of the receiver class. The immediate superclass is the original class used on a subclass or a mixinClass method. For the Object Class, superClass returns .nil.

Example 5.17. Class class — superClass method

```
say .object~superclass -- displays "The NIL object"
say .class~superclass  -- displays "The Object class"
say .set~superclass    -- displays "The Table class"
```

5.1.1.26. superClasses

Returns the immediate superclasses of the receiver class in the form of a single-dimensional Array of the required size. The immediate superclasses are the original class used on a subclass or a mixinClass method, plus any additional superclasses defined with the inherit method. The array is in the order in which the class has inherited the classes. The original class used on a subclass or mixinClass method is the first item of the array.
### Example 5.18. Class class — superClasses method

```rexx
z=class~superClasses
/* To obtain the information this returns, you could use: */
do i over z
   say i
end
```

### 5.1.1.27. uninherit

![uninherit](image)

Nullifies the effect of any previous `inherit` message sent to the receiver for the class `classobj`.

**Note**

You cannot change the classes that Rexx provides by sending `uninherit` messages.

### Example 5.19. Class class — uninherit method

```rexx
location=.object~mixinClass("Location")
room=.object~subclass("Room")~inherit(location) /* Creates subclass */
/* and specifies inheritance */
room~uninherit(location)
```

### 5.1.2. Message Class

A message object provides for the deferred or asynchronous sending of a message. You can create a message object by using the `new` (Class Method) method of the Message class or the `start` and `startWith` methods of the Object class.

#### Table 5.2. Message Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Object class</strong></td>
</tr>
<tr>
<td><strong>Class (Metaclass)</strong></td>
</tr>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
<tr>
<td><strong>+ AlarmNotification</strong> (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>AlarmNotification class</strong></td>
</tr>
<tr>
<td>cancel</td>
</tr>
<tr>
<td>triggered (Abstract method)</td>
</tr>
<tr>
<td><strong>+ MessageNotification</strong> (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>MessageNotification class</strong></td>
</tr>
</tbody>
</table>
**messageComplete (Abstract Method)**

<table>
<thead>
<tr>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>new (Class Method)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>arguments</th>
<th>messageComplete</th>
<th>start</th>
</tr>
</thead>
<tbody>
<tr>
<td>completed</td>
<td>notify</td>
<td>startWith</td>
</tr>
<tr>
<td>errorCondition</td>
<td>reply</td>
<td>target</td>
</tr>
<tr>
<td>halt</td>
<td>replyWith</td>
<td>triggered</td>
</tr>
<tr>
<td>hasError</td>
<td>result</td>
<td>wait</td>
</tr>
<tr>
<td>hasResult</td>
<td>send</td>
<td></td>
</tr>
<tr>
<td>messageComplete</td>
<td>sendWith</td>
<td></td>
</tr>
</tbody>
</table>

### 5.1.2.1. new (Class Method)

Initializes the message object for sending the message name `messagename` to object `target`.

The `messagename` can be a string or an array. If `messagename` is an array object, its first item is the name of the message and its second item is a class object to use as the starting point for the method search.

If you specify neither **Individual** nor **Array**, the message sent has no arguments.

If you specify the **Individual** or **Array** option, any remaining arguments are arguments for the message. (You need to specify only the first letter; all characters following the first are ignored.)

**Individual**

If you specify this option, specifying `argument` is optional. Any arguments are passed as message arguments to `target` in the order you specify them.

**Array**

If you specify this option, you must specify `arguments`, which is an Array object. The member items of the array are passed to `target` as arguments. The first argument is at index 1, the second argument at index 2, and so on. If you omitted any indexes when creating the array, the corresponding message arguments are also omitted.

**Note**

This method does not send the message `messagename` to object `target`. See methods `start/startWith`, `send/sendWith`, and `reply/replyWith`.
5.1.2.2. arguments

Arguments

Returns an array of argument objects used to invoke the message.

5.1.2.3. completed

Completed

Returns 1 if the message object has completed executing its message, or 0. You can use this method to test for completion as an alternative to calling result and waiting for the message to complete.

5.1.2.4. errorCondition

Error Condition

Returns an error condition object from any execution error with the message object's message invocation. If the message completed normally, or is still executing, errorCondition returns .nil.

5.1.2.5. halt

Halt

Returns true if it could raise the halt condition for the message the receiving message object is currently executing. Returns false if there is no message executing.

An optional string description can be supplied, which the halted message can retrieve by requesting the "DESCRIPTION" item of the CONDITION built-in function or the Condition Object.

Example 5.20. Message class — halt method

```lisp
::class Task
  -- a long-running task that we may want to terminate early
  ::method runsLong
    use strict arg seconds
    signal on halt
    do s = 0 to seconds by 0.1 -- split SysSleep to enable halting
      call SysSleep 0.1 -- do "hard work"
    end
    return "task finished"
  halt:
    return condition("DESCRIPTION") -- return description from halt()

::class Watchdog inherit AlarmNotification

::class Task
```

134
-- sets a time-out, after which a running task will be halted
::method init
  expose timeOut
  use strict arg timeOut

-- watches over a task, halting it if it runs too long
::method watchTask
  expose timeOut
  use strict arg message
  -- we set an Alarm for 'timeOut' seconds, which, upon triggering
  -- will call method triggered(), passing this Alarm object as an argument
  -- (this is why we inherit from AlarmNotification)
  -- we also attach 'message' to enable triggered() to halt the task
  alarm = .Alarm~new(timeOut, self, message)

  -- now we just wait for 'message' to finish; either normally, or halted
  msgResult = message~result
  alarm~cancel                        -- cancel alarm; may still be active
  return msgResult

::method triggered unguarded
  expose timeOut
  -- our watchTask Alarm has triggered
  -- this means that the task has run too long
  use arg alarm
  message = alarm~attachment         -- message is our attachment
  message~halt("task took longer than" timeOut "sec")

will output

  task finished
  task took longer than 1 sec

### 5.1.2.6. hasError

Returns 1 if the message object’s message was terminated with an error condition. Returns 0 if the message has not completed or completed without error.

### 5.1.2.7. hasResult

Returns true if the message object has completed executing its message, and the message has returned a result. Returns false if the message object has not yet completed executing its message, or the message hasn’t returned a result.

See also method result.

### 5.1.2.8. messageComplete
This method implements the *MessageNotification interface*. It will be called whenever a message completes processing, for which notification was requested by using the `notify` method with a message object as the notification target.

As a result of receiving such a notification, `messageComplete` will send the receiving message to start processing. Any `source` argument will be ignored.

See also class *MessageNotification*.

**Example 5.21. Message class — messageComplete method**

```plaintext
msg = .Message~new(.Array~new(1000)~fill(0)~allIndexes, "sort")
msg~notify(.Message~new(.stdout, "say", "I", "sorting complete"))
msg~notify(.Message~new(.r~new, "items", "I", msg))
msg~start
say "processing continues"
```

will output

```
processing continues
sorting complete
1000 items sorted
```

**5.1.2.9. messageName**

`messageName`

Returns the message name of the receiving message object.

**5.1.2.10. notify**

`notify(target)`

Requests notification about the completion of processing of the message.

The notification `target` must be an object that implements the *MessageNotification interface*. Upon completion of message processing, `target` will be sent a `messageComplete` message, with the completed Message object as argument `source`.

Any number of notifications can be requested for a message.

To retrieve the result of the completed message, use method `result`.

See also class *MessageNotification*.

**Example 5.22. Message class — notify method**

```plaintext
msg = .Message~new(.Array~new(1000)~fill(0)~allIndexes, "sort")
```
msg-notify(.Sorter-new)
msg-start
say "processing continues"

::class Sorter inherit MessageNotification
::method messageComplete
use strict arg message
say message-target-items "items sorted"

will output

<table>
<thead>
<tr>
<th>processing continues</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 items sorted</td>
</tr>
</tbody>
</table>

5.1.2.11. reply

Returns and sends a copy of the message to start processing, while the sender also continues processing.

As this method, other than the similar method start, starts a copy of the message, it can be called multiple times with the same receiving message.

If target is specified, the message is sent to target and both the receiving message and the message copy are changed to use the new target. Otherwise the message is sent to the target the message object provides.

If any arguments are specified, the message is sent with these arguments and both the receiving message and the message copy are changed to use the new arguments. Otherwise the message is sent with any arguments the message object provides.

This method returns as soon as possible and does not wait until message processing is complete.

The notify method can be used to request notification that message processing is complete. When message processing is complete, the message object retains any result and holds it until requested via the result method.

See also
• method replyWith,
• methods start and startWith, and
• methods send and sendWith.

5.1.2.12. replyWith

Returns and sends a copy of the message with the specified arguments to start processing, while the sender also continues processing.
As this method, other than the similar method `startWith`, starts a copy of the message, it can be called multiple times with the same receiving message.

If `target` is specified, the message is sent to `target` and both the receiving message and the message copy are changed to use the new `target`. Otherwise the message is sent to the target the message object provides.

The `arguments` array items are used as message arguments and both the receiving message and the message copy are changed to use these new `arguments`.

This method returns as soon as possible and does not wait until message processing is complete.

The `notify` method can be used to request notification that message processing is complete. When message processing is complete, the message object retains any result and holds it until requested via the `result` method.

See also
- method `reply`,
- methods `start` and `startWith`, and
- methods `send` and `sendWith`.

### 5.1.2.13. result

Returns the result of the message `send` or `start`. If message processing is not yet complete, this method waits until it completes. If the message `send` or `start` raises an error condition, this method also raises an error condition.

For an example see `halt method example`.

### 5.1.2.14. send

Returns the result (if any) of sending the message.

If `target` is specified, the message is sent to `target` and the receiving message object is changed to use the new `target`. Otherwise the message is sent to the target the message object provides.

If any `arguments` are specified, the message is sent with these `arguments` and the receiving message object is changed to use the new `arguments`. Otherwise the message is sent with any arguments the message object provides.

This method does not return until message processing is complete.

The `notify` method can be used to request notification that message processing is complete. When message processing is complete, the message object retains any result and holds it until requested via the `result` method.

See also
- method `sendWith`,
- methods `start` and `startWith`, and
• methods \texttt{reply} and \texttt{replyWith}.

5.1.2.15. \texttt{sendWith}

\begin{center}
\texttt{sendWith(target, arguments)}
\end{center}

Returns the result (if any) of sending the message with the specified \texttt{arguments}.

If \texttt{target} is specified, the message is sent to \texttt{target} and the receiving message object is changed to use the new \texttt{target}. Otherwise the message is sent to the target the message object provides.

The \texttt{arguments} array items are used as message arguments and the receiving message object is changed to use these new \texttt{arguments}.

This method does not return until message processing is complete.

The \texttt{notify} method can be used to request notification that message processing is complete. When message processing is complete, the message object retains any result and holds it until requested via the \texttt{result} method.

See also
• method \texttt{send},
• methods \texttt{start} and \texttt{startWith}, and
• methods \texttt{reply} and \texttt{replyWith}.

5.1.2.16. \texttt{start}

\begin{center}
\texttt{start(target, argument)}
\end{center}

Sends the message to start processing, while the sender also continues processing.

If \texttt{target} is specified, the message is sent to \texttt{target} and the receiving message object is changed to use the new \texttt{target}. Otherwise the message is sent to the target the message object provides.

If any \texttt{arguments} are specified, the message is sent with these \texttt{arguments} and the receiving message object is changed to use the new \texttt{arguments}. Otherwise the message is sent with any arguments the message object provides.

This method returns as soon as possible and does not wait until message processing is complete.

Note that once a message object has been started with either the \texttt{start} or the \texttt{startWith} method, it cannot be run with any of the \texttt{send/sendWith}, \texttt{start/startWith}, \texttt{reply/replyWith} methods again. A message object can be run multiple times with methods \texttt{send/sendWith} and \texttt{reply/replyWith}.

The \texttt{notify} method can be used to request notification that message processing is complete. When message processing is complete, the message object retains any result and holds it until requested via the \texttt{result} method.

See also
• method \texttt{startWith},
• methods \texttt{send} and \texttt{sendWith}, and
• methods \textit{reply} and \textit{replyWith}.

\begin{example}
\textbf{Example 5.23. Message class - start method}

```plaintext
ez=.testclass~new                  /* Creates a new instance of Testclass */

/* Creates and starts message mymsg to send SHOWMSG to ez */
mymsg=ez~start("SHOWMSG","Hello, Ollie!",5)

/* Continue with main processing while SHOWMSG runs concurrently */
do 5
  say "Hello, Stan!"
end

/* Get final result of the SHOWMSG method from the mymsg message object */
say mymsg~result
say "Goodbye, Stan..."
exit
```

::class testclass public             /* Directive defines Testclass */
::method showmsg                     /* Directive creates new method SHOWMSG */
use arg text,reps                    /* class Testclass */
do reps
  say text
end
reply "Bye Bye, Ollie..."
return
```

The following output is possible:

```
Hello, Ollie!
Hello, Stan!
Hello, Ollie!
Hello, Stan!
Hello, Ollie!
Hello, Stan!
Hello, Ollie!
Hello, Stan!
Hello, Ollie!
Hello, Stan!
Bye Bye, Ollie...
Goodbye, Stan...
```

\subsection{5.1.2.17. startWith}

Sends the message with the specified \textit{arguments} to start processing, while the sender also continues processing.

If \textit{target} is specified, the message is sent to \textit{target} and the receiving message object is changed to use the new \textit{target}. Otherwise the message is sent to the target the message object provides.

The \textit{arguments} array items are used as message arguments and the receiving message object is changed to use these new \textit{arguments}.

This method returns as soon as possible and does not wait until message processing is complete.
Note that once a message object has been started with either the `start` or the `startWith` method, it cannot be run with any of the `send/sendWith, start/startWith, reply/replyWith` methods again. A message object can be run multiple times with methods `send/sendWith` and `reply/replyWith`.

The `notify` method can be used to request notification that message processing is complete. When message processing is complete, the message object retains any result and holds it until requested via the `result` method.

See also
• method `start`,
• methods `send` and `sendWith`, and
• methods `reply` and `replyWith`.

### 5.1.2.18. target

Returns the object that is the target of the invoked message.

### 5.1.2.19. triggered

This method implements the `AlarmNotification interface`. It will be called whenever an `Alarm` or a `Ticker` triggers, for which a Message object was set as notification target.

As a result of receiving such a notification, `triggered` will send the receiving message to start processing. Any `source` argument will be ignored.

See also class `AlarmNotification`.

#### Example 5.24. Message class — triggered method

```lua
arg = .Array~new(1)
msg = .Message~new(.a~new, "ring", "A", arg)
oneSecond = .Alarm~new(1, msg, arg)
arg~append(oneSecond)
-- oneSecond = .Alarm~new(1, .Message~new(.a~new, "ring", "I", alarm))
say "processing continues"
::class a
::method ring
use strict arg alarm
say "alarm went off at" .DateTime~new
say "was scheduled for" alarm~scheduledTime
```

may output

```
processing continues
alarm went off at 2015-12-18T16:54:15.550000
was scheduled for 2015-12-18T16:54:15.545000
```
5.1.2.20. wait

Waits until the message object has completed executing its message.

See also method completed.

5.1.3. Method Class

The Method class creates method objects from Rexx source code. It is a subclass of the Object class.

Table 5.3. Method Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the</td>
<td></td>
</tr>
<tr>
<td>Object class</td>
<td></td>
</tr>
<tr>
<td>Class (MetaClass)</td>
<td></td>
</tr>
<tr>
<td>Methods inherited from the</td>
<td></td>
</tr>
<tr>
<td>Class class</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td></td>
</tr>
<tr>
<td>new (Class Method)</td>
<td>annotation</td>
</tr>
<tr>
<td>loadExternalMethod (Class</td>
<td>isPackage</td>
</tr>
<tr>
<td>Method)</td>
<td>isPrivate</td>
</tr>
<tr>
<td>newFile (Class Method)</td>
<td>isAbstract</td>
</tr>
<tr>
<td></td>
<td>isProtected</td>
</tr>
<tr>
<td></td>
<td>isAttribute</td>
</tr>
<tr>
<td></td>
<td>isConstant</td>
</tr>
<tr>
<td></td>
<td>isGuarded</td>
</tr>
<tr>
<td></td>
<td>package</td>
</tr>
<tr>
<td></td>
<td>scope</td>
</tr>
<tr>
<td></td>
<td>setGuarded</td>
</tr>
<tr>
<td></td>
<td>setPrivate</td>
</tr>
<tr>
<td></td>
<td>setProtected</td>
</tr>
<tr>
<td></td>
<td>setSecurityManager</td>
</tr>
<tr>
<td></td>
<td>setUnguarded</td>
</tr>
</tbody>
</table>

5.1.3.1. new (Class Method)

Returns a new instance of the Method class, which is an executable representation of the code contained in the source. The name is a string. The source can be a single string or an array of strings containing individual method lines.

The context allows the created method to inherit class and routine lookup scope from another source. If specified, context can be a Method object, a Routine object, a Package object, or the string "PROGRAMSCOPE". PROGRAMSCOPE is the default, and specifies that the newly created method will inherit the class and routine search scope from the caller of the new method.

5.1.3.2. newFile (Class Method)
Returns a new instance of the Method class, which is an executable representation of the code contained in the file filename. Raises an error if the file filename cannot be read. The filename is a string.

The context allows the created method to inherit class and routine lookup scope from another source. If specified, context can be a Method object, a Routine object, a Package object, or the string "PROGRAMSCOPE". PROGRAMSCOPE is the default, and specifies that the newly created method will inherit the class and routine search scope from the caller of the newFile method.

5.1.3.3. loadExternalMethod (Class Method)

Resolves a native method in an external library package and returns a Method object instance that can be used to call the external method. The descriptor is a string containing whitespace-delimited tokens that identify the location of the native method. The first token identifies the type of native function and must be LIBRARY. The second token must identify the name of the external library. The external library is located using platform-specific mechanisms for loading libraries. For Unix-based systems, the library name is case-sensitive. The third token is optional and specifies the name of the method within the library package. If not specified, name is used. The method name is not case sensitive. If the target method cannot be resolved, .nil is returned.

Example 5.25. Method class — loadExternalMethod method

```ruby
method = .Method~loadExternalMethod("homeAddress=", "LIBRARY mylib setHomeAddress")
```

5.1.3.4. annotation

Returns the value of the annotation named name for this method. If no such annotation exists, .nil is returned.

See also
• method annotations and
• ::=ANNOTATE directive.

5.1.3.5. annotations

Returns a StringTable of all annotation name/value pairs for this method.

See also
• method annotation and
• ::=ANNOTATE directive.

5.1.3.6. isAbstract
Returns `.true.` if the method is an Abstract method, otherwise returns `.false.`.

See also
- ABSTRACT option of the `::METHOD` directive and
- ABSTRACT option of the `::ATTRIBUTE` directive.

Example 5.26. Method class — isAbstract method

<table>
<thead>
<tr>
<th>Code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>say .InputStream-method(&quot;lines&quot;)-isAbstract</td>
<td>-- 1</td>
</tr>
<tr>
<td>say .Stream-method(&quot;lines&quot;)-isAbstract</td>
<td>-- 0</td>
</tr>
</tbody>
</table>

### 5.1.3.7. isAttribute

Returns `.true.` if the method is an Attribute method, otherwise returns `.false.`.

See also `::ATTRIBUTE` directive.

Example 5.27. Method class — isAttribute method

<table>
<thead>
<tr>
<th>Code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>say .File-method(&quot;lastModified&quot;)-isAttribute</td>
<td>-- 1</td>
</tr>
<tr>
<td>say .File-method(&quot;lastModified=&quot;)-isAttribute</td>
<td>-- 1</td>
</tr>
</tbody>
</table>

### 5.1.3.8. isConstant

Returns `.true.` if the method is a Constant method, otherwise returns `.false.`.

See also `::CONSTANT` directive.

Example 5.28. Method class — isConstant method

<table>
<thead>
<tr>
<th>Code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>say .physics-method(&quot;c&quot;)-isConstant</td>
<td>-- 1</td>
</tr>
<tr>
<td>::class physics</td>
<td></td>
</tr>
<tr>
<td>::constant c 299792458</td>
<td></td>
</tr>
</tbody>
</table>

### 5.1.3.9. isGuarded

Returns `.true.` if the method is a Guarded method. Returns `.false.` for Unguarded methods.

### 5.1.3.10. isPackage
Returns `.true` if the method is a Package-scope method. Returns `.false` for Public methods or Private methods. See Section 4.2.8, “Public, Package-Scope, and Private Methods” for details on package-scope method restrictions.

See also method `isPrivate`.

### 5.1.3.11. isPrivate

Returns `.true` if the method is a Private method. Returns `.false` for Public methods or Package-scope methods. See Section 4.2.8, “Public, Package-Scope, and Private Methods” for details on private method restrictions.

See also method `isPackage`.

### 5.1.3.12. isProtected

Returns `.true` if the method is a Protected method. Returns `.false` for unprotected methods.

### 5.1.3.13. package

Returns the Package class instance that defined the method instance. The package instance controls and defines the search order for classes and routines referenced by the method code.

### 5.1.3.14. scope

Returns the defining class scope for a method. Returns `.nil` for any method not defined by a class scope.

---

Example 5.29. Method class — scope method

```
scopes = .Stem~new
scopes[] = 0
loop with item method over .List~methods
  scopes[method~scope] += 1
  -- possible output:
end
-- 27 methods from The List class
loop scope over scopes
  -- 19 methods from The Collection class
  say scopes[scope] "methods from" scope
  -- possible output:
end
-- 20 methods from The OrderedCollection class
-- 31 methods from The Object class
.Method~new("any", "nop")~scope        -- .nil
.methods["FLOAT"]~scope              -- .nil
::method float                        -- .nil
```
5.1.3.15. setGuarded

Specifies that the method is a guarded method that requires exclusive access to its scope variable pool to run. If the receiver is already guarded, a setGuarded message has no effect. Guarded is the default state for method objects.

5.1.3.16. setPrivate

Specifies that a method is a private method. By default, method objects are created as public methods. See Section 4.2.8, “Public, Package-Scope, and Private Methods” for details on private method restrictions.

5.1.3.17. setProtected

Specifies that a method is a protected method. Method objects are not protected by default. (See Chapter 13, The Security Manager for details.)

5.1.3.18. setSecurityManager

Replaces the existing security manager with the specified security_manager_object. If security_manager_object is omitted, any existing security manager is removed.

5.1.3.19. setUnguarded

Turns off the guard attribute of the method, allowing this method to run on an object even if another method has acquired exclusive access to the scope variable pool. Methods are guarded by default.

A guarded method can be active for an object only when no other method requiring exclusive access to the object's variable pool is active in the same object. This restriction does not apply if an object sends itself a message to run a method and it already has exclusive use of the same object variable pool. In this case, the method runs immediately regardless of its guarded state.

5.1.3.20. source

Returns the method source code as a single-dimensional Array of source lines. If the source code is not available, source returns an array of zero items.
5.1.4. Object Class

The Object class is the root of the class hierarchy. The instance methods of the Object class are, therefore, available on all objects.

Table 5.4. Object Class

<table>
<thead>
<tr>
<th>Class (Metaclass)</th>
<th>Methods inherited from the Class class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>new (Class Method)</td>
</tr>
</tbody>
</table>

Comparison Methods: 

- =
- ==
- <>
- ><
- \=<
- \==

Concatenation Methods: 

- (abuttal)
- ||
- (blank)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>isA</td>
</tr>
<tr>
<td>copy</td>
<td>isnInstanceOf</td>
</tr>
<tr>
<td>defaultName</td>
<td>isNil</td>
</tr>
<tr>
<td>hashCode</td>
<td>objectName</td>
</tr>
<tr>
<td>hasMethod</td>
<td>objectName=</td>
</tr>
<tr>
<td>identityHash</td>
<td>request</td>
</tr>
<tr>
<td>instanceMethod</td>
<td>run (Private Method)</td>
</tr>
<tr>
<td>instanceMethods</td>
<td>send</td>
</tr>
<tr>
<td>sendWith</td>
<td>setMethod (Private Method)</td>
</tr>
<tr>
<td>start</td>
<td></td>
</tr>
<tr>
<td>startWith</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td></td>
</tr>
<tr>
<td>unsetMethod (Private Method)</td>
<td></td>
</tr>
</tbody>
</table>

5.1.4.1. new (Class Method)

Returns a new instance of the receiver class.

5.1.4.2. Comparison Methods

Returns .true or .false, the result of performing a specified comparison operation.

For the Object class, if argument is the same object as the receiver object, the result is .true, otherwise .false is returned. Subclasses may override this method to define equality using different criteria. For example, the String class determines equality based on the value of the string data.

Note

The MapCollection classes such as Table and Relation use the == operator combined with the hashCode method to determine index and item equivalence. It is generally necessary for a class to override both the hashCode method and the == operator method to maintain the contract specified for the hashCode method.

The comparison operators you can use in a message are:
\[=, ==\]

- **true** if the terms are the same object.

\[\neq, ><, <>, \neq\]

- **true** if the terms are not the same object (inverse of =).

### 5.1.4.3. Concatenation Methods

```
concatenation_operator( argument )
```

Returns a new string that is the concatenation of receiver object's string value with `argument`. (See Section 1.11.2.1, "String Concatenation"). The `concatenation_operator` can be:

```
==
```

concatenates without an intervening blank. The abuttal operator `""` is the null string. The language processor uses the abuttal operator to concatenate two terms that another operator does not separate.

```
||
```

concatenates without an intervening blank.

```
" "
```

concatenates with one blank between the receiver object and the `argument`. (The operator `" "` is a blank.)

### 5.1.4.4. class

```
class
```

Returns the class object that created the object instance.

### 5.1.4.5. copy

```
copy
```

Returns a copy of the receiver object. The copied object has the same methods as the receiver object and an equivalent set of object variables, with the same values.

---

**Example 5.30. Object class — copy method**

```javascript
myarray=.array-<>of("N","S","E","W")
/* Copies array myarray to array directions */
directions=myarray-copy
```
The copy method is a “shallow copy”. Only the target object is copied. Additional objects referenced by the target object are not copied. For example, copying an Array object instance only copies the Array, it does not copy any of the objects stored in the Array.

5.1.4.6. defaultName

defaultname

Returns a short human-readable string representation of the object. The exact form of this representation depends on the object and might not alone be sufficient to reconstruct the object. All objects must be able to produce a short string representation of themselves in this way, even if the object does not have a string value. See Section 4.2.11, “Required String Values” for more information. defaultName returns a string that identifies the class of the object, for example, an Array or a Directory.

See also methods objectName and string.

See objectName= for an example using defaultName.

5.1.4.7. hashCode

hashCode

Returns a string value that is used as a hash value for MapCollections such as Table, Relation, Set, Bag, and Directory. MapCollections use this string value to hash an object for hash table-based searches.

Object implementations are expected to abide by a general contract for hash code usage:

- Whenever hashCode is invoked on the same object more than once, hashCode must return the same hashcode value, provided than none of the internal information the object uses for an "==" comparison has changed.

- If two object instances compare equal using the "==" operator, the hashCode methods for both object instances must return the same value.

- It is not required that two object instances that compare unequal using "==" return different hash code values.

- Returning a wide range of hash values will produce better performance when an object is used as an index for a MapCollection. A return value of 4 string characters is recommended. The characters in the hash value may be any characters from '00'x to 'ff'x, inclusive.

5.1.4.8. hasMethod

hasMethod( methodName )


Returns \texttt{true} if the receiver object has a method named \textit{methodname} (translated to uppercase). Otherwise, it returns \texttt{false}.

\textbf{Note}

The \texttt{hasMethod} method will return \texttt{true} even if the target method is defined as private. A private method has restricted access rules, so it's possible to receive an unknown method error (error 97) when invoking \textit{methodname} even if \texttt{hasMethod} indicates the method exists. See Section 4.2.8, “Public, Package-Scope, and Private Methods” for private method restrictions.

\subsection*{5.1.4.9. identityHash}

\texttt{identityHash}

Returns a unique identity number for the object. This number is guaranteed to be unique for the receiver object until the object is garbage collected.

\subsection*{5.1.4.10. init}

\texttt{init}

Performs any required object initialization. Subclasses of the Object class can override this method to provide more specific initialization.

\subsection*{5.1.4.11. instanceMethod}

\texttt{instanceMethod(\texttt{methodname})}

Returns the corresponding Method class instance if the \textit{methodname} is a valid method of the class. Otherwise it returns \texttt{nil}.

\subsection*{5.1.4.12. instanceMethods}

\texttt{instanceMethods(\texttt{class_object})}

Returns a \texttt{Supplier} object for all the object methods of the receiving object and its superclasses, if no argument is specified. In this case, the supplier's indexes may contain duplicate entries, if classes override methods in superclasses.

If a \texttt{class_object} is specified, \texttt{instanceMethods} returns a Supplier object for only the object methods of the receiving object. If the receiving object object is not an instance of \texttt{class_object}, an empty Supplier is returned.

The returned supplier's indexes are the method names and the supplier's items are their associated \texttt{Method} objects. The Supplier enumerates all the names and methods existing at the time of the supplier's creation.
Note

Methods that have been hidden with a setMethod or define method are included with the other methods that instanceMethods returns. The hidden methods have `.nil` for their associated method.

Example 5.31. Object class — instanceMethods method

```ruby
-- list all class methods of .String only
say .String-instanceMethods(.String)-allIndexes --> CR,NEW,NL,NULL,TAB

-- count all class methods of .String and its superclasses
-- 4 .String class methods, 32 .Object class methods, 34 .Class class methods
say .String-instanceMethods-allIndexes-items --> 70

-- count all instance methods of .String only
say ''-instanceMethods(.String)-allIndexes-items --> 116

-- count all instance methods of .String and its superclasses
say ''-instanceMethods-allIndexes-items --> 148
```

5.1.4.13. isA

```ruby
isA(class)
```

Note

This method is an alias of the `isInstanceOf` method.

5.1.4.14. isInstanceOf

```ruby
isInstanceOf(class)
```

Returns `.true` if the object is an instance of the specified `class`, otherwise it returns `.false`. An object is an instance of a class if the object is directly an instance of the specified `class` or if `class` is in the object's direct or mixin class inheritance chain. For example:

Example 5.32. Object class — isInstanceOf method

```ruby
"abc"-isInstanceOf(.string)        --> 1
"abc"-isInstanceOf(.object)        --> 1
"abc"-isInstanceOf(.mutablebuffer) --> 0
```
5.1.4.15. isNil

Returns .true if the receiving object is the .nil object. Returns .false otherwise.

Example 5.33. Object class — isNil method

```plaintext
say .Object~isNil    -- 0
say .nil~isNil       -- 1
```

5.1.4.16. objectName

Returns any name set on the receiver object using the objectName= method. If the receiver object does not have a name, this method returns the result of the defaultName method. See Section 4.2.11, “Required String Values” for more information. See the objectName= method for an example using objectName.

5.1.4.17. objectName=

Sets the receiver object's name to the string newName.

Example 5.34. Object class — objectName= method

```plaintext
points=.array~of("N","S","E","W")
say points~objectName         /* (no change yet) Says: "an Array"    */
points~objectName=("compass") /* Changes obj name POINTS to "compass"*/
say points~objectName         /* Shows new obj name. Says: "compass" */
say points~defaultName        /* Default is still available.         */
                          /* Says "an Array" */
say points                   /* Says string representation of */
                          /* points "compass" */
say points[3]                 /* Says: "E"  Points is still an array */
                          /* of 4 items */
```

5.1.4.18. request

Returns an object of the classid class, or .nil if the request cannot be satisfied.

This method first compares the identity of the object's class (see the id method of the Class class) to classid. If they are the same, the receiver object is returned as the result. Otherwise, request tries to obtain and return an object satisfying classid by sending the receiver object the conversion message make with the string classid appended (converted to uppercase). For example,
a request("string") message causes a makeString message to be sent. If the object does not have the required conversion method, request returns .nil.

The conversion methods cause objects to produce different representations of themselves. The presence or absence of a conversion method defines an object’s capability to produce the corresponding representations. For example, lists can represent themselves as arrays, because they have a makeArray method, but they cannot represent themselves as directories, because they do not have a makeDirectory method. Any conversion method must return an object of the requested class. For example, makeArray must return an array. The language processor uses the makeString method to obtain string values in certain contexts; see Section 4.2.11, “Required String Values”.

5.1.4.19. run (Private Method)

Runs method, which can be either
• a method object, or
• a string containing a method source line, or an Array of strings containing individual method source lines (for these cases an equivalent method object is created).

The method has access to the object variables of the receiver object, as if the receiver object had defined the method by using setMethod.

If you specify neither Individual nor Array, the method runs without arguments.

If you specify the Individual or Array option, any remaining arguments are arguments for the method. (You need to specify only the first letter; all characters following the first character are ignored.)

Individual
Passes any remaining arguments to the method as arguments in the order you specify them.

Array
Requires arguments, which is an Array object. The member items of the array are passed to the method as arguments. The first argument is at index 1, the second argument at index 2, and so on. If you omitted any indexes when creating the array, the corresponding arguments are omitted when passing the arguments.

Notes:
1. The run method is a private method (see Section 4.2.8, “Public, Package-Scope, and Private Methods”) with the additional restriction that it can only be called
   • from an instance method of the receiving object itself, or
   • from a class method in the receiving object’s inheritance chain.

2. The run method is a protected method.

5.1.4.20. send
Returns the result of invoking a method on the target object using the specified message name and arguments. The send method allows methods to be invoked using dynamically constructed method names.

The *messagename* can be a string or an array. If *messagename* is an array object, its first item is the name of the message and its second item is a class object to use as the starting point for the method search.

Any *arguments* are passed to the receiver as arguments for *messagename* in the order you specify them.

Example 5.35. Object class — send method

```ruby
world = .WorldObject~new
-- these calls are equivalent and produce "Hello World, I'm Fred!"
say world-hello("World", "Fred")
say world-send("HELLO", "World", "Fred")
::class WorldObject
::method hello
  use strict arg place, name
  return "Hello" place", I'm" name"!"
```

### 5.1.4.21. sendWith

Returns the result of invoking a method on the target object using the specified message name and arguments. The sendWith method allows methods to be invoked using dynamically constructed method names and arguments.

The *messagename* can be a string or an array. If *messagename* is an array object, its first item is the name of the message and its second item is a class object to use as the starting point for the method search.

The *arguments* argument must be a single-dimensional Array instance. The values contained in *arguments* are passed to the receiver as arguments for *messagename* in the order you specify them.

Example 5.36. Object class — sendWith method

```ruby
world = .WorldObject~new
-- these calls are equivalent and produce "Hello World, I'm Fred!"
say world-hello("World", "Fred")
say world-sendWith("HELLO", .Array~of("World", "Fred"))
::class WorldObject
::method hello
  use strict arg place, name
  return "Hello" place", I'm" name"!"
```
5.1.4.22. setMethod (Private Method)

setMethod(
  methodname,  
  method,      
  "FLOAT",     
  "OBJECT")

Adds method to the receiver object’s collection of object methods.

The methodname is the name of the new method. This name is translated to uppercase. If you previously defined a method with the same name using setMethod, the new method replaces the earlier one. If you omit method, setMethod makes the method name methodname unavailable for the receiver object. In this case, sending a message of that name to the receiver object runs the unknown method (if any).

The method can be either
• a method object, or
• a string containing a method source line, or an Array of strings containing individual method source lines (for these cases an equivalent method object is created).

The third parameter is optional, and describes if the method that is attached to an object should have OBJECT or FLOAT scope. FLOAT scope, which is the default, means that it shares the same scope with methods that were defined outside of a class. OBJECT scope means it shares the scope with other, potentially statically defined, methods of the object it is attached to.

Notes:

1. The setMethod method is a private method (see Section 4.2.8, “Public, Package-Scope, and Private Methods”) with the additional restriction that it can only be called
   • from an instance method of the receiving object itself, or
   • from a class method in the receiving object’s inheritance chain.

2. The setMethod method is a protected method.

5.1.4.23. start

start(
  messagename,  
  argument)

Returns a message object and sends it a start message to start concurrent processing. The object receiving the message messagename processes this message concurrently with the sender’s continued processing.

The messagename can be a string or an array. If messagename is an array object, its first item is the name of the message and its second item is a class object to use as the starting point for the method search.

Any arguments are passed to the receiver as arguments for messagename in the order you specify them.
When the receiver object has finished processing the message, the message object retains its result and holds it until the sender requests it by sending a result message. For further details, see Message class method start.

Example 5.37. Object class — start method

```
world = .WorldObject~new
-- these calls are equivalent and produce "Hello World, I'm Fred!"
msg1 = world~start("HELLO", "World", "Fred")
msg2 = .message~new(world, "HELLO", "i", "World", "Fred")~start
say msg1~result
say msg2~result
```

::class WorldObject
::method hello
use strict arg place, name
return "Hello" place", I'm" name"!"

5.1.4.24. startWith

```
startWith( messageName
, arguments
)
```

Returns a message object and sends it a start message to start concurrent processing. The object receiving the message `messageName` processes this message concurrently with the sender's continued processing.

The `messageName` can be a string or an array. If `messageName` is an array object, its first item is the name of the message and its second item is a class object to use as the starting point for the method search.

The `arguments` argument must be a single-dimensional Array instance. Any values contained in `arguments` are passed to the receiver as arguments for `messageName` in the order you specify them.

When the receiver object has finished processing the message, the message object retains its result and holds it until the sender requests it by sending a result message. For further details, see Message class method start.

Example 5.38. Object class — startWith method

```
world = .WorldObject~new
-- these calls are equivalent and produce "Hello World, I'm Fred!"
msg1 = world~startWith("HELLO", .Array~of("World", "Fred"))
msg2 = .message~new(world,"HELLO", "a", .Array~of("World", "Fred"))~start
say msg1~result
say msg2~result
```

::class WorldObject
::method hello
use strict arg place, name
return "Hello" place", I'm" name"!"

5.1.4.25. string
Returns a human-readable string representation of the object. The exact form of this representation depends on the object and might not alone be sufficient to reconstruct the object. All objects must be able to produce a string representation of themselves in this way.

The object's string representation is obtained from the objectName method (which can in turn use the defaultName method).

The distinction between this method, the makeString method (which obtains string values) and the request method is important. All objects have a string method, which returns a string representation (human-readable form) of the object. This form is useful in tracing and debugging. Only those objects that have information with a meaningful string form have a makeString method to return this value. For example, Directory objects have a readable string representation ("a Directory"), but no string value, and, therefore, no makeString method.

Of the classes that Rexx provides, the Array Class, the CircularQueue Class, the DateTime Class, the File Class, the MutableBuffer Class, the StackFrame Class, the String Class, and the TimeSpan Class have a makeString method. Any subclasses of these classes inherit this method by default, so these subclasses also have string values. Any other class can also provide a string value by defining a makeString method.

### 5.1.4.26. unsetMethod (Private Method)

```plaintext
unsetMethod(methodname)
```

Cancels the effect of all previous setMethod for method methodname. It also removes any method methodname introduced with enhanced when the object was created. If the object has received no setMethod method, no action is taken.

**Notes:**

1. The unsetMethod method is a private method (see Section 4.2.8, “Public, Package-Scope, and Private Methods”) with the additional restriction that it can only be called
   - from an instance method of the receiving object itself, or
   - from a class method in the receiving object's inheritance chain.

2. The unsetMethod method is a protected method.

### 5.1.5. Package Class

The Package class contains the source code for a package of Rexx code. A package instance holds all of the routines, classes, and methods created from a source code unit and also manages external dependencies referenced by ::REQUIRES directives. The files loaded by ::REQUIRES are also contained in Package class instances. It is a subclass of the Object class.

**Table 5.5. Package Class**

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>Package</td>
</tr>
<tr>
<td>new (Class Method)</td>
</tr>
</tbody>
</table>
### 5.1.5.1. new (Class Method)

Returns a new instance of the package class, which is a representation of the code contained in the `source`. The `name` is a string. The `source` can be a single string or an array of strings containing individual method lines. If `source` isn't specified, `name` identifies a file that will be used as the package source. The file is searched for using the *external program search order*.

The `context` allows the created package to inherit class and routine lookup scope from another source. If specified, `context` can be a Method object, a Routine object, or a Package object. If not specified, the newly created package will inherit the class and routine search scope from the caller of new method.

### 5.1.5.2. addClass

adds the class object `class` to the available classes under the name `name`. This is added to the package as a non-public class.

### 5.1.5.3. addPackage

Adds the package object `package` to the dependent packages. An optional string `namespace` may be specified, which will allow to qualify references to classes or routines within `package`.

The added package is processed as if it had been added using a `::REQUIRES` directive in the original package source.

#### Example 5.39. Package class — addPackage method

```
.context~package~addPackage(.Package~new('winsystm.cls'), 'windows')
say windows:virtualkeycodes~new~keyName(36) -- HOME
```
5.1.5.4. addPublicClass

```
addPublicClass(name, class)
```

Adds the class object `class` to the available public classes under the name `name`. This is added to the package as a public class.

5.1.5.5. addPublicRoutine

```
addPublicRoutine(name, routine)
```

Adds the routine object `routine` to the available routines under the name `name`. This is added to the package as a public routine.

5.1.5.6. addRoutine

```
addRoutine(name, routine)
```

Adds the routine object `routine` to the available routines under the name `name`. This is added to the package as a non-public routine.

5.1.5.7. annotation

```
annotation(name)
```

Returns the value of the annotation named `name` for this package. If no such annotation exists, `.nil` is returned.

See also
- method `annotations` and
- `::ANNOTATE` directive.

5.1.5.8. annotations

```
annotations
```

Returns a StringTable of all annotation name/value pairs for this package.

See also
- method `annotation` and
- `::ANNOTATE` directive.

5.1.5.9. classes

```
classes
```

Returns a StringTable containing all classes defined by this package.

See also methods `publicClasses` and `importedClasses`.
5.1.5.10. definedMethods

```
definedMethods
```

Returns a StringTable containing all unattached methods defined by this package. This is the same StringTable available to code within the package via the .METHODS environment symbol.

5.1.5.11. digits

```
digits
```

Returns the initial NUMERIC DIGITS setting used for all Rexx code contained within the package. The default value is 9. The ::OPTIONS directive can override the default value.

5.1.5.12. findClass

```
findClass(name)
```

Performs the standard environment symbol searches given name. The search is performed using the same search mechanism used for environment symbols or class names specified on ::CLASS directives. If the name is not found, .nil will be returned.

Note that the standard environment symbol search will return an object instance (and not a class) when searching e.g. for "nil", "true", "false", "endofline", "RexxInfo", or objects in the .LOCAL directory.

See also method findPublicClass.

Example 5.40. Package class — findClass method

```rexx
say .RexxInfo~package~findClass("Dummy")  -- The NIL object
say .context~package~findClass("Dummy")   -- The DUMMY class
::class Dummy
```

5.1.5.13. findNamespace

```
findNamespace(name)
```

Returns the Package object that has been tagged with namespace name. Returns .nil if namespace name does not exist.

See also method addPackage.

Example 5.41. Package class — findNamespace method

```rexx
say .context~package~findNamespace("rexx")      -- The REXX Package
say .context~package~findNamespace("windows")   -- a Package
::requires "winsystm.cls" namespace windows
```
5.1.5.14. findProgram

Locates program `name` using the target package context and returns the fully resolved filename. `name` must be a string that specifies the filename or path to an external program. The program is searched for using the *external program search order*.

Returns `.nil` if `name` cannot be located.

Example 5.42. Package class — findProgram method

```plaintext
say .context~package~findProgram("mime.cls")  -- e. g. C:\Program Files\ooRexx\mime.cls
```

5.1.5.15. findPublicClass

Returns the public class named `name`. Returns `.nil` if no public class of the specified name exists within the scope of the receiving package.

See also method `findClass`.

Example 5.43. Package class — findPublicClass method

```plaintext
say .context~package~findPublicClass("String")   -- The String class
say .context~package~findPublicClass("Dummy")    -- The NIL object
say .context~package~findPublicClass("RexxInfo") -- The NIL object
::class Dummy       -- private class
```

5.1.5.16. findPublicRoutine

Returns the public routine named `name`. Returns `.nil` if no public routine of the specified name exists within the scope of the receiving package.

See also methods `publicRoutines` and `findRoutine`.

Example 5.44. Package class — findPublicRoutine method

```plaintext
say .context~package~findPublicRoutine("Dummy")~source    -- "nop"
::routine Dummy public
nop
```

5.1.5.17. findRoutine
findRoutine(name)

Searches for a routine within the package search order. This includes ::ROUTE directives within the package, public routines imported from other packages, or routines added using the addRoutine method. The argument name must be a string object. If the name is not found, .nil will be returned.

5.1.5.18. form

form

Returns the initial NUMERIC FORM setting used for all Rexx code contained within the package. The default value is SCIENTIFIC. The ::OPTIONS directive can override the default value.

5.1.5.19. fuzz

fuzz

Returns the initial NUMERIC FUZZ setting used for all Rexx code contained within the package. The default value is 0. The ::OPTIONS directive can override the default value.

5.1.5.20. importedClasses

importedClasses

Returns a StringTable containing all public classes imported from other packages.

See also methods classes and publicClasses.

5.1.5.21. importedPackages

importedPackages

Returns an Array containing all packages imported by the target package.

5.1.5.22. importedRoutines

importedRoutines

Returns a StringTable containing all public routines imported from other packages.

5.1.5.23. loadLibrary

loadLibrary(name)

Loads a native library package and adds it to the list of libraries loaded by the interpreter. The name identifies a native external library file that will be located and loaded as if it had been named on a ::REQUIRES LIBRARY directive. If the library is successfully loaded, loadLibrary will return .true, otherwise it returns .false.
5.1.5.24. loadPackage

Loads a package and adds it to the list of packages loaded by the package manager. If only `name` is specified, `name` identifies a file that will be located and loaded as if it had been named on a `::REQUIRES` directive. The file is searched for using the external program search order.

If `source` is given, it must be an array of strings that is the source for the loaded package.

If a package `name` has already been loaded by the package manager, the previously loaded version will be used.

The resolved package object will be added to the receiving package object's dependent packages.

5.1.5.25. local

Returns a `Directory` of objects local to the receiving package.

See also `.LOCAL` for a Directory of objects local to the interpreter instance.

5.1.5.26. name

Returns the string name of the package.

The package name may be
- the absolute path of the executing program,
- any name that was specified when creating an instance of a Package or Routine class,
- "REXX" for classes defined by Rexx, or
- "INSTORE" for code executed through the `rexx -e` command.

Example 5.45. Package class — name method

```plaintext
say .context~package~name -- e. g. C:\ExamplePackageName.rex
say .Class~package~name -- REXX
say .Routine-new("rtn", "return .context~package~name")[[] -- rtn
say .Package-new("pkg", ")-name -- pkg
```

5.1.5.27. namespaces

Returns a StringTable of all namespaces defined in the target package.

See also method `findNamespace`.
Example 5.46. Package class — namespaces method

```plaintext
say .context~package~namespaces~allIndexes      -- WINDOWS
::requires "winsystm.cls" namespace windows
```

5.1.5.28. prolog

Returns a routine object that represents the code of the target package that precedes any directives.

Example 5.47. Package class — prolog method

```plaintext
say .context~package~prolog~source  -- "say .context~package~prolog~source"
::options noprolog
```

5.1.5.29. publicClasses

Returns a StringTable containing all public classes defined in this package.

See also methods `findPublicClass` and `classes`.

Example 5.48. Package class — publicClasses method

```plaintext
say .package~new("csvstream.cls")~publicClasses~allIndexes -- CSVSTREAM
say .context~package~publicClasses~allIndexes              -- DUMMY
say .rexxinfo~package~publicClasses~items                  -- 56
::class dummy public
```

5.1.5.30. publicRoutines

Returns a StringTable containing all public routines defined in this package. The StringTable indexes are the routine names, the StringTable values are individual routine objects.

See also method `findPublicRoutine`.

Example 5.49. Package class — publicRoutines method

```plaintext
say .package-new("csvstream.cls")-publicRoutines-items  -- 0
say .context-package-publicRoutines-allIndexes         -- DUMMY
say .rexxinfo-package-publicRoutines-items              -- 0
::routine dummy public
```
5.1.5.31. resource

resource(name)

Returns an Array of resources data lines a ::RESOURCE name directive in the target package defines. Returns .nil, if no resource name exists.

See also
- method resources and
- ::RESOURCE directive.

Example 5.50. Package class — resource method

```
say .context~package~resource("GREYCAT")~makeString  -- La nuit, tous les chats sont gris
::resource greyCat
La nuit, tous les chats sont gris
::END
```

5.1.5.32. resources

resources

Returns a StringTable of all data resources that ::RESOURCE directives in the target package define. The StringTable indexes are the resource names, the StringTable values are arrays of individual resource data lines.

See also
- method resource and
- ::RESOURCE directive.

Example 5.51. Package class — resources method

```
say .context~package~resources~allIndexes  -- "BROWN FOX"
-- "GREYCAT"
::resource greyCat
La nuit, tous les chats sont gris
::END
::resource "brown fox"
The quick brown fox jumps over the lazy dog
::END
```

5.1.5.33. routines

routines

Returns a StringTable containing all routines defined in this package. The StringTable indexes are the routine names, the StringTable values are individual routine objects.
5.1.5.34. `setSecurityManager`

```
setSecurityManager(security_manager_object)
```

Replaces the existing security manager with the specified `security_manager_object`. If `security_manager_object` is omitted, any existing security manager is removed.

5.1.5.35. `source`

```
source
```

Returns the package source code as a single-dimensional Array of source lines. If the source code is not available, `source` returns an array of zero items.

5.1.5.36. `sourceLine`

```
sourceLine(n)
```

Returns the `n`th source line from the package source. If the source code is not available or the indicated line does not exist, a null string is returned.

5.1.5.37. `sourceSize`

```
sourceSize
```

Returns the size of the source code for the package object. If the source code is not available, 0 is returned.

5.1.5.38. `trace`

```
trace
```

Returns the initial `TRACE` setting used for all Rexx code contained within the package. The default value is Normal. The `::OPTIONS` directive can override the default value.

5.1.6. Routine Class

The Routine class creates routine objects from Rexx source code. It is a subclass of the `Object class`.

<table>
<thead>
<tr>
<th>Table 5.6. Routine Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
</tr>
<tr>
<td>Methods inherited from the <code>Object class</code></td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the <code>Class class</code></td>
</tr>
<tr>
<td>Routine</td>
</tr>
</tbody>
</table>
5.1.6.1. new (Class Method)

```
new(name, source, context)
```

Returns a new instance of the **Routine** class, which is an executable representation of the code contained in the `source`. The `name` is a string. The `source` can be a single string or an array of strings containing individual method lines.

The `context` allows the created routine to inherit class and routine lookup scope from another source. If specified, `context` can be a Method object, a Routine object, a Package object, or the string "PROGRAMSCOPE". PROGRAMSCOPE is the default, and specifies that the newly created routine will inherit the class and routine search scope from the caller of the new method.

5.1.6.2. newFile (Class Method)

```
newFile(filename, context)
```

Returns a new instance of the **Routine** class, which is an executable representation of the code contained in the file `filename`. The `filename` is a string.

The `context` allows the created routine to inherit class and routine lookup scope from another source. If specified, `context` can be a Method object, a Routine object, a Package object, or the string "PROGRAMSCOPE". PROGRAMSCOPE is the default, and specifies that the newly created routine will inherit the class and routine search scope from the caller of the newFile method.

5.1.6.3. loadExternalRoutine (Class method)

```
loadExternalRoutine(name, descriptor)
```

Resolves a native routine in an external library package and returns a Routine object instance that can be used to call the external routine. The `descriptor` is a string containing whitespace-delimited tokens that identify the location of the native routine. The first token identifies the type of native routine and must be `LIBRARY`. The second token must identify the name of the external library. The external library is located using platform-specific mechanisms for loading libraries. For Unix-based systems, the library name is case-sensitive. The third token is optional and specifies the name of the routine within the library package. If not specified, `name` is used. The routine name is not case sensitive. If the target routine cannot be resolved, `.nil` is returned.

**Example 5.52. Routine class — loadExternalRoutine method**

```ruby
pi = .Routine:loadExternalRoutine("pi", "library rxmath rxcalcpi")
```
5.1.6.4. []

Calls the routine object using the provided arguments. The code in the routine object is called as if it was an external routine call. The return value will be any value returned by the executed routine.

See also method call for which this method is a synonym.

5.1.6.5. annotation

Returns the value of the annotation named name for this routine. If no such annotation exists, .nil is returned.

See also
• method annotations and
• ::ANNOTATE directive.

5.1.6.6. annotations

Returns a StringTable of all annotation name/value pairs for this routine.

See also
• method annotation and
• ::ANNOTATE directive.

5.1.6.7. call

Calls the routine object using the provided arguments. The code in the routine object is called as if it was an external routine call. The return value will be any value returned by the executed routine.

See also method [] for which this method is a synonym.

5.1.6.8. callWith
5.1.6.9. **package**

Returns the Package class instance that defined the routine instance. The package instance controls and defines the search order for classes and routines referenced by the routine code.

5.1.6.10. **setSecurityManager**

Replaces the existing security manager with the specified `security_manager_object`. If `security_manager_object` is omitted, any existing security manager is removed.

5.1.6.11. **source**

Returns the routine source code as a single-dimensional Array of source lines. If the source code is not available, `source` returns an array of zero items.

### 5.1.7. String Class

String objects represent character-string data values. A character string value can have any length and contain any characters.

**Table 5.7. String Class**

<table>
<thead>
<tr>
<th>String</th>
<th>Object</th>
<th>Methods inherited from the <strong>Object class</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ Comparable (Mixin Class)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Methods inherited from the <strong>Comparable class</strong></td>
</tr>
<tr>
<td></td>
<td><code>compareTo</code></td>
<td>compareTo</td>
</tr>
<tr>
<td><code>new</code> (Class Method)</td>
<td><code>digit</code> (Class Method)</td>
<td><code>punct</code> (Class Method)</td>
</tr>
<tr>
<td><code>alnum</code> (Class Method)</td>
<td><code>graph</code> (Class Method)</td>
<td><code>space</code> (Class Method)</td>
</tr>
<tr>
<td><code>alpha</code> (Class Method)</td>
<td><code>lower</code> (Class Method)</td>
<td><code>tab</code> (Class Method)</td>
</tr>
<tr>
<td><code>blank</code> (Class Method)</td>
<td><code>nl</code> (Class Method)</td>
<td><code>upper</code> (Class Method)</td>
</tr>
<tr>
<td><code>cntrl</code> (Class Method)</td>
<td><code>null</code> (Class Method)</td>
<td><code>xdigit</code> (Class Method)</td>
</tr>
<tr>
<td>Arithmetic Methods</td>
<td>Comparison Methods</td>
<td>Concatenation Methods</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>+, -, *, **, /, //, %</td>
<td>===, &lt;, &lt;=, &lt;&lt;=, &gt;=, &gt;, &gt;&gt;, &gt;&gt;=,</td>
<td>(abuttal)</td>
</tr>
</tbody>
</table>

### 5.1.7.1. `new` (Class Method)

```plaintext
new(s t r i n g v a l u e)
```

Returns a new string object initialized with the characters in `stringValue`.

### 5.1.7.2. `alnum` (Class Method)

```plaintext
alnum
```

Returns the string `0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`, a character sequence representing the POSIX character class name **ALNUM** (alphanumeric characters).

See also class methods `alpha` and `digit`. 
5.1.7.3. alpha (Class Method)

Returns the string `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`, a character sequence representing the POSIX character class name `ALPHA` (alphabetic characters).

See also class methods `lower` and `upper`.

5.1.7.4. blank (Class Method)

Returns the string `'09 20'x`, a character sequence representing the POSIX character class name `BLANK` (tab and space character).

See also class method `space`.

5.1.7.5. cntrl (Class Method)

Returns the string `'00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 7F'x`, a character sequence representing the POSIX character class name `CNTRL` (control characters).

See also class method `print (Class Method)`.

5.1.7.6. cr (Class Method)

Returns the single character string for the carriage-return character, which has the value `'0d'x`.

5.1.7.7. digit (Class Method)

Returns the string `0123456789`, a character sequence representing the POSIX character class name `DIGIT` (digits).

See also class method `xdigit`.

5.1.7.8. graph (Class Method)
Returns the string 'graph', a character sequence representing the POSIX character class name GRAPH (visible characters).

See also class method print.

### 5.1.7.9. lower (Class Method)

Returns the string abcdefghijklmnopqrstuvwxyz, a character sequence representing the POSIX character class name LOWER (lowercase letters).

See also class methods upper and alpha.

### 5.1.7.10. nl (Class Method)

Returns the single character string for the line-feed character, which has the value '0a'x.

### 5.1.7.11. null (Class Method)

Returns the single character string for the null character, which has the value '00'x.

### 5.1.7.12. print (Class Method)

Returns the string !"#$%&'()*/+,.-0123456789:;<=?>@ABCDEFGHIJKLMNOPQRSTUVWXYZ\[\]^_` abcdefghijklmnopqrstuvwxyz{|}~, a character sequence representing the POSIX character class name PRINT (visible characters and space character).

See also class method graph.

### 5.1.7.13. punct (Class Method)

Returns the string !"#$%&'()*/+,.-0123456789:;<=?>@ABCDEFGHIJKLMNOPQRSTUVWXYZ\[\]^_` abcdefghijklmnopqrstuvwxyz{|}~, a character sequence representing the POSIX character class name PUNCT (punctuation characters).
5.1.7.14. space (Class Method)

Returns the string '\09 \0A \0B \0C \0D \20'x, a character sequence representing the POSIX character class name SPACE (whitespace characters horizontal tab, line feed, vertical tab, form feed, carriage return, and space).

See also class method blank.

5.1.7.15. tab (Class Method)

Returns the single character string for the tab character, which has the value '\09'x.

5.1.7.16. upper (Class Method)

Returns the stringABCDEFGHIJKLMNOPQRSTUVWXYZ, a character sequence representing the POSIX character class name UPPER (uppercase letters).

See also class method lower.

5.1.7.17. xdigit (Class Method)

Returns the string 0123456789ABCDEFabcdef, a character sequence representing the POSIX character class name XDIGIT (hexadecimal digits).

See also class method digit.

5.1.7.18. Arithmetic Methods

Note

The syntax diagram above is for the non-prefix operators. For the prefix operators, omit the parentheses and argument.
String Class

Returns the result of performing the specified arithmetic operation on the receiver object. The receiver object and the **argument** must be valid **numbers**. The **arithmetic_operator** can be:

+  Addition
-  Subtraction
*  Multiplication
/  Division
%  Integer division (divide and return the integer part of the result)
//  Remainder (divide and return the remainder—not **modulo**, because the result can be negative)
**  Exponentiation (raise a number to a whole-number power)
Prefix -  Same as the subtraction: 0 - **number**
Prefix +  Same as the addition: 0 + **number**

See **Chapter 10, Numbers and Arithmetic** for details about precision, the format of valid numbers, and the operation rules for arithmetic. Note that if an arithmetic result is shown in exponential notation, it might have been rounded.

---

**Example 5.53. String class — arithmetic methods**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5+5</td>
<td>-&gt; 10</td>
</tr>
<tr>
<td>8-5</td>
<td>-&gt; 3</td>
</tr>
<tr>
<td>5*2</td>
<td>-&gt; 10</td>
</tr>
<tr>
<td>6/2</td>
<td>-&gt; 3</td>
</tr>
<tr>
<td>9//4</td>
<td>-&gt; 1</td>
</tr>
<tr>
<td>9%4</td>
<td>-&gt; 2</td>
</tr>
<tr>
<td>2**3</td>
<td>-&gt; 8</td>
</tr>
</tbody>
</table>
| +5  | -> 5  /* Prefix + */
| -5  | -> -5 /* Prefix - */

---

**5.1.7.19. Comparison Methods**

Returns **true** or **false**, the result of performing the specified comparison operation. The receiver object and the **argument** are the terms compared. Both must be string objects. If **argument** is not a string object, it is converted to its string representation for the comparison. The one exception is when **argument** is **nil** for the **==**, **\=\=**, **=**, **\=**, **>**, and **<** operators. A string object will never compare equal to **nil** even when the string matches the string value of **nil** ("The NIL object"). As a result, **==** will always return **false** when compared to **nil** and **\=\=** will always return **true**. All of the relational comparisons (for example, **<**, **>**, **\=<**, etc.) will always return **false** when compared to **nil**.

The comparison operators you can use in a message are:

=  **true** if the terms are equal (for example, numerically or when padded). **false** if **argument** is **nil**.
\=, <, <=  **true** if the terms are not equal (inverse of =). **true** if **argument** is **nil**.
>  Greater than. **false** if **argument** is **nil**.
<  Less than. false if argument is .nil.
>  Greater than or equal to. false if argument is .nil.
\< Not less than. false if argument is .nil.
\> Not greater than. false if argument is .nil.

Example 5.54. String class — comparison methods

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5=5</td>
<td>-&gt;</td>
<td>1</td>
<td>/* equal */</td>
</tr>
<tr>
<td>42=41</td>
<td>-&gt;</td>
<td>1</td>
<td>/* All of these are */</td>
</tr>
<tr>
<td>42&gt;&lt;41</td>
<td>-&gt;</td>
<td>1</td>
<td>/* &quot;not equal&quot; */</td>
</tr>
<tr>
<td>42&lt;&gt;41</td>
<td>-&gt;</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>13&gt;12</td>
<td>-&gt;</td>
<td>1</td>
<td>/* Variations of */</td>
</tr>
<tr>
<td>12&lt;13</td>
<td>-&gt;</td>
<td>1</td>
<td>/* less than and */</td>
</tr>
<tr>
<td>13&gt;=12</td>
<td>-&gt;</td>
<td>1</td>
<td>/* greater than */</td>
</tr>
<tr>
<td>12&lt;13</td>
<td>-&gt;</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>12&lt;=13</td>
<td>-&gt;</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>12&gt;13</td>
<td>-&gt;</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

All strict comparison operations have one of the characters doubled that define the operator. The
== and \== operators check whether two strings match exactly. The two strings must be identical
(character by character) and of the same length to be considered strictly equal.

The strict comparison operators such as >> or << carry out a simple character-by-character
comparison. There is no padding of either of the strings being compared. The comparison of the two
strings is from left to right. If one string is shorter than and a leading substring of another, then it is
smaller than (less than) the other. The strict comparison operators do not attempt to perform a numeric
comparison on the two operands.

For all the other comparison operators, if both terms are numeric, the String class does a numeric
comparison (ignoring, for example, leading zeros—see Section 10.4, “Numeric Comparisons”).
Otherwise, it treats both terms as character strings, ignoring leading and trailing whitespace characters
and padding the shorter string on the right with blanks.

Character comparison and strict comparison operations are both case-sensitive, and for both the exact
collating order can depend on the character set. In an ASCII environment, the digits are lower than
the alphabetic characters, and lowercase alphabetic characters are higher than uppercase alphabetic
characters.

The strict comparison operators you can use in a message are:

== .true if terms are strictly equal (identical)
\== .true if the terms are NOT strictly equal (inverse of ==)
>> Strictly greater than
<< Strictly less than
>>= Strictly greater than or equal to
\<< Strictly NOT less than
\<<= Strictly less than or equal to
**Example 5.55. String class — comparison methods**

```
"space"=="space"   ->  1      /* Strictly equal */
"space\"==" space"  ->  1      /* Strictly not equal */
"space">>" space"  ->  1      /* Variations of */
" space"<<<"space"  ->  1      /* strictly greater */
"space"\<<" space"  ->  1
" space"<<="space"  ->  1
" space">>"space"   ->  1
```

**5.1.7.20. Logical Methods**

```

Note

For NOT (prefix \), omit the parentheses and argument.

Returns .true or .false, the result of performing the specified logical operation. The receiver object and the argument are character strings that evaluate to 1 or 0.

The logical operator can be:

\&            AND (Returns 1 if both terms are .true.)
|              Inclusive OR (Returns 1 if either term or both terms are true.)
&&             Exclusive OR (Returns 1 if either term, but not both terms, is .true.)
Prefix \      Logical NOT (Negates; 1 becomes 0, and 0 becomes 1.)
```

**Example 5.56. String class — logical methods**

```
1&0        ->  0
1|0        ->  1
1&&0      ->  1
\1        ->  0
```

**5.1.7.21. Concatenation Methods**

```

Concatenates the receiver object with argument. (See Section 1.11.2.1, "String Concatenation".) The concatenation operator can be:
```

176
concatenates without an intervening blank. The abuttal operator "" is the null string. The language processor uses the abuttal to concatenate two terms that another operator does not separate.

||
|| concatenates without an intervening blank.

" " concatenates with one blank between the receiver object and the argument. (The operator " " is a blank.)

Example 5.57. String class — concatenation methods

```
f = "abc"
f"def"    ->  "abcdef"
f || "def"  ->  "abcdef"
f "def"    ->  "abc def"
```

5.1.7.22. []

If `length` is omitted, returns the character at position `n` of the receiving string. If `n` is larger than the length of the receiving string, a null string is returned.

If `length` is specified, returns the substring of the receiving string starting at position `n` and of length `length`. No padding occurs for any portion of the returned string not within the boundaries of the receiving string.

The `n` must be a positive whole number, and, if specified, `length` must be a a non-negative whole number.

See also methods `subChar` and `substr`.

Example 5.58. String class — [] method

```
"abc"[2]        --  "b"
"abc"[2,4]      --  "bc"
```

5.1.7.23. ? (inline if)

```
? (first, second)
```

Returns `first` if the receiving string is `.true`, returns `second` if it is `.false`.

Example 5.59. String class — ? method

```
do apples = 0 to 2
   say apples (apples = 1)-?("apple", "apples")
end
```

" 0 apples " 1 apple --
-- 2 apples
### 5.1.7.24. `abbrev`

**Abbreviation**

`abbrev(info, length)`

Returns 1 if `info` is equal to the leading characters of the receiving string and the length of `info` is not less than `length`. Returns 0 if either of these conditions is not met.

If you specify `length`, it must be a positive whole number or zero. The default for `length` is the number of characters in `info`.

**Example 5.60. String class — abbrev method**

```
"Print"~abbrev("Pri")      ->    1
"PRINT"~abbrev("Pri")      ->    0
"PRINT"~abbrev("PRI",4)     ->    0
"PRINT"~abbrev("PRY")      ->    0
"PRINT"~abbrev(""")        ->    1
"PRINT"~abbrev("",1)        ->    0
```

**Note**

A null string always matches if a length of 0, or the default, is used. This allows a default keyword to be selected automatically if desired.

**Example 5.61. String class — abbrev method**

```
say "Enter option:";   pull option .
select /* keyword1 is to be the default */
  when "keyword1"~abbrev(option) then ... 
  when "keyword2"~abbrev(option) then ...
  ...
  otherwise nop;
end;
```

### 5.1.7.25. `abs`

**Absolute Value**

`abs(string)`

Returns the absolute value of the receiving string. The result has no sign and is formatted according to the current NUMERIC settings.

**Example 5.62. String class — abs method**

```
12.3~abs      ->    12.3
"-0.307"~abs  ->    0.307
```

---

178
5.1.7.26. append

Returns a string consisting of string appended to the receiving string.

5.1.7.27. b2x

Returns a string, in character format, that represents the receiving binary string converted to hexadecimal.

The receiving string is a string of binary (0 or 1) digits. It can be of any length. It can optionally include whitespace characters (at 4-digit boundaries only, not leading or trailing). These are to improve readability and are ignored.

The returned string uses uppercase alphabetic characters for the values A-F and does not include whitespace.

If the receiving binary string is a null string, b2x returns a null string. If the number of binary digits in the receiving string is not a multiple of four, up to three 0 digits are added on the left before the conversion to make a total that is a multiple of four.

Example 5.63. String class — b2x method

<table>
<thead>
<tr>
<th>Binary String</th>
<th>b2x</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;11000011&quot;</td>
<td>b2x</td>
<td>&quot;C3&quot;</td>
</tr>
<tr>
<td>&quot;10111&quot;</td>
<td>b2x</td>
<td>&quot;17&quot;</td>
</tr>
<tr>
<td>&quot;101&quot;</td>
<td>b2x</td>
<td>&quot;5&quot;</td>
</tr>
<tr>
<td>&quot;1 1111 0000&quot;</td>
<td>b2x</td>
<td>&quot;1F0&quot;</td>
</tr>
</tbody>
</table>

You can combine b2x with the methods x2d and x2c to convert a binary number into other forms.

Example 5.64. String class — b2x method with x2d

"1011" b2x x2d -> "23" /* decimal 23 */

5.1.7.28. bitAnd

Returns a string composed of the receiver string and the argument string logically ANDed together, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If you omit the pad character, the AND operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If you provide pad, it extends the shorter of the two strings on the right before the logical operation. The default for string is the zero-length (null) string.
String Class

Example 5.65. String class — bitand method

```
"12"x~bitAnd                   ->    "12"x
"73"x~bitAnd("27"x)           ->    "23"x
"13"x~bitAnd("5555"x)         ->    "1155"x
"13"x~bitAnd("5555"x,"74"x)   ->    "1154"x
"pQrS"~bitAnd(,"DF"x)         ->    "PQRS"      /* ASCII   */
```

5.1.7.29. bitOr

```
bitOr(
  string,
  pad
)
```

Returns a string composed of the receiver string and the argument `string` logically inclusive-ORed, bit by bit. The encodings of the strings are used in the logical operation. The length of the result is the length of the longer of the two strings. If you omit the `pad` character, the OR operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If you provide `pad`, it extends the shorter of the two strings on the right before the logical operation. The default for `string` is the zero-length (null) string.

Example 5.66. String class — bitor method

```
"12"x~bitOr                   ->    "12"x
"15"x~bitOr("24"x)           ->    "35"x
"15"x~bitOr("2456"x)         ->    "3556"x
"15"x~bitOr("2456"x,"F0"x)   ->    "35F6"x
"1111"x~bitOr(,"4D"x)        ->    "5D5D"x
"pQrS"~bitOr(,"20"x)         ->    "pqrs" /* ASCII   */
```

5.1.7.30. bitXor

```
bitXor(
  string,
  pad
)
```

Returns a string composed of the receiver string and the argument `string` logically eXclusive-ORed, bit by bit. The encodings of the strings are used in the logical operation. The length of the result is the length of the longer of the two strings. If you omit the `pad` character, the XOR operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If you provide `pad`, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for `string` is the zero-length (null) string.

Example 5.67. String class — bitxor method

```
"12"x~bitXor                      ->  "12"x
"12"x~bitXor("22"x)               ->  "30"x
"1211"x~bitXor("22"x)             ->  "3011"x
```

180
5.1.7.31. c2d

Returns the decimal value of the binary representation of the receiving string. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS. If you specify n, it is the length of the returned result. If you do not specify n, the receiving string is processed as an unsigned binary number. If the receiving string is null, C2D returns 0.

Example 5.68. String class — c2d method

<table>
<thead>
<tr>
<th>String</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;09&quot;X</td>
<td>9</td>
</tr>
<tr>
<td>&quot;81&quot;X</td>
<td>129</td>
</tr>
<tr>
<td>&quot;FF81&quot;X</td>
<td>65409</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;a&quot;X</td>
<td>97</td>
</tr>
</tbody>
</table>

If you specify n, the receiving string is taken as a signed number expressed in n characters. The number is positive if the leftmost bit is off, and negative if the leftmost bit is on. In both cases, it is converted to a whole number, which can therefore be negative. The receiving string is padded on the left with "00"x characters (not "sign-extended"), or truncated on the left to n characters. This padding or truncation is as though receiving_string-right(n, '00'x) had been processed. If n is 0, c2d always returns 0.

Example 5.69. String class — c2d method

<table>
<thead>
<tr>
<th>String</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;01&quot;X-c2d(1)</td>
<td>-127</td>
</tr>
<tr>
<td>&quot;01&quot;X-c2d(2)</td>
<td>129</td>
</tr>
<tr>
<td>&quot;FF81&quot;X-c2d(2)</td>
<td>-127</td>
</tr>
<tr>
<td>&quot;FF81&quot;X-c2d(1)</td>
<td>-127</td>
</tr>
<tr>
<td>&quot;FF7F&quot;X-c2d(1)</td>
<td>127</td>
</tr>
<tr>
<td>&quot;F081&quot;X-c2d(2)</td>
<td>-3967</td>
</tr>
<tr>
<td>&quot;F081&quot;X-c2d(1)</td>
<td>-127</td>
</tr>
<tr>
<td>&quot;0031&quot;X-c2d(0)</td>
<td>0</td>
</tr>
</tbody>
</table>

5.1.7.32. c2x

Returns a string, in character format, that represents the receiving string converted to hexadecimal. The returned string contains twice as many bytes as the receiving string. On an ASCII system, sending a c2x message to the receiving string 1 returns 31 because "31"X is the ASCII representation of 1.
The returned string has uppercase alphabetic characters for the values A-F and does not include whitespace. The receiving string can be of any length. If the receiving string is null, c2x returns a null string.

Example 5.70. String class — c2x method

```
"0123" ~ c2x -> "0123" /* "30313233"X in ASCII */
"ZD8" ~ c2x -> "5A4438" /* "354134343338"X in ASCII */
```

5.1.7.33. caselessAbbrev

```
caselessAbbrev(info, length)
```

Returns 1 if info is equal to the leading characters of the receiving string and the length of info is not less than length. Returns 0 if either of these conditions is not met. The characters are tested using a caseless comparison.

If you specify length, it must be a positive whole number or zero. The default for length is the number of characters in info.

Example 5.71. String class — caselessAbbrev method

```
"Print" ~ caselessAbbrev("Pri") -> 1
"PRINT" ~ caselessAbbrev("Pri") -> 1
"PRINT" ~ caselessAbbrev("PRI", 4) -> 0
"PRINT" ~ caselessAbbrev("PRY") -> 0
"PRINT" ~ caselessAbbrev("") -> 1
"PRINT" ~ caselessAbbrev("", 1) -> 0
```

Note

A null string always matches if a length of 0, or the default, is used. This allows a default keyword to be selected automatically if desired.

Example 5.72. String class — caselessAbbrev method

```
say "Enter option:"; parse pull option.
select /* keyword1 is to be the default */
when "keyword1" ~ caselessAbbrev(option) then ...
when "keyword2" ~ caselessAbbrev(option) then ...
...
otherwise nop;
end;
```
5.1.7.34. caselessChangeStr

Returns a copy of the receiver object in which newneedle replaces occurrences of needle. If count is not specified, all occurrences of needle are replaced. If count is specified, it must be a non-negative, whole number that gives the maximum number of occurrences to be replaced. The needle searches are performed using caseless comparisons.

Here are some examples:

Example 5.73. String class — caselessChangeStr method

"AbaAbb"~caselessChangeStr("A","") -> "bbb"
AbaBabAB~changeStr("ab","xy") -> "xyxyxyxy"
AbaBabAB~changeStr("ab","xy",1) -> "xyaBabAB"

5.1.7.35. caselessCompare

Returns 0 if the argument string is identical to the receiving string using a caseless comparison. Otherwise, returns the position of the first character that does not match. The shorter string is padded on the right with pad if necessary. The default pad character is a blank.

Example 5.74. String class — caselessCompare method

"abc"~caselessCompare("ABC") -> 0
"abc"~caselessCompare("Ak") -> 2
"ab "~caselessCompare("AB") -> 0
"AB "~caselessCompare("ab"," ") -> 0
"ab "~caselessCompare("ab","x") -> 3
"abXX "~caselessCompare("ab","x") -> 5

5.1.7.36. caselessCompareTo

Performs a caseless sort comparison of the target string to the string argument. If the two strings are equal, 0 is returned. If the target string is larger, 1 is returned. -1 if the string argument is the larger string. The comparison is performed starting at character n for length characters in both strings. n must be a positive whole number. If n is omitted, the comparison starts at the first character. length must be a non-negative whole number. If omitted, the comparison will take place to the end of the target string.
Example 5.75. String class — caselessCompareTo method

<table>
<thead>
<tr>
<th>String</th>
<th>caselessCompareTo</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot;</td>
<td>&quot;abc&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;b&quot;</td>
<td>&quot;a&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>&quot;b&quot;</td>
<td>-1</td>
</tr>
<tr>
<td>&quot;abc&quot;</td>
<td>&quot;aBc&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;aBc&quot;</td>
<td>&quot;abc&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;000abc000&quot;</td>
<td>&quot;111abc111&quot;, 4, 3</td>
<td>0</td>
</tr>
</tbody>
</table>

5.1.7.37. caselessContains

```ruby
caselessContains(other, start, length)
```

Returns `.true` if the receiving string contains the `other` string. It returns `.false` if `other` is the null string or is not found within the receiving string. The search is performed using caseless comparisons.

By default, the search starts at the first character of the receiving string and continues to the end. You can override this by specifying `start`, the point at which the search starts, and `length`, the bounding limit for the search.

If specified, `start` must be a positive whole number and `length` must be a non-negative whole number.

See also methods `contains`, `caselessStartsWith`, `caselessEndsWith`, and `caselessPos`.

Example 5.76. String class — caselessContains method

```ruby
say "-abcdef-"~caselessContains("EF")         -- 1
say "-abcdef-"~caselessContains(".-", 2, 6)    -- 0
```

5.1.7.38. caselessContainsWord

```ruby
caselessContainsWord(phrase, start)
```

Returns `.true` if `phrase` is found in the receiving string. Returns `.false` if `phrase` contains no words or if `phrase` is not found. Word matches are made independent of case. Multiple whitespace characters between words in either `phrase` or the receiving string are treated as a single blank for the comparison, but, otherwise, the words must match, except for case.

By default the search starts at the first word in the receiving string. You can override this by specifying `start` (which must be a positive whole number), the word at which the search is to be started.

See also methods `containsWord` and `caselessWordPos` (caselessContainsWord returns `.false` exactly if caselessWordPos would have returned "0").

Example 5.77. String class — caselessContainsWord method

```ruby
good = "Now is the time for all good men"
```
5.1.7.39. \texttt{caselessCountStr}

\begin{tikzpicture}
  \node[draw, rectangle, minimum height=1em] at (0,0) {\texttt{caselessCountStr}}; \node[above=0.2em] at (0,0) {\texttt{needle}}; \node at (1,0) {\texttt{}};
\end{tikzpicture}

Returns a count of the occurrences of \textit{needle} in the receiving string that do not overlap. All matches are made using caseless comparisons.

Here are some examples:

Example 5.78. String class — \texttt{caselessCountStr} method

\begin{verbatim}
"a0Aa0A"-\texttt{caselessCountStr("a")} -> 4
"J0kKk0"-\texttt{caselessCountStr("KK")} -> 1
\end{verbatim}

5.1.7.40. \texttt{caselessEndsWith}

\begin{tikzpicture}
  \node[draw, rectangle, minimum height=1em] at (0,0) {\texttt{caselessEndsWith}}; \node[above=0.2em] at (0,0) {\texttt{other}}; \node at (1,0) {\texttt{}};
\end{tikzpicture}

Returns \texttt{.true} if the characters of the \textit{other} match the characters at the end of the target string. Returns \texttt{.false} if the characters are not a match, or if \textit{other} is the null string. The match is made using caseless comparisons.

The \texttt{caselessEndsWith} method is useful for efficient string parsing as it does not require new string objects be extracted from the target string.

See also methods \texttt{caselessStartsWith, endsWith}, and \texttt{caselessMatch}.

5.1.7.41. \texttt{caselessEquals}

\begin{tikzpicture}
  \node[draw, rectangle, minimum height=1em] at (0,0) {\texttt{caselessEquals}}; \node[above=0.2em] at (0,0) {\texttt{other}}; \node at (1,0) {\texttt{}};
\end{tikzpicture}

Returns \texttt{.true} if the target string is strictly equal to the \textit{other} string, using a caseless comparison. Returns \texttt{.false} if the two strings are not strictly equal.

Example 5.79. String class — \texttt{caselessEquals} method

\begin{verbatim}
"a"-\texttt{caselessEquals("A")} -> 1
"aa"-\texttt{caselessEquals("A")} -> 0
"4"-\texttt{caselessEquals("3")} -> 0
\end{verbatim}
5.1.7.42. **caselessLastPos**

Returns the position of the last occurrence of a string, *needle*, in the receiving string. It returns 0 if *needle* is the null string or not found. By default, the search starts at the last character of the receiving string and scans backward to the beginning of the string. You can override this by specifying *start*, the point at which the backward scan starts and *length*, the range of characters to scan. The *start* must be a positive whole number and defaults to *receiving_string−length* if larger than that value or omitted. The *length* must be a non-negative whole number and defaults to *start*. The search is performed using caseless comparisons.

See also methods *lastPos*, *caselessPos*, and *pos*.

Example 5.80. String class — caselessLastPos method

```
"abc def ghi"~caselessLastPos(" ")      ->    8
"abcdefghi"~caselessLastPos(" ")      ->    0
"efgxyz"~caselessLastPos("XY")         ->    4
"abc def ghi"~caselessLastPos(" ",7)    ->    4
"abc def ghi"~caselessLastPos(" ",7,3)  ->    0
```

5.1.7.43. **caselessMatch**

Returns `.true.` if the characters of the *other* match the characters of the target string beginning at position *start*. Returns `.false.` if the characters are not a match. The matching is performed using caseless comparisons. *start* must be a positive whole number.

If *n* is specified, the match will be performed starting with character *n* of *other*. The default value for *n* is "1". *n* must be a positive whole number less than or equal to the length of *other*.

If *length* is specified, it defines a substring of *other* that is used for the match. *length* must be a positive whole number and the combination of *n* and *length* must be a valid substring within the bounds of *other*.

The caselessMatch method is useful for efficient string parsing as it does not require new string objects be extracted from the target string.

Example 5.81. String class — caselessMatch method

```
"Saturday"~caselessMatch(6, "day")         ->    1
"Saturday"~caselessMatch(6, "DAY")         ->    1
"Saturday"~caselessMatch(6, "SUNDAY", 4, 3) ->    1
"Saturday"~caselessMatch(6, "daytime", 1, 3) ->    1
```
5.1.7.44. caselessMatchChar

Returns `.true.` if the character at position `n` matches any character of the string `chars`. Returns `.false.` if the character does not match any of the characters in the reference set. The match is made using caseless comparisons. The argument `n` must be a positive whole number.

Example 5.82. String class — caselessMatchChar method

<table>
<thead>
<tr>
<th>String</th>
<th>caselessMatchChar(2, &quot;+-*/&quot;)</th>
<th>caselessMatchChar(1, &quot;+-*/&quot;)</th>
<th>Friday&quot;-caselessMatchChar(3, &quot;aeiou&quot;)</th>
<th>FRIDAY&quot;-caselessMatchChar(3, &quot;aeiou&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;a+b&quot;</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>&quot;a+b&quot;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

5.1.7.45. caselessPos

Returns the position in the receiving string of another string, `needle`. It returns 0 if `needle` is the null string or is not found or if `start` is greater than the length of the receiving string. The search is performed using caseless comparisons. By default, the search starts at the first character of the receiving string (that is, the value of `start` is 1), and continues to the end of the string. You can override this by specifying `start`, the point at which the search starts, and `length`, the bounding limit for the search. If specified, `start` must be a positive whole number and `length` must be a non-negative whole number.

See also methods `pos` and `caselessLastPos`.

Example 5.83. String class — caselessPos method

<table>
<thead>
<tr>
<th>String</th>
<th>caselessPos(&quot;DAY&quot;)</th>
<th>caselessPos(&quot;x&quot;)</th>
<th>caselessPos(&quot; &quot;)</th>
<th>caselessPos(&quot; &quot;,5)</th>
<th>caselessPos(&quot; &quot;,5,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Saturday&quot;</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>&quot;abc def ghi&quot;</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

5.1.7.46. caselessStartsWith

Returns `.true.` if the characters of the `other` match the characters at the start of the target string. Returns `.false.` if the characters are not a match, or if `other` is the null string. The match is made using caseless comparisons.

The caselessStartsWith method is useful for efficient string parsing as it does not require new string objects be extracted from the target string.
See also methods startsWith, caselessEndsWith, and caselessMatch.

5.1.7.47. caselessWordPos

Returns the word number of the first word of phrase found in the receiving string, or 0 if phrase contains no words or if phrase is not found. Word matches are made independent of case. Several whitespace characters between words in either phrase or the receiving string are treated as a single blank for the comparison, but, otherwise, the words must match exactly.

By default the search starts at the first word in the receiving string. You can override this by specifying start (which must be positive), the word at which the search is to be started.

Example 5.84. String class — caselessWordPos method

```
"now is the time"~caselessWordPos("the")    -> 3
"now is the time"~caselessWordPos("The")    -> 3
"now is the time"~caselessWordPos("IS THE") -> 2
"now is the time"~caselessWordPos("is the") -> 2
"now is the time"~caselessWordPos("is time ") -> 0
"To be or not to be"~caselessWordPos("BE")  -> 2
"To be or not to be"~caselessWordPos("BE",3) -> 6
```

5.1.7.48. ceiling

Returns the largest integer not less than the receiving string value. The receiving string is first rounded according to standard Rexx rules, as though the operation receiving_string+0 had been carried out. The ceiling is then calculated from that result and returned. The result is never in exponential form. If there are no nonzero digits in the result, any minus sign is removed.

Example 5.85. String class — ceiling method

```
2-ceiling    ->  2
'-2'-ceiling ->  -2
12.3-ceiling ->  13
'-12.3'-ceiling -> -12
'-0.1'-ceiling ->  0
```
The number is rounded according to the current setting of NUMERIC DIGITS if necessary, before the method processes it.

5.1.7.49. center/centre

Returns a string of length length with the receiving string centered in it. The pad characters are added as necessary to make up length. The length must be a positive whole number or zero. The default pad character is blank. If the receiving string is longer than length, it is truncated at both ends to fit. If an odd number of characters are truncated or added, the right-hand end loses or gains one more character than the left-hand end.

Example 5.86. String class — center method

```
abc~center(7)               ->    "  ABC  
abc~CENTER(8,"-")           ->    "--ABC---
"The blue sky"~centre(8)    ->    "e blue s"
"The blue sky"~centre(7)    ->    "e blue 
```

5.1.7.50. changeStr

Returns a copy of the receiver object in which newneedle replaces occurrences of needle.

If count is not specified, all occurrences of needle are replaced. If count is specified, it must be a non-negative, whole number that gives the maximum number of occurrences to be replaced.

Here are some examples:
Example 5.87. String class — changeStr method

<table>
<thead>
<tr>
<th>String</th>
<th>Change Method</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;1&quot;</td>
<td>changeStr(&quot;1&quot;,&quot;&quot;)</td>
<td>&quot;000&quot;</td>
</tr>
<tr>
<td>&quot;1&quot;</td>
<td>changeStr(&quot;1&quot;,&quot;x&quot;)</td>
<td>&quot;X0XX00&quot;</td>
</tr>
<tr>
<td>&quot;1&quot;</td>
<td>changeStr(&quot;1&quot;,&quot;x&quot;,1)</td>
<td>&quot;X01100&quot;</td>
</tr>
</tbody>
</table>

-- a Quine: will print an exact copy of itself
-- (see https://en.wikipedia.org/wiki/Quine_%28computing%29)

r=";say'r='r.'r~changeStr(.,'22'x,2);say'r='r.'r~changeStr(.,'22'x,2)

5.1.7.51. compare

```plaintext
compare(string, pad)
```

Returns 0 if the argument `string` is identical to the receiving string. Otherwise, returns the position of the first character that does not match. The shorter string is padded on the right with `pad` if necessary. The default `pad` character is a blank.

Example 5.88. String class — compare method

<table>
<thead>
<tr>
<th>String</th>
<th>Compare Method</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot;</td>
<td>compare(&quot;abc&quot;)</td>
<td>0</td>
</tr>
<tr>
<td>&quot;abc&quot;</td>
<td>compare(&quot;ak&quot;)</td>
<td>2</td>
</tr>
<tr>
<td>&quot;ab&quot;</td>
<td>compare(&quot;ab&quot;)</td>
<td>0</td>
</tr>
<tr>
<td>&quot;ab&quot;</td>
<td>compare(&quot;ab&quot;,&quot; &quot;)</td>
<td>0</td>
</tr>
<tr>
<td>&quot;ab&quot;</td>
<td>compare(&quot;ab&quot;,&quot;x&quot;)</td>
<td>3</td>
</tr>
<tr>
<td>&quot;ab--&quot;</td>
<td>compare(&quot;ab&quot;,&quot;-&quot;)</td>
<td>5</td>
</tr>
</tbody>
</table>

5.1.7.52. compareTo

```plaintext
compareTo(string, n, length)
```

Performs a sort comparison of the target string to the `string` argument. If the two strings are equal, 0 is returned. If the target string is larger, 1 is returned. -1 if the `string` argument is the larger string. The comparison is performed starting at character `n` for `length` characters in both strings. `n` must be a positive whole number. If `n` is omitted, the comparison starts at the first character. `length` must be a non-negative whole number. If omitted, the comparison will take place to the end of the target string.

Example 5.89. String class — compareTo method

<table>
<thead>
<tr>
<th>String</th>
<th>CompareTo Method</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot;</td>
<td>compareTo(&quot;abc&quot;)</td>
<td>0</td>
</tr>
<tr>
<td>&quot;b&quot;</td>
<td>compareTo(&quot;a&quot;)</td>
<td>1</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>compareTo(&quot;b&quot;)</td>
<td>-1</td>
</tr>
<tr>
<td>&quot;abc&quot;</td>
<td>compareTo(&quot;aBc&quot;)</td>
<td>1</td>
</tr>
<tr>
<td>&quot;aBc&quot;</td>
<td>compareTo(&quot;abc&quot;)</td>
<td>-1</td>
</tr>
<tr>
<td>&quot;000abc000&quot;</td>
<td>compareTo(&quot;111abc111&quot;, 4, 3)</td>
<td>0</td>
</tr>
</tbody>
</table>
5.1.7.53. contains

Returns \texttt{true} if the receiving string contains the \texttt{other} string. It returns \texttt{false} if \texttt{other} is the null string or is not found within the receiving string.

By default, the search starts at the first character of the receiving string and continues to the end. You can override this by specifying \texttt{start}, the point at which the search starts, and \texttt{length}, the bounding limit for the search. If specified, \texttt{start} must be a positive whole number and \texttt{length} must be a non-negative whole number.

See also
• method \texttt{caselessContains},
• method \texttt{startsWith},
• method \texttt{endsWith}, and
• method \texttt{pos}.

Example 5.90. String class — caselessContains method

\begin{verbatim}
say "-abcdef-"~contains("ef")         -- 1
say "-abcdef-"~contains("-", 2, 6)    -- 0
\end{verbatim}

5.1.7.54. containsWord

Returns \texttt{true} if \texttt{phrase} is found in the receiving string. Returns \texttt{false} if \texttt{phrase} contains no words or if \texttt{phrase} is not found. Multiple whitespace characters between words in either \texttt{phrase} or the receiving string are treated as a single blank for the comparison, but, otherwise, the words must match exactly.

By default the search starts at the first word in the receiving string. You can override this by specifying \texttt{start} (which must be positive whole number), the word at which the search is to be started.

See also
• method \texttt{caselessContainsWord} and
• method \texttt{wordPos} (containsWord returns \texttt{false} exactly if wordPos would have returned "0").

Example 5.91. String class — containsWord method

\begin{verbatim}
good = "Now is the time for all good men"
say good-containsWord("the")          -- .true
say good-containsWord("The")          -- .false
say good-containsWord("is the")       -- .true
say good-containsWord("is the ")      -- .true
say good-containsWord("is time")      -- .false
say good-containsWord("time")         -- .true
say good-containsWord("time", 5)      -- .false
\end{verbatim}
5.1.7.55. copies

Returns \( n \) concatenated copies of the receiving string. The \( n \) must be a positive whole number or zero.

Example 5.92. String class — copies method

```
"abc"-copies(3)    ->    "abcabcabc"
"abc"-copies(0)    ->    ""
```

5.1.7.56. countStr

Returns a count of the occurrences of \( needle \) in the receiving string that do not overlap.

Here are some examples:

Example 5.93. String class — countStr method

```
"101101"-countStr("1")        ->    4
"J0KK0"-CountStr("KK")       ->    1
```

5.1.7.57. d2c

Returns a string, in character format, that is the ASCII representation of the receiving string, a decimal number. If you specify \( n \), it is the length of the final result in characters; leading blanks are added to the returned string. The \( n \) must be a positive whole number or zero.

The receiving string must not have more digits than the current setting of NUMERIC DIGITS.

If you omit \( n \), the receiving string must be a positive whole number or zero, and the result length is as needed. Therefore, the returned result has no leading "00"x characters.

Example 5.94. String class — d2c method

```
"65"-d2c       ->   "A"      /* "41"x is an ASCII "A"    */
"65"-d2c(1)    ->   "A"      /* "60"x is an ASCII "A"    */
"65"-d2c(2)    ->   "A"      /* "60x is an ASCII "A"    */
"65"-d2c(5)    ->   "A"      /* "60"x is an ASCII "A"    */
"109"-d2c      ->   "m"      /* "41"x is an ASCII "m"    */
"-109"-d2c(1)  ->   "ô"      /* "93"x is an ASCII "ô"    */
```

192
5.1.7.58. d2x

Returns a string, in character format, that represents the receiving string, a decimal number converted to hexadecimal. The returned string uses uppercase alphabetic characters for the values A-F and does not include whitespace.

The receiving string must not have more digits than the current setting of NUMERIC DIGITS.

If you specify \( n \), it is the length of the final result in characters. After conversion the returned string is sign-extended to the required length. If the number is too big to fit into \( n \) characters, it is truncated on the left. If you specify \( n \), it must be a positive whole number or zero.

If you omit \( n \), the receiving string must be a positive whole number or zero, and the returned result has no leading zeros.

Example 5.95. String class — d2x method

```
"76"~d2c(2)  ->  "L"    /* "4C"x is an ASCII "L" */
"-180"~d2c(2) ->  "L"
```

5.1.7.59. dataType

Returns NUM if you specify no argument and the receiving string is a valid Rexx number that can be added to 0 without error. It returns CHAR if the receiving string is not a valid number.

If you specify type, it returns .true if the receiving string matches the type. Otherwise, it returns .false. If the receiving string is null, the method returns .false (except when the type is X or B, for which dataType returns .true for a null string). The following are valid types. You need to specify only the capitalized letter, or 9 for the 9Digits option. The language processor ignores all characters following it.

Alphanumeric
   returns .true if the receiving string contains only characters from the ranges a-z, A-Z, and 0-9.
Binary
returns .true if the receiving string contains only the characters 0 or 1, or whitespace. Whitespace characters can appear only between groups of 4 binary characters. It also returns .true if string is a null string, which is a valid binary string.

Internal whole number
returns .true if the receiving string is a Rexx whole number that built-in functions will accept. Rexx built-in functions internally work with NUMERIC DIGITS 9 for 32-bit systems or NUMERIC DIGITS 18 for 64-bit systems.

Lowercase
returns .true if the receiving string contains only characters from the range a-z.

Mixed case
returns .true if the receiving string contains only characters from the ranges a-z and A-Z.

Number
returns .true if receiving_string~dataType returns NUM.

Logical
returns .true if the receiving string is exactly 0 or 1. Otherwise it returns .false.

Symbol
returns .true if the receiving string is a valid symbol, that is, if SYMBOL(receiving_string) does not return BAD. See also Section 1.10.4.4, “Symbols”. Note that both uppercase and lowercase alphabetic characters are permitted.

Uppercase
returns .true if the receiving string contains only characters from the range A-Z.

Variable
returns .true if the receiving string could appear on the left-hand side of an assignment without causing a SYNTAX condition.

Whole number
returns .true if the receiving string is a whole number under the current setting of NUMERIC DIGITS.

Hexadecimal
returns .true if the receiving string contains only characters from the ranges a-f, A-F, 0-9, and whitespace characters (as long as whitespace characters appear only between pairs of hexadecimal characters). Also returns .true if the receiving string is a null string.

9 Digits
returns .true if receiving_string~dataType("W") returns .true when NUMERIC DIGITS is set to 9.

Example 5.96. String class — datatype method

| "12"~dataType                     | ->  | "NUM" |
| ""~dataType                       | ->  | "CHAR" |
| "123"~dataType                    | ->  | "CHAR" |
| "12.3"~dataType("N")             | ->  | 1     |
| "12.3"~dataType("W")             | ->  | 0     |
String Class

<table>
<thead>
<tr>
<th>String</th>
<th>dataType</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Fred&quot;</td>
<td>&quot;M&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;M&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;Fred&quot;</td>
<td>&quot;L&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;?20K&quot;</td>
<td>&quot;s&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;BCd3&quot;</td>
<td>&quot;X&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;BC d3&quot;</td>
<td>&quot;X&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;1&quot;</td>
<td>&quot;O&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;11&quot;</td>
<td>&quot;O&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

Note

The `dataType` method tests the meaning or type of characters in a string, independent of the encoding of those characters (for example, ASCII or EBCDIC).

5.1.7.60. decodeBase64

Returns a new string containing the decoded version of the base64 encoded receiving string. If the receiving string is not in base64 format, an error results.

Example 5.97. String class — decodeBase64 method

```
"YWJjZGVm"~decodeBase64 -> "abcdef"
```

5.1.7.61. delStr

Returns a copy of the receiving string after deleting the substring that begins at the nth character and is of length characters. If n is omitted, it defaults to 1. If you omit length, or if length is greater than the number of characters from n to the end of string, the method deletes the rest of string (including the nth character). The length must be a positive whole number or zero. The n must be a positive whole number. If n is greater than the length of the receiving string, the method returns the receiving string unchanged.

Example 5.98. String class — delStr method

```
"abcd"~delStr(3) -> "ab"
"abcde"~delStr(3,2) -> "abe"
"abcde"~delStr(6) -> "abcde"
```
5.1.7.62. delWord

Returns a copy of the receiving string after deleting the substring that starts at the \( n \)th word and is of \( \text{length} \) whitespace-delimited words. If you omit \( \text{length} \), or if \( \text{length} \) is greater than the number of words from \( n \) to the end of the receiving string, the method deletes the remaining words in the receiving string (including the \( n \)th word). The \( \text{length} \) must be a positive whole number or zero. The \( n \) must be a positive whole number. If \( n \) is greater than the number of words in the receiving string, the method returns the receiving string unchanged. The string deleted includes any whitespace characters following the final word involved but none of the whitespace characters preceding the first word involved.

Example 5.99. String class — delWord method

```
"Now is the time"-delWord(2, 2) -> "Now time"
"Now is the time"-delWord(3)   -> "Now is "
"Now is the time"-delWord(5)   -> "Now is the time"
"Now is the time"-delWord(3, 1) -> "Now is   time"
```

5.1.7.63. encodeBase64

 Returns a new string that is the base64 encoded version of the receiving string.

Example 5.100. String class — encodeBase64 method

```
"abcdef"-encodeBase64       ->    "YWJjZGVm"
```

5.1.7.64. endsWith

Returns \texttt{true} if the characters of the \texttt{other} match the characters at the end of the target string. Returns \texttt{false} if the characters are not a match, or if \texttt{other} is the null string.

The endsWith method is useful for efficient string parsing as it does not require new string objects be extracted from the target string.

See also methods \texttt{startsWith}, \texttt{caselessEndsWith}, and \texttt{match}.

5.1.7.65. equals
String Class

Returns .true if the target string is strictly equal to the other string. Returns .false if the two strings are not strictly equal. This is the same comparison performed by the "==" comparison method.

Example 5.101. String class — equals method

```
"3"~equals("3")   ->  1
"33"~equals("3")  ->  0
"4"~equals("3")   ->  0
```

5.1.7.66. floor

Returns the largest integer not greater than the receiving string value. The receiving string value is first rounded according to standard Rexx rules, as though the operation receiving_string+0 had been carried out. The floor is then calculated from that result and returned. The result is never in exponential form. If there are no nonzero digits in the result, any minus sign is removed.

Example 5.102. String class — floor method

```
2~floor          ->  2
'-2'~floor       ->  -2
12.3~floor       ->  12
'-12.3'~floor    ->  -13
```

Note

The number is rounded according to the current setting of NUMERIC DIGITS if necessary, before the method processes it.

5.1.7.67. format

Returns a copy of the receiving string, a number, rounded and formatted.

The number is first rounded according to standard Rexx rules, as though the operation receiving_string+0 had been carried out. If you specify no arguments the result of the method is the same as the result of this operation. If you specify any options, the number is formatted as described in the following.

The before and after options describe how many characters are to be used for the integer and decimal parts of the result. If you omit either or both of them, the number of characters for that part is as needed.
If *before* is not large enough to contain the integer part of the number (plus the sign for a negative number), an error results. If *before* is larger than needed for that part, the number is padded on the left with blanks. If *after* is not the same size as the decimal part of the number, the number is rounded (or extended with zeros) to fit. Specifying 0 causes the number to be rounded to an integer.

**Example 5.103. String class — format method**

```
"3"~format(4) -> "3"
"1.73"~format(4,0) -> "2"
"1.73"~format(4,3) -> "1.730"
"-.76"~format(4,1) -> "-0.8"
"3.03"~format(4)  -> "3.03"
" - 12.73"~format(4) -> "-12.7300"
" - 12.73"~format  -> "-12.73"
"0.000"~format     -> "0"
```

`expp` and `expt` control the exponent part of the result, which, by default, is formatted according to the current NUMERIC settings of DIGITS and FORM. `expp` sets the number of places for the exponent part; the default is to use as many as needed (which can be zero). `expt` specifies when the exponential expression is used. The default is the current setting of NUMERIC DIGITS.

If `expp` is 0, the number is not an exponential expression. If `expp` is not large enough to contain the exponent, an error results.

If the number of places needed for the integer or decimal part exceeds `expt` or twice `expt`, respectively, exponential notation is used. If `expt` is 0, exponential notation is always used unless the exponent would be 0. (If `expp` is 0, this overrides a 0 value of `expt`.) If the exponent would be 0 when a nonzero `expp` is specified, then `expp+2` blanks are supplied for the exponent part of the result. If the exponent would be 0 and `expp` is not specified, the number is not an exponential expression.

**Example 5.104. String class — format method**

```
"12345.73"~format(,2,2)   -> "1.234573E+04"
"12345.73"~format(,3,0)   -> "1.235E+4"
"1234573"~format(,3,0)    -> "1.235"
"12345.73"~format(,3,6)   -> "12345.73"
"1234567e5"~format(,3,0)  -> "123456700000.000"
```

### 5.1.7.68. `hashCode`

Returns a string value that is used as a hash value for *MapCollections* such as *Table*, *Relation*, *Set*, *Bag*, or *Directory*. The String `hashCode` method will return the same hash value for all pairs of string instances for which the == operator is `.true.`

See also *Object* method `hashCode` for details.

### 5.1.7.69. `insert`
String Class

 insert(n, new, length, pad)

Returns a copy of the receiver string with the string new, padded or truncated to length length, inserted after the nth character. The default value for n is 0, which means insertion at the beginning of the string. If specified, n and length must be positive whole numbers or zero. If n is greater than the length of the receiving string, the string new is padded at the beginning. The default value for length is the length of new. If length is less than the length of the string new, then insert truncates new to length length. The default pad character is a blank.

Example 5.105. String class — insert method

```
"abc"~insert("123")            ->    "123abc"
"abcdef"~insert(" ", 3)         ->    "abc def"
"abc"~insert("123", 5, 6)       ->    "abc 123 "
"abc"~insert("123", 5, 6, "+")  ->    "abc++123+++"
"abc"~insert("123", , 5, "+")   ->    "123--abc"
```

5.1.7.70. lastPos

lastPos(needle, start, length)

Returns the position of the last occurrence of a string, needle, in the receiving string. It returns 0 if needle is the null string or not found. By default, the search starts at the last character of the receiving string and scans backward to the beginning of the string. You can override this by specifying start, the point at which the backward scan starts and length, the range of characters to scan. The start must be a positive whole number and defaults to receiving_string~length if larger than that value or omitted. The length must be a non-negative whole number and defaults to start.

See also methods pos and caselessPos.

Example 5.106. String class — lastPos method

```
"abc def ghi"~lastPos( " ")       ->    8
"abcdefghi"~lastPos( " ")        ->    0
"efgxyz"~lastPos("xy")           ->    4
"abc def ghi"~lastPos( " ",7)     ->    4
"abc def ghi"~lastPos( " ",7,3)   ->    0
```

5.1.7.71. left

left(length, pad)

Returns a string of length length, containing the leftmost length characters of the receiving string. The string returned is padded with pad characters (or truncated) on the right as needed. The default pad
character is a blank. The length must be a positive whole number or zero. The left method is exactly equivalent to `substr(1, length, pad)`.

---

**Example 5.107. String class — left method**

```
"abc d"~left(8)  ->  "abc d  
"abc d"~left(8,"."  ->  "abc d...
"abc def"~left(7)  ->  "abc  de"
```

---

### 5.1.7.72. length

![length](image)

Returns the length of the receiving string.

---

**Example 5.108. String class — length method**

```
"abcdefgh"~length     ->  8
"abc defg"~length     ->  8
""~length             ->  0
```

---

### 5.1.7.73. lower

![lower](image)

Returns a new string with the characters of the target string beginning with character `n` for `length` characters converted to lowercase. If `n` is specified, it must be a positive whole number. If `n` is not specified, the case conversion will start with the first character. If `length` is specified, it must be a non-negative whole number. If `length` is not specified, the default is to convert the remainder of the string.

---

**Example 5.109. String class — lower method**

```
"Albert Einstein"~lower      ->  "albert einstein"
"ABCDEF"~lower(4)            ->  "ABCdef"
"ABCDEF"~lower(3,2)          ->  "ABcDE"
```

---

### 5.1.7.74. makeArray

![makeArray](image)

This method returns an Array of the receiving string’s strings substrings that were separated by the `separator` string. `separator` may be any string, including the null string. If the null string is used, an
Array containing each character of the string is returned. If the target string starts with the separator, the first Array item will be a null string. If the string ends with a separator, no extra null string item will be added. If separator isn't specified, any line-end indicator is honored.

Example 5.110. String class — makeArray method

```plaintext
string = "hello".endofline"world".endofline"this is an array." 
array = string~makeArray 
say "the second line is:" array[2] /* world */

string = "hello".world"this is an array." 
array = string~makeArray("*") 
say "the third line is:" array[3] /* this is an array. */

string = "hello".world"this is an array." 
array = string~makeArray("*") /* contains 3 items */
```

5.1.7.75. makeString

Returns a string with the same string value as the receiver object. If the receiver is an instance of a subclass of the String class, this method returns an equivalent string object. If the receiver is a string object (not an instance of a subclass of the String class), this method returns the receiver object. See Section 4.2.11, "Required String Values".

5.1.7.76. match

Returns .true if the characters of the other match the characters of the target string beginning at position start. Returns .false if the characters are not a match. start must be a positive whole number.

If n is specified, the match will be performed starting with character n of other. The default value for n is "1". n must be a positive whole number less than or equal to the length of other.

If length is specified, it defines a substring of other that is used for the match. length must be a positive whole number and the combination of n and length must be a valid substring within the bounds of other.

The match method is useful for efficient string parsing as it does not require new string objects be extracted from the target string.

Example 5.111. String class — match method

```
"Saturday"~match(6, "day")           ->    1
"Saturday"~match(6, "DAY")           ->    0
"Saturday"~match(6, "Sunday", 4, 3)  ->    1
"Saturday"~match(6, "daytime", 1, 3) ->    1
```
5.1.7.77. matchChar

Returns `.true.` if the character at position `n` matches any character of the string `chars`. Returns `.false.` if the character does not match any of the characters in the reference set. The argument `n` must be a positive whole number.

Example 5.112. String class — matchChar method

```
"a+b"~matchChar(2, "+-*/")           ->    1
"a+b"~matchChar(1, "+-*/")           ->    0
"Friday"~matchChar(3, "aeiou")       ->    1
"FRIDAY"~matchChar(3, "aeiou")       ->    0
```

5.1.7.78. max

Returns the largest number from among the receiver and any arguments. The number that `max` returns is formatted according to the current NUMERIC settings. You can specify any number of numbers.

Example 5.113. String class — max method

```
12~max(6,7,9)                                                ->    12
17.3~max(19,17.03)                                           ->    19
"-7"~max("-3","-4.3")                                        ->    -3
1~max(2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21)   ->    21
```

5.1.7.79. min

Returns the smallest number from among the receiver and any arguments. The number that `min` returns is formatted according to the current NUMERIC settings. You can specify any number of numbers.

Example 5.114. String class — min method

```
12~min(6,7,9)                                                ->     6
17.3~min(19,17.03)                                           ->    17.03
```
5.1.7.80. modulo

```
"-7"~MIN("-3", "-4.3")        ->    -7
21~min{20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1} ->    1
```

Returns the remainder after dividing the receiving string by \( n \). The receiving string must be a whole number and \( n \) must be a positive whole number. The returned remainder is always in the range 0 through \( n \) minus one.

If both the receiving string and \( n \) are non-negative whole numbers, the result is the same as the result of the remainder (\(/\)\/) operation.

**Example 5.115. String class — modulo method**

```
say 10~modulo(3)    -- 1
say (-10)~modulo(3) -- 2
```

5.1.7.81. overlay

```
overlay( new, n, length, pad )
```

Returns a copy of the receiving string, which, starting at the \( n \)th character, is overlaid with the string \( new \), padded or truncated to length \( length \). The overlay can extend beyond the end of the receiving string. If you specify \( length \), it must be a positive whole number or zero. The default value for \( length \) is the length of \( new \). If \( n \) is greater than the length of the receiving string, padding is added before the \( new \) string. The default \( pad \) character is a blank, and the default value for \( n \) is 1. If you specify \( n \), it must be a positive whole number.

**Example 5.116. String class — overlay method**

```
"abcdef"~overlay(" ",3)         ->    "ab def"
"abcdef"~overlay( ",",3,2)      ->    "ab. ef"
"abcd"~overlay("qq")           ->    "qqcd"
"abcd"~overlay("qq",4)          ->    "abcqq"
"abc"~overlay("123",5,6,"+")   ->    "abc+123+++"
```

5.1.7.82. pos

```
pos( needle, start, length )
```

Returns the position in the receiving string of another string, \( needle \). It returns 0 if \( needle \) is the null string or is not found or if \( start \) is greater than the length of the receiving string. By default, the search starts at the first character of the receiving string (that is, the value of \( start \) is 1), and continues to the end of the string. You can override this by specifying \( start \), the point at which the search starts, and
length, the bounding limit for the search. If specified, start must be a positive whole number and length must be a non-negative whole number.

See also methods lastPos and caselessPos.

Example 5.117. String class — pos method

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Saturday&quot;~pos(&quot;day&quot;)</td>
<td>-&gt; 6</td>
</tr>
<tr>
<td>&quot;abc def ghi&quot;~pos(&quot;x&quot;)</td>
<td>-&gt; 0</td>
</tr>
<tr>
<td>&quot;abc def ghi&quot;~pos(&quot; &quot;)</td>
<td>-&gt; 4</td>
</tr>
<tr>
<td>&quot;abc def ghi&quot;~pos(&quot; &quot;,5)</td>
<td>-&gt; 8</td>
</tr>
<tr>
<td>&quot;abc def ghi&quot;~pos(&quot; &quot;,5,3)</td>
<td>-&gt; 0</td>
</tr>
</tbody>
</table>

5.1.7.83. replaceAt

Returns a copy of the receiving string, with the characters from the n<sup>th</sup> character for length characters replaced with new. The replacement position and length can extend beyond the end of the receiving string. The starting position, n, is required and must be a positive whole number. The length argument is optional and must be a positive whole number or zero. If omitted, length defaults to the length of new.

If n is greater than the length of the receiving string, padding is added before the new string. The default pad character is a blank.

Example 5.118. String class — replaceAt method

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abcdef&quot;~replaceAt(&quot; &quot;,3, 1)</td>
<td>-&gt; &quot;ab def&quot;</td>
</tr>
<tr>
<td>&quot;abcdef&quot;~replaceAt(&quot; &quot;,3, 3)</td>
<td>-&gt; &quot;ab f&quot;</td>
</tr>
<tr>
<td>&quot;abc&quot;~replaceAt(&quot;123&quot;,5,6,&quot;++&quot;)</td>
<td>-&gt; &quot;abc123&quot;</td>
</tr>
</tbody>
</table>

5.1.7.84. reverse

Returns a copy of the receiving string reversed.

Example 5.119. String class — reverse method

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ABc.&quot;~reverse</td>
<td>-&gt; &quot;.cBA&quot;</td>
</tr>
<tr>
<td>&quot;XYZ&quot;~reverse</td>
<td>-&gt; &quot; ZYX&quot;</td>
</tr>
</tbody>
</table>

5.1.7.85. right
Returns a string of length \( \text{length} \) containing the rightmost \( \text{length} \) characters of the receiving string. The string returned is padded with \( \text{pad} \) characters, or truncated, on the left as needed. The default \( \text{pad} \) character is a blank. The \( \text{length} \) must be a positive whole number or zero.

**Example 5.120. String class — right method**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc d&quot;-right(8)</td>
<td>&quot; abc d&quot;</td>
</tr>
<tr>
<td>&quot;abc d e f&quot;-right(5)</td>
<td>&quot; c d e f&quot;</td>
</tr>
<tr>
<td>&quot;12&quot;-right(5,&quot;0&quot;)</td>
<td>&quot;00012&quot;</td>
</tr>
</tbody>
</table>

### 5.1.7.86. round

Returns the nearest integer to the receiving string value. Half is always rounded away from zero. The receiving string is first rounded according to standard Rexx rules, as though the operation \( \text{receiving_string}+0 \) had been carried out. The rounded value is then calculated from that result and returned. The result is never in exponential form. If there are no nonzero digits in the result, any minus sign is removed.

**Example 5.121. String class — round method**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-round</td>
<td>2</td>
</tr>
<tr>
<td>'-2'-round</td>
<td>-2</td>
</tr>
<tr>
<td>2.4-round</td>
<td>2</td>
</tr>
<tr>
<td>'-2.4'-round</td>
<td>-2</td>
</tr>
<tr>
<td>2.5-round</td>
<td>3</td>
</tr>
<tr>
<td>'-2.5'-round</td>
<td>-3</td>
</tr>
<tr>
<td>'-0.1'-round</td>
<td>0</td>
</tr>
</tbody>
</table>

### 5.1.7.87. sign

Returns a number that indicates the sign of the receiving string, which is a number. The receiving string is first rounded according to standard Rexx rules, as though the operation \( \text{receiving_string}+0 \) had been carried out. It returns -1 if the receiving string is less than 0, 0 if it is 0, and 1 if it is greater than 0.

**Example 5.122. String class — sign method**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;12.3&quot;-sign</td>
<td>1</td>
</tr>
<tr>
<td>&quot; -0.307&quot;-sign</td>
<td>-1</td>
</tr>
</tbody>
</table>
5.1.7.88. space

Returns a copy of receiving string, with \( n \) pad characters between each whitespace-delimited word. If you specify \( n \), it must be a positive whole number or zero. If it is 0, all whitespace characters are removed. Leading and trailing whitespace characters are always removed. The default for \( n \) is 1, and the default pad character is a blank.

Example 5.123. String class — space method

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc def&quot;-space</td>
<td>&quot;abc def&quot;</td>
</tr>
<tr>
<td>&quot; abc def&quot;-space(3)</td>
<td>&quot;abc def&quot;</td>
</tr>
<tr>
<td>&quot;abc def&quot;-space(1)</td>
<td>&quot;abc def&quot;</td>
</tr>
<tr>
<td>&quot;abc def&quot;-space(0)</td>
<td>&quot;abcdef&quot;</td>
</tr>
<tr>
<td>&quot;abc def&quot;-space(2,&quot;++&quot;)</td>
<td>&quot;abc++def&quot;</td>
</tr>
</tbody>
</table>

5.1.7.89. startsWith

Returns .true if the characters of the other match the characters at the start of the target string. Returns .false if the characters are not a match, or if other is the null string.

The startsWith method is useful for efficient string parsing as it does not require new string objects be extracted from the target string.

See also methods abbrev, caselessStartsWith, endsWith, and match.

5.1.7.90. strip

Returns a copy of the receiving string with leading characters, trailing characters, or both, removed, based on the option you specify. The following are valid options. (You need to specify only the first capitalized letter; all characters following it are ignored.)

**Both**
Removes both leading and trailing characters. This is the default.

**Leading**
Removes leading characters.

**Trailing**
Removes trailing characters.
The chars specifies the set of characters to be removed, and the default is to remove all whitespace characters (spaces and horizontal tabs). If chars is a null string, then no characters are removed. Otherwise, any occurrences of the characters in chars will be removed.

### Example 5.124. String class — strip method

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot; ab c &quot;--strip</td>
<td>&quot;ab c&quot;</td>
</tr>
<tr>
<td>&quot; ab c &quot;--strip(&quot;L&quot;)</td>
<td>&quot;ab c &quot;</td>
</tr>
<tr>
<td>&quot; ab c &quot;--strip(&quot;t&quot;)</td>
<td>&quot; ab c&quot;</td>
</tr>
<tr>
<td>&quot;12.7000&quot;--strip(0)</td>
<td>&quot;12.7&quot;</td>
</tr>
<tr>
<td>&quot;0012.700&quot;--strip(0)</td>
<td>&quot;12.7&quot;</td>
</tr>
<tr>
<td>&quot;0012.000&quot;--strip(,&quot;.0&quot;)</td>
<td>&quot;12&quot;</td>
</tr>
</tbody>
</table>

#### 5.1.7.91. subChar

subChar($n$)

Returns the $n$'th character of the receiving string. $n$ must be a positive whole number. If $n$ is greater than the length of the receiving string then a zero-length string is returned.

See also methods $[]$ and $substr$.

#### 5.1.7.92. substr

substr($n$, $length$, $pad$)

Returns the substring of the receiving string that begins at the $n$th character and is of length $length$, padded with $pad$ if necessary. The $n$ must be a positive whole number. If $n$ is greater than receiving_string$length$, only pad characters are returned.

If you omit $length$, the rest of the string is returned. The default $pad$ character is a blank.

See also methods $[]$, $subChar$, $left$, and $right$.

In some situations the positional (numeric) patterns of Parsing templates are more convenient for selecting substrings, in particular if you need to extract more than one substring from a string.

### Example 5.125. String class — substr method

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot;--substr(2)</td>
<td>&quot;bc&quot;</td>
</tr>
<tr>
<td>&quot;abc&quot;--substr(2,4)</td>
<td>&quot;bc &quot;</td>
</tr>
<tr>
<td>&quot;abc&quot;--substr(2,6,&quot;&quot;)</td>
<td>&quot;bc....&quot;</td>
</tr>
</tbody>
</table>

#### 5.1.7.93. subWord

subWord($n$, $length$)


Returns the substring of the receiving string that starts at the \textit{n}th word and is up to \textit{length} whitespace-delimited words. The \textit{n} must be a positive whole number. If you omit \textit{length}, it defaults to the number of remaining words in the receiving string. The returned string never has leading or trailing whitespace, but includes all whitespace characters between the selected words.

**Example 5.126. String class — subWord method**

\begin{verbatim}
"Now is the time"-subWord(2,2)    ->    "is the"
"Now is the time"-subWord(3)      ->    "the time"
"Now is the time"-subWord(5)      ->    ""
\end{verbatim}

5.1.7.94. \textbf{subWords}

\textbf{subWords(\textit{n}, \textit{length})}

Returns an array containing all words within the substring of the receiving string that starts at the \textit{n}th word and is up to \textit{length} whitespace-delimited words. The \textit{n} must be a positive whole number. If you omit \textit{n}, it defaults to 1. If you omit \textit{length}, it defaults to the number of remaining words in the receiving string. The strings in the returned array never have leading or trailing whitespace.

**Example 5.127. String class — subWords method**

\begin{verbatim}
"Now is the time"-subWords         ->    .array~of("Now", "is", "the", "time")
"Now is the time"-subWords(2,2)    ->    .array~of("is", "the")
"Now is the time"-subWords(3)      ->    .array~of("the", "time")
"Now is the time"-subWords(5)      ->    .array~new(0)
\end{verbatim}

The subWords method is useful for iterating over the individual words in a string.

**Example 5.128. String class — subWords method**

\begin{verbatim}
do word over source~subWords    -- extract all of the words to loop over
say word
end
\end{verbatim}

5.1.7.95. \textbf{translate}

\textbf{translate(\textit{tableo}, \textit{tablei}, \textit{pad}, \textit{n}, \textit{length})}

208
String Class

Returns a copy of the receiving string with each character translated to another character or unchanged. You can also use this method to reorder the characters in the output table. (See last example)

The output table is `tableo` and the input translation table is `tablei`. `translate` searches `tablei` for each character in the receiving string. If the character is found, the corresponding character in `tableo` is used in the result string. If there are duplicates in `tablei`, the first (leftmost) occurrence is used. If the character is not found, the original character in the receiving string is used. The result string is always of the same length as the receiving string.

The tables can be of any length. If you specify neither translation table and omit `pad`, the receiving string is translated to uppercase (that is, lowercase `a`-`z` to uppercase `A`-`Z`), but if you include `pad` the entire string is translated to `pad` characters. `tablei` defaults to `XRANGE("00"x, "FF"x)`, and `tableo` defaults to the null string and is padded with `pad` or truncated as necessary. The default `pad` is a blank.

`n` is the position of the first character of the translated range. The default starting position is 1. `length` is the range of characters to be translated. If `length` is omitted, the remainder of the string from the starting position to the end is used.

### Example 5.129. String class — translate method

```
"abcdef"-translate                        ->    "ABCDEF"
"abcdef"-translate(, , , 3, 2)             ->    "abCDef"
"abcdef"-translate("12", "ec")             ->    "ab2d1f"
"abcdef"-translate("12", "abcd", ":")     ->    "12..ef"
"APQRV"-translate(, "PR")                 ->    "A Q V"
"APQRV"-translate(XRANGE("00"X, "Q"))     ->    "APQ  
"4123"-translate("abcd", "1234", , 2, 2)   ->    "4ab3"
"4123"-translate("abcd", "1234")          ->    "dabc"
```

#### Note

The last example shows how to use the `translate` method to reorder the characters in a string. In the example, the last character of any 4-character string specified as the first argument would be moved to the beginning of the string.

### 5.1.7.96. trunc

Returns the integer part the receiving string, which is a number, and `n` decimal places. The default `n` is 0 and returns an integer with no decimal point. If you specify `n`, it must be a positive whole number or zero. The receiving string is first rounded according to standard Rexx rules, as though the operation `receiving_string+0` had been carried out. This number is then truncated to `n` decimal places or trailing zeros are added if needed to reach the specified length. The result is never in exponential form. If there are no nonzero digits in the result, any minus sign is removed.
Example 5.130. String class — trunc method

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.3-trunc</td>
<td>12</td>
</tr>
<tr>
<td>127.09782-trunc(3)</td>
<td>127.097</td>
</tr>
<tr>
<td>127.1-trunc(3)</td>
<td>127.100</td>
</tr>
<tr>
<td>127-trunc(2)</td>
<td>127.00</td>
</tr>
</tbody>
</table>

Note

The number is rounded according to the current setting of NUMERIC DIGITS if necessary, before the method processes it.

5.1.7.97. upper

```plaintext
upper(n, length)
```

Returns a new string with the characters of the target string beginning with character \( n \) for \( length \) characters converted to uppercase. If \( n \) is specified, it must be a positive whole number. If \( n \) is not specified, the case conversion will start with the first character. If \( length \) is specified, it must be a non-negative whole number. If \( length \) is not specified, the default is to convert the remainder of the string.

Example 5.131. String class — upper method

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Albert Einstein&quot;-upper</td>
<td>&quot;ALBERT EINSTEIN&quot;</td>
</tr>
<tr>
<td>&quot;abcdef&quot;-upper(4)</td>
<td>&quot;abcDEF&quot;</td>
</tr>
<tr>
<td>&quot;abcdef&quot;-upper(3,2)</td>
<td>&quot;abCDef&quot;</td>
</tr>
</tbody>
</table>

5.1.7.98. verify

```plaintext
verify(reference, start, length, option)
```

Returns a number that, by default, indicates whether the receiving string is composed only of characters from \( reference \). It returns 0 if all characters in the receiving string are in \( reference \) or returns the position of the first character in the receiving string not in \( reference \).

The \( option \) can be either \texttt{Nomatch} (the default) or \texttt{Match}. (You need to specify only the first capitalized and highlighted letter; all characters following the first character are ignored)

If you specify \texttt{Match}, the method returns the position of the first character in the receiving string that is in \( reference \), or returns 0 if none of the characters are found.
The default for `start` is 1. Thus, the search starts at the first character of the receiving string. You can override this by specifying a different `start` point, which must be a positive whole number.

The default for `length` is the length of the string from `start` to the end of the string. Thus, the search proceeds to the end of the receiving string. You can override this by specifying a different `length`, which must be a non-negative whole number.

If the receiving string is null, the method returns 0, regardless of the value of the `option`. Similarly, if `start` is greater than `receiving_string-length`, the method returns 0. If `reference` is null, the method returns 0 if you specify Match. Otherwise, the method returns the `start` value.

### Example 5.132. String class — verify method

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;123&quot;-verify(&quot;1234567890&quot;)</td>
<td>0</td>
</tr>
<tr>
<td>&quot;1Z3&quot;-verify(&quot;1234567890&quot;)</td>
<td>2</td>
</tr>
<tr>
<td>&quot;AB4T&quot;-verify(&quot;1234567890&quot;)</td>
<td>1</td>
</tr>
<tr>
<td>&quot;AB4T&quot;-verify(&quot;123456789&quot;,&quot;M&quot;)</td>
<td>3</td>
</tr>
<tr>
<td>&quot;AB4T&quot;-verify(&quot;123456789&quot;,&quot;N&quot;)</td>
<td>1</td>
</tr>
<tr>
<td>&quot;1P3Q4&quot;-verify(&quot;1234567890&quot;, ,3)</td>
<td>4</td>
</tr>
<tr>
<td>&quot;123&quot;-verify(&quot;&quot;, ,2)</td>
<td>2</td>
</tr>
<tr>
<td>&quot;ABCDE&quot;-verify(&quot;&quot;, ,3)</td>
<td>3</td>
</tr>
<tr>
<td>&quot;AB3CD5&quot;-verify(&quot;1234567890&quot;,&quot;,&quot;M&quot;,4)</td>
<td>6</td>
</tr>
<tr>
<td>&quot;ABCDEF&quot;-verify(&quot;ABC&quot;, &quot;N&quot;,2,3)</td>
<td>4</td>
</tr>
<tr>
<td>&quot;ABCDEF&quot;-verify(&quot;ADEF&quot;, &quot;M&quot;,2,3)</td>
<td>4</td>
</tr>
</tbody>
</table>

#### 5.1.7.99. word

**.word**\(_n\)  

Returns the \(_n\)th whitespace-delimited word in the receiving string or the null string if the receiving string has fewer than \(_n\) words. The \(_n\) must be a positive whole number. This method is exactly equivalent to \(\text{subWord}(n, 1)\).

### Example 5.133. String class — word method

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Now is the time&quot;-word(3)</td>
<td>&quot;the&quot;</td>
</tr>
<tr>
<td>&quot;Now is the time&quot;-word(5)</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

#### 5.1.7.100. wordIndex

**wordIndex**\(_n\)  

Returns the position of the first character in the \(_n\)th whitespace-delimited word in the receiving string. It returns 0 if the receiving string has fewer than \(_n\) words. The \(_n\) must be a positive whole number.

### Example 5.134. String class — wordIndex method

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Now is the time&quot;-wordIndex(3)</td>
<td>8</td>
</tr>
</tbody>
</table>
5.1.7.101. wordLength

Returns the length of the \textit{n}th whitespace-delimited word in the receiving string or 0 if the receiving string has fewer than \textit{n} words. The \textit{n} must be a positive whole number.

Example 5.135. String class — wordLength method

\begin{verbatim}
"Now is the time"-wordLength(2)       ->    2
"Now comes the time"-wordLength(2)    ->    5
"Now is the time"-wordLength(6)       ->    0
\end{verbatim}

5.1.7.102. wordPos

Returns the word number of the first word of \textit{phrase} found in the receiving string, or 0 if \textit{phrase} contains no words or if \textit{phrase} is not found. Several whitespace characters between words in either \textit{phrase} or the receiving string are treated as a single blank for the comparison, but, otherwise, the words must match exactly.

By default the search starts at the first word in the receiving string. You can override this by specifying \textit{start} (which must be positive), the word at which the search is to be started.

Example 5.136. String class — wordPos method

\begin{verbatim}
"now is the time"-wordPos("the")              ->  3
"now is the time"-wordPos("The")              ->  0
"now is the time"-wordPos("is the")           ->  2
"now is the time"-wordPos("is the")           ->  2
"now is the time"-wordPos("is time ")         ->  0
"To be or not to be"-wordPos("be")            ->  2
"To be or not to be"-wordPos("be",3)          ->  6
\end{verbatim}

5.1.7.103. words

Returns the number of whitespace-delimited words in the receiving string.
Example 5.137. String class — words method

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Now is the time&quot;-words</td>
<td>-&gt; 4</td>
<td></td>
</tr>
<tr>
<td>&quot; &quot;-words</td>
<td>-&gt; 0</td>
<td></td>
</tr>
</tbody>
</table>

5.1.7.104. x2b

Returns a string, in character format, that represents the receiving string, which is a string of hexadecimal characters converted to binary. The receiving string can be of any length. Each hexadecimal character is converted to a string of 4 binary digits. The receiving string can optionally include whitespace characters (at byte boundaries only, not leading or trailing) to improve readability; they are ignored.

The returned string has a length that is a multiple of four, and does not include any whitespace.

If the receiving string is null, the method returns a null string.

Example 5.138. String class — x2b method

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;C3&quot;-x2b</td>
<td>-&gt; &quot;11000011&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;7&quot;-x2b</td>
<td>-&gt; &quot;0111&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;1 C1&quot;-x2b</td>
<td>-&gt; &quot;000111000001&quot;</td>
<td></td>
</tr>
</tbody>
</table>

You can combine x2b with the methods d2x and c2x to convert numbers or character strings into binary form.

Example 5.139. String class — x2b method with c2x

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;C3&quot;-c2x-x2b</td>
<td>-&gt; &quot;11000011&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;129&quot;-d2x-x2b</td>
<td>-&gt; &quot;10000001&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;12&quot;-d2x-x2b</td>
<td>-&gt; &quot;1100&quot;</td>
<td></td>
</tr>
</tbody>
</table>

5.1.7.105. x2c

Returns a string, in character format, that represents the receiving string, which is a hexadecimal string converted to character. The returned string is half as many bytes as the receiving string. The receiving string can be any length. If necessary, it is padded with a leading 0 to make an even number of hexadecimal digits.

You can optionally include whitespace in the receiving string (at byte boundaries only, not leading or trailing) to improve readability; they are ignored.

If the receiving string is null, the method returns a null string.
Example 5.140. String class — x2c method

```
"4865 6c6c 6f"~x2c -> "Hello" /* ASCII */
"3732 73"~x2c -> "72s" /* ASCII */
```

5.1.7.106. x2d

Returns the decimal representation of the receiving string, which is a string of hexadecimal characters.
If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS.

You can optionally include whitespace characters in the receiving string (at byte boundaries only, not leading or trailing) to improve readability; they are ignored.

If the receiving string is null, the method returns 0.

If you do not specify n, the receiving string is processed as an unsigned binary number.

Example 5.141. String class — x2d method

```
"0E"~x2d        ->    14
"81"~x2d        ->    129
"F81"~x2d        ->    3969
"FF81"~x2d      ->    65409
"46 30"X~x2d    ->    240          /*  ASCII   */
"66 30"X~x2d    ->    240          /*  ASCII   */
```

If you specify n, the receiving string is taken as a signed number expressed in n hexadecimal digits. If the leftmost bit is off, then the number is positive; otherwise, it is a negative number. In both cases it is converted to a whole number, which can be negative. If n is 0, the method returns 0.

If necessary, the receiving string is padded on the left with 0 characters (note, not "sign-extended"), or truncated on the left to n characters.

Example 5.142. String class — x2d method

```
"81"~x2d(2)      ->    -127
"81"~x2d(4)      ->    129
"F081"~x2d(4)    ->    -3967
"F081"~x2d(3)    ->    129
"F081"~x2d(2)    ->    -127
"F081"~x2d(1)    ->    1
"0031"~x2d(0)    ->    0
```

5.2. Stream Classes

This section describes the Rexx classes which implement Rexx data streams: `InputStream`, `OutputStream`, `InputOutputStream`, and `Stream` class.
5.2.1. InputOutputStream Class

This class is defined as an abstract mixin class. It must be implemented by subclassing it or inheriting from it as a mixin. Many of the methods in this class are abstract and must be overridden or they will throw a syntax error when invoked.

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Object class</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class (MetaClass)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>+ OutputStream (Mixin Class)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>OutputStream class</strong></td>
</tr>
<tr>
<td>arrayOut</td>
</tr>
<tr>
<td>charIn</td>
</tr>
<tr>
<td>charOut (Abstract Method)</td>
</tr>
<tr>
<td>chars</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>+ InputStream (Mixin Class)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>InputStream class</strong></td>
</tr>
<tr>
<td>arrayIn</td>
</tr>
<tr>
<td>charIn (Abstract Method)</td>
</tr>
<tr>
<td>charOut</td>
</tr>
<tr>
<td>chars (Abstract Method)</td>
</tr>
</tbody>
</table>

**InputOutputStream (Mixin Class)**
(no class or instance methods)

5.2.2. InputStream Class

This class is defined as an abstract mixin class. It must be implemented by subclassing it or inheriting from it as a mixin. Many of the methods in this class are abstract and must be overridden or they will throw a syntax error when invoked.

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Object class</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class (MetaClass)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>InputStream (Mixin Class)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrayIn</td>
</tr>
<tr>
<td>charIn (Abstract Method)</td>
</tr>
<tr>
<td>charOut</td>
</tr>
<tr>
<td>chars (Abstract Method)</td>
</tr>
</tbody>
</table>

5.2.2.1. arrayIn

This method is a default **arrayIn** implementation using **lineIn** to fill the array.
5.2.2.2. charIn (Abstract Method)

This method is defined as an abstract method. Invoking it will cause syntax error 93.965 to be raised.

5.2.2.3. charOut

This is an unsupported operation for InputStreams. Invoking it will cause syntax error 93.963 to be raised.

5.2.2.4. chars (Abstract Method)

This method is defined as an abstract method. Invoking it will cause syntax error 93.965 to be raised.

5.2.2.5. close

This method is a NOP by default.

5.2.2.6. lineIn (Abstract Method)

This method is defined as an abstract method. Invoking it will cause syntax error 93.965 to be raised.

5.2.2.7. lineOut

This is an unsupported operation for InputStreams. Invoking it will cause syntax error 93.963 to be raised.

5.2.2.8. lines (Abstract Method)

This method is defined as an abstract method. Invoking it will cause syntax error 93.965 to be raised.

5.2.2.9. open

This method is a NOP method.

5.2.2.10. position

This method is an optionally supported operation. By default, it will cause syntax error 93.963 to be raised.

5.2.3. OutputStream Class

This class is defined as an abstract mixin class. It must be implemented by subclassing it or inheriting from it as a mixin. Many of the methods in this class are abstract and must be overridden or they will throw a syntax error when invoked.
Table 5.10. OutputStream Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Object Class</th>
<th>OutputStream (Mixin Class)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Object class</td>
<td></td>
<td>close</td>
</tr>
<tr>
<td>Class (Metaclass)</td>
<td></td>
<td>lineIn</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
<td></td>
<td>lineOut (Abstract Method)</td>
</tr>
<tr>
<td>arrayOut</td>
<td>charIn</td>
<td>charOut (Abstract Method)</td>
</tr>
<tr>
<td>charOut (Abstract Method)</td>
<td>chars</td>
<td>lines</td>
</tr>
</tbody>
</table>

5.2.3.1. arrayOut

This method is a default arrayOut implementation that writes all lines to the stream using lineOut.

5.2.3.2. charIn

This is an unsupported operation for OutputStreams. Invoking it will cause syntax error 93.963 to be raised.

5.2.3.3. charOut (Abstract Method)

This method is defined as an abstract method. Invoking it will cause syntax error 93.965 to be raised.

5.2.3.4. chars

This is an unsupported operation for OutputStreams. Invoking it will cause syntax error 93.963 to be raised.

5.2.3.5. close

This method is a NOP by default.

5.2.3.6. lineIn

This is an unsupported operation for OutputStreams. Invoking it will cause syntax error 93.963 to be raised.

5.2.3.7. lineOut (Abstract Method)

This method is defined as an abstract method. Invoking it will cause syntax error 93.965 to be raised.

5.2.3.8. lines
This is an unsupported operation for OutputStreams. Invoking it will cause syntax error 93.963 to be raised.

### 5.2.3.9. open

This method is a NOP by default.

### 5.2.3.10. position

This method is an optionally supported operation. By default, it will cause syntax error 93.963 to be raised.

### 5.2.4. Stream Class

A stream object allows external communication from Rexx. (See *Chapter 14, Input and Output Streams* for a discussion of Rexx input and output.)

The Stream class is a subclass of the *InputOutputStream class*.

#### Table 5.11. Stream Class

<table>
<thead>
<tr>
<th>Method</th>
<th>OutputStream class</th>
<th>InputStream class</th>
<th>InputOutputStream class</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td></td>
<td>open</td>
<td></td>
</tr>
<tr>
<td>lineIn</td>
<td></td>
<td>position</td>
<td></td>
</tr>
<tr>
<td>lineOut (Abstract Method)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lines</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arrayOut</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>charIn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>charOut (Abstract Method)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chars</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chars (Abstract Method)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chars (Abstract Method)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lines</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flush</td>
<td></td>
<td>qualify</td>
<td></td>
</tr>
<tr>
<td>init</td>
<td></td>
<td>query</td>
<td></td>
</tr>
<tr>
<td>lineIn</td>
<td></td>
<td>say</td>
<td></td>
</tr>
<tr>
<td>lineOut</td>
<td></td>
<td>seek</td>
<td></td>
</tr>
<tr>
<td>lines</td>
<td></td>
<td>state</td>
<td></td>
</tr>
</tbody>
</table>
5.2.4.1. new (Inherited Class Method)

new(name)

Initializes a stream object for stream name, but does not open the stream. Returns the new stream object.

name can either be a string or a File object.

5.2.4.2. arrayIn

arrayIn()

"Lines"
"Chars"

Returns an Array that contains the data of the receiving Stream, starting from the current read position. If LINES is specified, the Array items returned are the Stream's lines, that were separated with any line-end indicator. LINES is the default. If CHARs is specified, the Array items returned are the Stream's characters.

If you have used the charIn method, the first line can be a partial line.

5.2.4.3. arrayOut

arrayOut(array)

"Lines"
"Chars"

Writes the data in array array to the stream. If LINES is specified, each element of the array is written using lineOut. If CHARs is specified, each element is written using charOut. The default method is LINES.

5.2.4.4. charIn

charIn(start, length)

Returns a string of up to length characters from the input stream. The stream advances the read pointer by the number of characters read. If you omit length, it defaults to 1. If you specify start, this positions the read pointer before reading. The start value must be a positive whole number within the bounds of the stream. If the value is not a positive whole number, a syntax condition is raised. When the value is past the end of the stream, the empty string is returned and the NOTREADY condition is raised. If the stream is not already open, the stream attempts to open for reading and writing. If that fails, the stream opens for input only.
5.2.4.5. **charOut**

```
charOut(string, start)
```

Returns the count of characters remaining after trying to write `string` to the output stream. The stream also advances the write pointer.

The `string` can be the null string. In this case, charOut writes no characters to the stream and returns 0. If you omit `string`, charOut writes no characters to the stream and returns 0. The stream is also closed.

If you specify `start`, this positions the write pointer before writing. If the stream is not already open, the stream attempts to open for reading and writing. If that fails, the stream opens for for output only.

5.2.4.6. **chars**

```
chars
```

Returns the total number of characters remaining in the input stream. The count includes any line separator characters, if these are defined for the stream. For persistent streams the count is the count of characters from the current read position. (See Chapter 14, Input and Output Streams for a discussion of Rexx input and output.) The total number of characters remaining cannot be determined for some streams (for example, STDIN or Windows/Unix-like system devices). For these streams, the `chars` method returns 1 to indicate that data is present, or 0 if no data is present.

5.2.4.7. **close**

```
close
```

Closes the stream. close returns READY: if closing the stream is successful, or an appropriate error message. If you have tried to close an unopened file, then the close method returns a null string ("").

5.2.4.8. **command**

```
command(stream_command)
```

Returns a string after performing the specified `stream_command`. The returned string depends on the `stream_command` performed and can be the null string. Commands are available to:

- Open a stream for reading, writing, or both
- Close a stream at the end of an operation
- Move the line read or write position within a persistent stream (for example, a file)
- Get information about a stream

If the method is unsuccessful, it returns an error message string in the same form that the description method uses.

For most error conditions, the additional information is in the form of a numeric return code. This return code is the value of ERRNO that is set whenever one of the file system primitives returns with a -1.
5.2.4.8.1. Command Strings

The argument \textit{stream\_command} can be any expression that evaluates to one of the following command strings:

\begin{itemize}
\item \texttt{Open}
\item \texttt{Close}
\item \texttt{Flush}
\item \texttt{Seek}
\item \texttt{Position}
\item \texttt{Read}
\item \texttt{Write}
\item \texttt{Char}
\item \texttt{Line}
\item \texttt{Query}
\item \texttt{Datetime}
\item \texttt{Exists}
\item \texttt{Handle}
\item \texttt{Sys}
\item \texttt{Size}
\item \texttt{Streamtype}
\item \texttt{Timestamp}
\end{itemize}

\textbf{OPTIONS:}

\begin{itemize}
\item \texttt{SHARED}
\item \texttt{NOBuffer}
\item \texttt{SHARERead}
\item \texttt{SHAREWrite}
\item \texttt{SHAREWrite}
\item \texttt{Binary}
\item \texttt{REClength}
\end{itemize}

\textbf{OPEN}

Opens the stream object and returns \texttt{READY}. (If unsuccessful, the previous information about return codes applies.) The default for OPEN is to open the stream for both reading and writing.
data, for example: 'OPEN BOTH'. To specify that the stream be only opened for input or output, add READ or WRITE, to the command string.

The following is a description of the options for OPEN:

READ
Opens the stream only for reading.

WRITE
Opens the stream only for writing.

BOTH
Opens the stream for both reading and writing. (This is the default.) The stream maintains separate read and write pointers.

APPEND
Positions the write pointer at the end of the stream. (This is the default.) The write pointer cannot be moved anywhere within the extent of the file as it existed when the file was opened.

REPLACE
Sets the write pointer to the beginning of the stream and truncates the file. In other words, this option deletes all data that was in the stream when opened.

SHARED
Enables another process to work with the stream in a shared mode. This mode must be compatible with the shared mode (SHARED, SHAREREAD, or SHAREWRITE) used by the process that opened the stream.

SHAREREAD
Enables another process to read the stream in a shared mode.

SHAREWRITE
Enables another process to write the stream in a shared mode.

NOBUFFER
Turns off buffering of the stream. All data written to the stream is flushed immediately to the operating system for writing. This option can have a severe impact on output performance. Use it only when data integrity is a concern, or to force interleaved output to a stream to appear in the exact order in which it was written.

BINARY
Opens the stream in binary mode. This means that line-end characters are ignored; they are treated like any other byte of data. This is intended to process binary data using the line operations.

Note

Specifying the BINARY option for a stream that does not exist but is opened for writing also requires the RECLENGTH option to be specified. Omitting the RECLENGTH option in this case raises an error condition.
RECLENGTH \textit{length}

Allows the specification of an exact length for each line in a stream. This allows line operations on binary-mode streams to operate on individual fixed-length records. Without this option, line operations on binary-mode files operate on the entire file (for example, as if you specified the \texttt{RECLENGTH} option with a length equal to that of the file). The \textit{length} must be 1 or greater.

\begin{verbatim}
Example 5.143. Stream command — OPEN option

\begin{verbatim}
stream-command("open")
stream-command("open write")
stream-command("open read")
stream-command("open read shared")
\end{verbatim}
\end{verbatim}

CLOSE

closes the stream object. The command method with the CLOSE option returns \texttt{READY}: if the stream is successfully closed or an appropriate error message otherwise. If an attempt to close an unopened file occurs, then the command method with the CLOSE option returns a null string ("\)

FLUSH

forces any data currently buffered for writing to be written to this stream.

SEEK \textit{offset}

sets the read or write position to a given number (\textit{offset}) within a persistent stream. If the stream is open for both reading and writing and you do not specify READ or WRITE, both the read and write positions are set.

\begin{itemize}
  \item \texttt{=} explicitly specifies the \textit{offset} from the beginning of the stream. This is the default if you supply no prefix. For example, an \textit{offset} of 1 with the LINE option means the beginning of the stream.
  \item \texttt{<} specifies \textit{offset} from the end of the stream.
  \item \texttt{+} specifies \textit{offset} forward from the current read or write position.
  \item \texttt{-} specifies \textit{offset} backward from the current read or write position.
\end{itemize}

Note

See Chapter 14, Input and Output Streams for a discussion of read and write positions in a persistent stream.

To use this command, you must first open the stream (with the OPEN stream command described previously or implicitly with an input or output operation). One of the following characters can precede the \textit{offset} number.
The command method with the SEEK option returns the new position in the stream if the read or write position is successfully located, or an appropriate error message.

The following is a description of the options for SEEK:

**READ**
- specifies that this command sets the read position.

**WRITE**
- specifies that this command sets the write position.

**CHAR**
- specifies the positioning in terms of characters. This is the default.

**LINE**
- specifies the positioning in terms of lines. For non-binary streams, this is potentially an operation that can take a long time to complete because, in most cases, the file must be scanned from the top to count the line-end characters. However, for binary streams with a specified record length, the new resulting line number is simply multiplied by the record length before character positioning. See Section 14.1.5, "Line versus Character Positioning" for a detailed discussion of this issue.

**Note**
If you do line positioning in a file open only for writing, you receive an error message.

**Example 5.144. Stream command — SEEK option**

```plaintext
stream~command("seek =2 read")
stream~command("seek +15 read")
stream~command("seek -7 write line")
fromend  = 125
stream~command("seek <"fromend "read")
```

**POSITION**
- is a synonym for SEEK.

**QUERY**
- Used with these QUERY stream commands, the command method returns specific information about a stream. Except for QUERY HANDLE and QUERY SEEK/POSITION, the stream returns the query information even if the stream is not open. The stream returns the null string for nonexistent streams.

**QUERY DATETIME**
- Returns the date and time stamps of a stream in US format. For example:

**Example 5.145. Stream command — QUERY DATETIME option**

```plaintext
stream~command("query datetime")
```
Stream Class

A sample output might be:

```
11-12-15 03:29:12
```

**QUERY EXISTS**
Returns the full path specification of the stream object, if it exists, or a null string. For example:

```
Example 5.146. Stream command — QUERY EXISTS option

stream-command("query exists")
```

A sample output might be:

```
c:\data\file.txt
```

**QUERY HANDLE**
Returns the handle associated with the open stream.

```
Example 5.147. Stream command — QUERY HANDLE option

stream-command("query handle")
```

A sample output might be: 3

**QUERY POSITION**
Returns the current read or write position for the stream, as qualified by the following options:

- **READ**
  Returns the current read position.

- **WRITE**
  Returns the current write position.

- **CHAR**
  Returns the position in terms of characters. This is the default.

- **LINE**
  Returns the position in terms of lines. For non-binary streams, this operation can take a long time to complete. This is because the language processor starts tracking the current line number if not already doing so, and, thus, might require a scan of the stream from the top to
count the line-end characters. See Section 14.1.5, “Line versus Character Positioning” for a detailed discussion of this issue.

Example 5.148. Stream command — QUERY POSITION WRITE option

```
stream-command("query position write")
```

A sample output might be:

```
247
```

SYS

Returns the operating system stream position in terms of characters.

QUERY SEEK

Is a synonym for QUERY POSITION.

QUERY SIZE

Returns the size, in bytes, of a persistent stream.

Example 5.149. Stream command — QUERY SIZE option

```
stream-command("query size")
```

A sample output might be:

```
1385
```

QUERY STREAMTYPE

Returns a string indicating whether the stream is PERSISTENT, TRANSIENT, or UNKNOWN.

QUERY TIMESTAMP

Returns the date and time stamps of a persistent stream in an international format. This is the preferred method of getting date and time because it provides the full 4-digit year

Example 5.150. Stream command — QUERY TIMESTAMP option

```
stream-command("query timestamp")
```

A sample output might be:

```
2015-11-12 03:29:12
```

5.2.4.9. description
Returns a descriptive string associated with the current state of the stream. The description method is identical to the state method except that the string that description returns is followed by a colon and, if available, additional information about ERROR or NOTREADY states.

5.2.4.10. flush

Returns READY:. It forces the stream to write any buffered data to the output stream.

5.2.4.11. init

Initializes a stream object defined by name.

name can either be a string or a File object.

5.2.4.12. lineIn

Returns the next count lines. The count must be 0 or 1. The stream advances the read pointer. If you omit count, it defaults to 1. A line number may be given to set the read position to the start of a specified line. This line number must be positive and within the bounds of the stream, and must not be specified for a transient stream. A value of 1 for line refers to the first line in the stream. If the stream is not already open, then the interpreter tries to open the stream for reading and writing. If that fails, the stream is opened for input only.

5.2.4.13. lineOut

Returns 0 if successful in writing string to the output stream, or 1 if an error occurs while writing the line. The stream advances the write pointer. If you specify line, this positions the write pointer before writing. If you omit both string and line, the stream is closed. If the stream is not already open, the stream attempts to open for reading and writing. If that fails, the stream is opened for output only.

5.2.4.14. lines

Returns the number of lines that are available for input if no option or option Count is specified. If the stream has already been read with charIn this can include an initial partial line. If no data remains,
lines returns 0. For persistent streams the count starts at the current read position. As such, lines reports whether a read action of `charIn` or `lineIn` will succeed. Option `Count` is the default.

If option `Normal` is specified, lines returns `.true` if at least one line remains in the stream, or `.false` if no lines remain.

Note

`lines("Count")` determines the actual number of lines by scanning the stream starting at the current position and counting the lines. For large streams, this can be a time-consuming operation. Therefore, avoid the use of `lines()` or `lines("Count")` in the condition of a loop reading a stream. It is recommended that you use `lines("Normal")` or the `chars` method instead.

For an explanation of input and output, see Chapter 14, Input and Output Streams.

For a `Queue` instance, the inherited method `lines` returns the actual number of lines in the queue.

5.2.4.15. makeArray

```
makeArray( "Line" )
```

Returns an Array that contains the data of the stream in line or character format, starting from the current read position. The line format is the default.

If you have used the `charIn` method, the first line can be a partial line.

5.2.4.16. open

```
open( REAd OPTIONS - fragment )
```

OPTIONS:

```
SHARED
SHARERead
SHAREWrite
```

Opens the stream and returns `READY`. If the method is unsuccessful, it returns an error message string in the same form that the `description` method uses.
For most error conditions, the additional information is in the form of a numeric return code. This return code is the value of \texttt{ERRNO}, which is set whenever one of the file system primitives returns with a -1.

By default, \texttt{open} opens the stream for both reading and writing data, for example: \texttt{'open BOTH'}. To specify that the stream be only opened for input or output, specify \texttt{READ} or \texttt{WRITE}.

The options for the \texttt{open} method are:

\textbf{READ}
\begin{itemize}
  \item Opens the stream for input only.
\end{itemize}

\textbf{WRITE}
\begin{itemize}
  \item Opens the stream for output only.
\end{itemize}

\textbf{BOTH}
\begin{itemize}
  \item Opens the stream for both input and output. (This is the default.) The stream maintains separate read and write pointers.
\end{itemize}

\textbf{APPEND}
\begin{itemize}
  \item Positions the write pointer at the end of the stream. (This is the default.) The write pointer cannot be moved anywhere within the extent of the file as it existed when the file was opened.
\end{itemize}

\textbf{REPLACE}
\begin{itemize}
  \item Sets the write pointer to the beginning of the stream and truncates the file. In other words, this option deletes all data that was in the stream when opened.
\end{itemize}

\textbf{SHARED}
\begin{itemize}
  \item Enables another process to work with the stream in a shared mode. This mode must be compatible with the shared mode (\texttt{SHARED}, \texttt{SHAREREAD}, or \texttt{SHAREWRITE}) used by the process that opened the stream.
\end{itemize}

\textbf{SHAREREAD}
\begin{itemize}
  \item Enables another process to read the stream in a shared mode.
\end{itemize}

\textbf{SHAREWRITE}
\begin{itemize}
  \item Enables another process to write the stream in a shared mode.
\end{itemize}

\textbf{NOBUFFER}
\begin{itemize}
  \item Turns off buffering of the stream. All data written to the stream is flushed immediately to the operating system for writing. This option can have a severe impact on output performance. Use it only when data integrity is a concern, or to force interleaved output to a stream to appear in the exact order in which it was written.
\end{itemize}

\textbf{BINARY}
\begin{itemize}
  \item Opens the stream in binary mode. This means that line-end characters are ignored; they are treated like any other byte of data. This is for processing binary record data using the line operations.
\end{itemize}
Note

Specifying the BINARY option for a stream that does not exist but is opened for writing also requires the RECLENGTH option to be specified. Omitting the RECLENGTH option in this case raises an error condition.

RECLENGTH length

Allows the specification of an exact length for each line in a stream. This allows line operations on binary-mode streams to operate on individual fixed-length records. Without this option, line operations on binary-mode files operate on the entire file (for example, as if you specified the RECLENGTH option with a length equal to that of the file). The length must be 1 or greater.

Example 5.151. Stream object — OPEN method

```plaintext
stream-open
stream-open("write")
stream-open("read")
```

5.2.4.17. position

position is a synonym for seek.

5.2.4.18. qualify

Returns the stream’s fully qualified name. The stream need not be open.

5.2.4.19. query
Used with these options, query returns specific information about a stream. Except for **HANDLE** and **SEEK/POSITION**, the stream returns the query information even if the stream is not open. A null string is returned for nonexistent streams.

**DATETIME**
returns the date and time stamps of a persistent stream in US format.

Example 5.152. Stream object — QUERY method

```
stream-query("datetime")
```

A sample output might be:

```
11-12-15 03:29:12
```

**EXISTS**
returns the full path specification of the stream, if it exists, or a null string. For example:

Example 5.153. Stream object — QUERY method

```
stream-query("exists")
```

A sample output might be:

```
c:\data\file.txt
```

**HANDLE**
returns the handle associated with the open stream.
Example 5.154. Stream object — QUERY method

```c
stream-query("handle")
```

A sample output might be:

```
3
```

POSITION
returns the current read or write position for the stream, as qualified by the following options:

READ
returns the current read position.

WRITE
returns the current write position.

Note
If the stream is open for both reading and writing, this returns the read position by default. Otherwise, this returns the specified position.

CHAR
returns the position in terms of characters. This is the default.

LINE
returns the position in terms of lines. For non-binary streams, this operation can take a long time to complete. This is because the language processor starts tracking the current line number if not already doing so, and, thus, might require a scan of the stream from the top to count the line-end characters. See Section 14.1.5, “Line versus Character Positioning” for a detailed discussion of this issue.

Example 5.155. Stream object — QUERY method

```c
stream-query("position write")
```

A sample output might be:

```
247
```

SYS
returns the operating system stream position in terms of characters.

SIZE
returns the size, in bytes, of a persistent stream.
Example 5.156. Stream object — QUERY method

```plaintext
stream-query("size")
```

A sample output might be:

```
1305
```

STREAMTYPE
returns a string indicating whether the stream object is **PERSISTENT**, **TRANSIENT**, or **UNKNOWN**.

TIMESTAMP
returns the date and time stamps of a persistent stream in an international format. This is the preferred method of getting the date and time because it provides the full 4-digit year.

Example 5.157. Stream object — QUERY method

```plaintext
stream-query("timestamp")
```

A sample output might be:

```
2015-11-12 03:29:12
```

### 5.2.4.20. say

```
say(string)
```

Returns 0 if successful in writing `string` to the output stream or 1 if an error occurs while writing the line.

### 5.2.4.21. seek

```
seek(= < + - offset)
```

Sets the read or write position to a given number (`offset`) within a persistent stream. If the stream is open for both reading and writing and you do not specify READ or WRITE, both the read and write positions are set.
To use this method, you must first open the stream object (with the open method or implicitly with an input or output operation). One of the following characters can precede the offset number:

- Explicitly specifies the offset from the beginning of the stream. This is the default if you supply no prefix. For example, an offset of 1 means the beginning of the stream.

< Specifies offset from the end of the stream.

+ Specifies offset forward from the current read or write position.

- Specifies offset backward from the current read or write position.

The seek method returns the new position in the stream if the read or write position is successfully located, or an appropriate error message.

The following is a description of the options for seek:

READ
 specifies that the read position be set.

WRITE
 specifies that the write position be set.

CHAR
 specifies that positioning be done in terms of characters. This is the default.

LINE
 specifies that the positioning be done in terms of lines. For non-binary streams, this is potentially an operation that can take a long time to complete because, in most cases, the file must be scanned from the top to count the line-end characters. However, for binary streams with a specified record length, the new resulting line number is simply multiplied by the record length before character positioning. See Section 14.1.5, “Line versus Character Positioning” for a detailed discussion of this issue.

Note

If you do line positioning in a file open only for writing, you receive an error message.
Example 5.158. Stream object — SEEK method

<table>
<thead>
<tr>
<th>Method Call</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream-&gt;seek(&quot;=2 read&quot;)</code></td>
</tr>
<tr>
<td><code>stream-&gt;seek(&quot;+15 read&quot;)</code></td>
</tr>
<tr>
<td><code>stream-&gt;seek(&quot;-7 write line&quot;)</code></td>
</tr>
<tr>
<td><code>fromend = 125</code></td>
</tr>
<tr>
<td><code>stream-&gt;seek(&quot;&lt;&quot;fromend read)</code></td>
</tr>
</tbody>
</table>

5.2.4.22. state

`state` method

Returns a string indicating the current stream state.

The returned strings are as follows:

**ERROR**

The stream has been subject to an erroneous operation (possibly during input, output, or through the various Stream methods). See Section 14.5, “Errors during Input and Output”. You might be able to obtain additional information about the error with the `description` method.

**NOTREADY**

The stream is known to be in such a state that the usual input or output operations attempted upon would raise the NOTREADY condition. (See Section 14.5, “Errors during Input and Output”.) For example, a simple input stream can have a defined length. An attempt to read that stream (with `charIn` or `lineIn`, perhaps) beyond that limit can make the stream unavailable until the stream has been closed (for example, with the `close` method) and then reopened.

**READY**

The stream is known to be in such a state that the usual input or output operations might be attempted. This is the usual state for a stream, although it does not guarantee that any particular operation will succeed.

**UNKNOWN**

The state of the stream is unknown. This generally means that the stream is closed or has not yet been opened.

5.2.4.23. string

`string` method

Returns a string that indicates the name of the object the stream represents i.e. the name of the file.

5.2.4.24. supplier

`supplier` method

Returns a `StreamSupplier` object for the stream containing the remaining stream lines and linenumber positions for the stream.
5.2.4.25. uninit

This method cleans up the object when it is garbage collected. It should not be invoked directly except via an uninit method of a subclass of the Stream class.

If the Stream class is subclassed and the subclass provides an uninit method then that method must invoke the superclass uninit method.

Example 5.159. Stream object — UNINIT method

```rexx
class CustomStream subclass Stream
...
method uninit
  /* the subclass instance cleanup code should be placed here */
  super~uninit -- this should be the last action in the method
return
```

5.3. Collection Classes

A Collection is an object that contains a number of items, which can be any objects. Every item stored in a Collection has an associated index that you can use to retrieve the item from the collection with the at or [] methods.

Each Collection defines its own acceptable index types. Rexx provides the following Collection classes:

**Array Class**
A sequenced collection of objects ordered by whole-number indexes.

**Bag Class**
A collection where the index and the item are the same object. Bag indexes can be any object and each index can appear more than once.

**CircularQueue Class**
The CircularQueue class allows for storing objects in a circular queue of a predefined size. Once the end of the queue has been reached, new item objects are inserted from the beginning, replacing earlier entries. The collected objects can be processed in FIFO (first-in, first-out) or in a stack-like LIFO (last-in, first-out) order.

**Directory Class**
A collection with character string indexes. Index comparisons are performed using the string == comparison method.

**IdentityTable Class**
A collection with indexes that can be any object. The IdentityTable class determines index item matches by using an object identity comparison. With object identity matches, an index will only match the same object instance. An identity table contains no duplicate indexes.
**List Class**
A sequenced collection that lets you add new items at any position in the sequence. A list generates and returns an index value for each item placed in the list. The returned index remains valid until the item is removed from the list.

**Properties Class**
A collection with character string indexes and values. Properties collections include support for saving and loading from disk files.

**Queue Class**
A sequenced collection with the items ordered as a queue. You can remove items from the head of the queue and add items at either its tail or its head. Queues index the items with whole-number indexes, in the order in which the items would be removed. The current head of the queue has index 1, the item after the head item has index 2, up to the number of items in the queue.

**Relation Class**
A collection with indexes that can be any object. A relation can contain duplicate indexes.

**Set Class**
A collection where the index and the item are the same object. Set indexes can be any object and each index is unique.

**Stem Class**
A collection with character string indexes constructed from one or more string segments. Index comparisons are performed using the string `==` comparison method.

**StringTable Class**
A collection with character string indexes. Index comparisons are performed using the string `==` comparison method.

**Table Class**
A collection with indexes that can be any object. A table contains no duplicate indexes.

### 5.3.1. Organization of the Collection Classes

The following shows the logical organization of the Collection Classes. This does not represent the order that methods are inherited but rather the organization of the classes.

**Collection Class**
- **MapCollection** classes
  - *Directory Class*
  - *IdentityTable Class*
  - *Properties Class*
  - *Relation Class*
  - *Stem Class*
  - *StringTable Class*
  - *Table Class*
- **OrderedCollection** classes
  - *Array Class*
  - *CircularQueue Class*
  - *List Class*
  - *Queue Class*
5.3.2. Collection Class

The **Collection** class is a MIXIN class that defines the basic set of methods implemented by all Collections. Many of the Collection class methods are abstract and must be implemented by the inheriting subclasses.

### Table 5.12. Collection Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>index</code></td>
<td>Returns the item associated with the specified index or indexes. If the collection has no item associated with the specified index or indexes, this method returns <code>.nil</code>. This is an abstract method that must be implemented by a subclasses.</td>
</tr>
<tr>
<td><code>put</code></td>
<td>Adds an item to the collection at the specified index. This is an abstract method that must be implemented by subclasses.</td>
</tr>
<tr>
<td><code>allIndexes</code></td>
<td>Returns an array of all indexes used by this collection. This is an abstract method that must be implemented by subclasses.</td>
</tr>
</tbody>
</table>
5.3.2.4. allItems (Abstract Method)

Returns an array containing all items stored in the collection. This is an abstract method that must be implemented by subclasses.

5.3.2.5. at (Abstract Method)

Returns the item associated with the specified index or indexes. If the collection has no item associated with the specified index or indexes, this method returns .nil. This is an abstract method that must be implemented by subclasses.

5.3.2.6. difference

Returns a new collection (of the same class as the receiver) containing only those items from the receiver whose indexes the argument collection does not contain. The argument can be a Collection object or any other object that supports a makeArray method. The argument must also allow all of the index values in the receiver collection.

5.3.2.7. disjoint

Returns .true if the receiver collection and argument collection do not have any items in common. .false otherwise. The argument can be a Collection object or any other object that supports a makeArray method. The argument must also allow all of the index values in the receiver collection.

5.3.2.8. equivalent

Returns .true if all indexes in the receiver collection are also contained in the argument collection and both collections contain the same number of items; returns .false otherwise. The argument can be a Collection object or any other object that supports a makeArray method. The argument must also allow all of the index values in the receiver collection.

5.3.2.9. hasIndex
Returns `.true` if the receiver collection contains an item associated with the specified index or indexes. Returns `.false` otherwise.

### 5.3.2.10. hasItem

![hasItem](image)

Returns `.true` if the collection contains the specified item at any index location. Returns `.false` otherwise.

### 5.3.2.11. index (Abstract Method)

![index](image)

Returns the index associated with `item`. If `item` occurs more than once in the collection, the returned index value is undetermined. This is an abstract method which must be implemented by a subclass of this class.

### 5.3.2.12. intersection

![intersection](image)

Returns a new collection (of the same class as the receiver) containing only those items from the receiver whose indexes are in both the receiver collection and the `argument` collection. The `argument` can be a `Collection` object or any other object that supports a `makeArray` method. The `argument` must also allow all of the index values in the receiver collection.

### 5.3.2.13. items

![items](image)

Returns the number of items in the collection.

### 5.3.2.14. makeArray

![makeArray](image)

Returns a single-dimensional Array with the same number of items as the receiver object. Any index with no associated item is omitted from the new array. Items in the new array will have the same order as the source array.

### 5.3.2.15. put (Abstract Method)

![put](image)

Adds an item to the collection at the specified index. This is an abstract method that must be implemented by a subclass of this class.
5.3.2.16. subset

```
subset(argument)
```

Returns `.true` if all indexes in the receiver collection are also contained in the `argument` collection; returns `.false` otherwise. The `argument` can be a `Collection` object or any other object that supports a `makeArray` method. The `argument` must also allow all of the index values in the receiver collection.

5.3.2.17. supplier

```
supplier
```

Returns a `Supplier` object for the collection. The supplier allows you to enumerate through the index/item pairs for the collection. The supplier is created from a snapshot of the collection and is unaffected by subsequent changes to the collection's contents.

5.3.2.18. union

```
union(argument)
```

Returns a new collection of the same class as the receiver that contains all the items from the receiver collection and selected items from the `argument` collection. This method includes an item from `argument` in the new collection only if there is no item with the same associated index in the receiver collection and the method has not already included an item with the same index. The order in which this method selects items in `argument` is unspecified (the program should not rely on any order). The `argument` can be a `Collection` object or any other object that supports a `makeArray` method. The `argument` must also allow all of the index values in the receiver collection.

5.3.2.19. xor

```
xor(argument)
```

Returns a new collection of the same class as the receiver that contains all items from the receiver collection and the `argument` collection; all indexes that appear in both collections are removed. The `argument` can be a `Collection` object or any other object that supports a `makeArray` method. The `argument` must also allow all of the index values in the receiver collection.

5.3.3. MapCollection Class

The `MapCollection` class is a MIXIN class that defines the basic set of methods implemented by all collections that create a mapping from an index object to a value.

This class is defined as a MIXIN class. The following classes inherit from `MapCollection`: `Directory`, `IdentityTable`, `Properties`, `Relation`, `Stem`, `StringTable`, `Table`, and `SetCollection` classes `Bag` and `Set`.

Table 5.13. MapCollection Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
</table>

241
## 5.3.3.1. of (Class Method)

```kotlin
of(pair)
```

Returns a newly created MapCollection object containing the specified index/item pairs. Each pair must be a single-dimensional Array with exactly two items: the index as the first Array item, and the value as the second item. The pairs are processed left-to-right and added to the MapCollection object.

### Example 5.160. MapCollection class — of method

```kotlin
say iso639~allIndexes~makeString(, ", ")           -- de, en, fr
say iso639~allItems~makeString(, ", ")             -- Deutsch, English, français
```

### 5.3.3.2. makeArray

```kotlin
makeArray
```

Returns a single-dimensional Array of the index values used by the receiver object. The index objects will not be ordered in any predictable order.

### 5.3.3.3. putAll

```kotlin
putAll(collection)
```

Returns the result of adding the elements of the specified collection to the receiver object.
Returns the receiving collection with all items in `collection` added to it. The `collection` argument can be any object that supports a `supplier` method. Items from `collection` are added using the index values returned by the supplier. The item indexes from the source `collection` must be strings. The items are added in the order provided by the supplier object. If duplicate indexes exist in `collection`, the last item provided by the supplier will overwrite previous items with the same index.

### 5.3.4. OrderedCollection Class

The **OrderedCollection** class is a MIXIN class that defines the basic set of methods implemented by all collections that have an inherent index ordering.

This class is defined as a MIXIN class. The following classes inherit from OrderedCollection: **Array**, **CircularQueue**, **List**, and **Queue**.

<table>
<thead>
<tr>
<th>Table 5.14. OrderedCollection Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
</tr>
<tr>
<td>Methods inherited from the <strong>Object class</strong></td>
</tr>
<tr>
<td><strong>Class (Metaclass)</strong></td>
</tr>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
<tr>
<td><strong>+ Collection (Mixin Class)</strong></td>
</tr>
<tr>
<td>Methods inherited from the <strong>Collection class</strong></td>
</tr>
<tr>
<td><code>[]</code> (Abstract Method)</td>
</tr>
<tr>
<td><code>[]=</code> (Abstract Method)</td>
</tr>
<tr>
<td>allIndexes (Abstract Method)</td>
</tr>
<tr>
<td>allItems (Abstract Method)</td>
</tr>
<tr>
<td>at (Abstract Method)</td>
</tr>
<tr>
<td>difference</td>
</tr>
<tr>
<td>disjoint</td>
</tr>
<tr>
<td>equivalent</td>
</tr>
<tr>
<td>hasIndex</td>
</tr>
<tr>
<td>hasItem</td>
</tr>
<tr>
<td>index (Abstract Method)</td>
</tr>
<tr>
<td>intersection</td>
</tr>
<tr>
<td>items</td>
</tr>
<tr>
<td>makeArray</td>
</tr>
<tr>
<td>put (Abstract Method)</td>
</tr>
<tr>
<td>subset</td>
</tr>
<tr>
<td>supplier</td>
</tr>
<tr>
<td>union</td>
</tr>
<tr>
<td>xor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>OrderedCollection (Mixin Class)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>append (Abstract Method)</td>
</tr>
<tr>
<td>appendAll</td>
</tr>
<tr>
<td>delete (Abstract Method)</td>
</tr>
<tr>
<td>difference</td>
</tr>
<tr>
<td>first (Abstract Method)</td>
</tr>
<tr>
<td>firstItem (Abstract Method)</td>
</tr>
<tr>
<td>insert (Abstract Method)</td>
</tr>
<tr>
<td>intersection</td>
</tr>
<tr>
<td>last (Abstract Method)</td>
</tr>
<tr>
<td>lastItem (Abstract Method)</td>
</tr>
<tr>
<td>next (Abstract Method)</td>
</tr>
<tr>
<td>previous (Abstract Method)</td>
</tr>
<tr>
<td>section (Abstract Method)</td>
</tr>
<tr>
<td>sort</td>
</tr>
<tr>
<td>sortWith</td>
</tr>
<tr>
<td>stableSort</td>
</tr>
<tr>
<td>stableSortWith</td>
</tr>
<tr>
<td>subset</td>
</tr>
<tr>
<td>union</td>
</tr>
<tr>
<td>xor</td>
</tr>
</tbody>
</table>

#### 5.3.4.1. append (Abstract Method)

![append](append_icon.png)

Append an item to the end of the collection ordering. This is an abstract method that must be implemented by a subclass of this class.

#### 5.3.4.2. appendAll

![appendAll](appendAll_icon.png)

Append all items from another collection added to the end of the collection ordering.
OrderedCollection Class

Returns the receiving collection with all items in collection appended to the end of it. The collection may be any object that implements an allItems method.

5.3.4.3. delete (Abstract Method)

<table>
<thead>
<tr>
<th>delete(index)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.nil</td>
</tr>
</tbody>
</table>

Returns and deletes the member item with the specified index from the collection. If there is no item with the specified index, .nil is returned and no item is deleted. All elements following the deleted item will be moved up in the collection ordering and the size of the collection will be reduced by one element. Depending on the nature of the collection, the indexes of the moved items may be modified by the deletion.

5.3.4.4. difference

<table>
<thead>
<tr>
<th>difference(argument)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.nil</td>
</tr>
</tbody>
</table>

Returns a new collection (of the same class as the receiver) containing only those items from the receiver that are not also contained in the argument collection. The argument can be a Collection object or any other object that supports a makeArray method.

5.3.4.5. first (Abstract Method)

<table>
<thead>
<tr>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>.nil</td>
</tr>
</tbody>
</table>

Returns the index of the first item in the collection order. Returns .nil if the collection is empty.

5.3.4.6. firstItem (Abstract Method)

<table>
<thead>
<tr>
<th>firstItem</th>
</tr>
</thead>
<tbody>
<tr>
<td>.nil</td>
</tr>
</tbody>
</table>

Returns the first item in the collection order. Returns .nil if the collection is empty.

5.3.4.7. insert (Abstract Method)

<table>
<thead>
<tr>
<th>insert(item, index)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.nil</td>
</tr>
</tbody>
</table>

Returns a collection-supplied index for item item, which is added to the collection. The inserted item follows an existing item with index index in the collection ordering. If index is .nil, item becomes the first item in the ordered collection. If you omit index, the item becomes the last item in the collection.

Inserting an item in the collection at position index will cause the items in the collection after position index to have their relative positions shifted by the collection object. Depending on the nature of the collection, the index values for any items already in the collection may be modified by the insertion.

This is an abstract method that must be implemented by a subclass of this class.
5.3.4.8. intersection

Returns a new collection (of the same class as the receiver) containing only those items from the receiver that are in both the receiver collection and the argument collection. The argument can be a Collection object or any other object that supports a makeArray method.

5.3.4.9. last (Abstract Method)

Returns the index of the last item in the collection order. Returns .nil if the collection is empty.

5.3.4.10. lastItem (Abstract Method)

Returns the first item in the collection order. Returns .nil if the collection is empty.

5.3.4.11. next (Abstract Method)

Returns the index of the item that follows the collection item having index index or returns .nil if the item having that index is last in the collection.

5.3.4.12. previous (Abstract Method)

Returns the index of the item that precedes the collection item having index index or returns .nil if the item having that index is first in the collection.

5.3.4.13. section (Abstract Method)

Returns a new collection (of the same class as the receiver) containing selected items from the receiver. The first item in the new collection is the item corresponding to index start in the receiver. Subsequent items in the new collection correspond to those in the receiver, in the same sequence. If you specify the whole number items, the new collection contains only this number of items (or the number of subsequent items in the receiver, if this is less than items). If you do not specify items, the new collection contains all subsequent items of the receiver. The receiver remains unchanged.

5.3.4.14. sort
OrderedCollection Class

sort

Sorts the collection of Comparable items into ascending order using an algorithm that is not
guaranteed to be stable, and returns the sorted collection. See Section 5.3.19, “Sorting Ordered
Collections” for details.

5.3.4.15. sortWith

sortWith( comparator )

Sorts the collection of items into ascending order using an algorithm that is not guaranteed to be
stable, and returns the sorted collection. Ordering of elements is determined using the comparator
argument. See Section 5.3.19, “Sorting Ordered Collections” for details.

5.3.4.16. stableSort

stableSort

Sorts the collection of Comparable items into ascending order using a stable Mergesort algorithm, and
returns the sorted collection. See Section 5.3.19, “Sorting Ordered Collections” for details.

5.3.4.17. stableSortWith

stableSortWith( comparator )

Sorts the collection of items into ascending order using a stable Mergesort algorithm, and returns
the sorted collection. Ordering of elements is determined using the comparator argument. See
Section 5.3.19, “Sorting Ordered Collections” for details.

5.3.4.18. subset

subset( argument )

Returns .true if all items in the receiver collection are also contained in the argument collection;
returns .false otherwise. The argument can be a Collection object or any other object that
supports a makeArray method.

5.3.4.19. union

union( argument )

Returns a new collection of the same class as the receiver that contains all the items from the
receiver collection and selected items from the argument collection. This method includes an item
from argument in the new collection only if there is no equivalent item in the receiver collection and
the method has not already included. The order in which this method selects items in argument is
unspecified (the program should not rely on any order). The argument can be a Collection object or
any other object that supports a makeArray method.
5.3.4.20. xor

Returns a new collection of the same class as the receiver that contains all items from the receiver collection and the argument collection; all items that appear in both collections are removed. The argument can be a Collection object or any other object that supports a makeArray method.

5.3.5. SetCollection Class

This is a tagging MIXIN class only and does not define any methods of its own. Collections that implement SetCollection are MapCollections that constrain the index and item to be be the same object.

This class is defined as a MIXIN class. The following classes inherit from SetCollection: Bag and Set.

<table>
<thead>
<tr>
<th>Table 5.15. SetCollection Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td>Object</td>
</tr>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td>Class (MetaClass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>+ Collection (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the Collection class</td>
</tr>
<tr>
<td>equivalent</td>
</tr>
<tr>
<td>put (Abstract Method)</td>
</tr>
<tr>
<td>hasIndex</td>
</tr>
<tr>
<td>subset</td>
</tr>
<tr>
<td>hasItem</td>
</tr>
<tr>
<td>supplier</td>
</tr>
<tr>
<td>index (Abstract Method)</td>
</tr>
<tr>
<td>union</td>
</tr>
<tr>
<td>items</td>
</tr>
<tr>
<td>xor</td>
</tr>
<tr>
<td>allIndexes (Abstract Method)</td>
</tr>
<tr>
<td>makeArray</td>
</tr>
<tr>
<td>allItems (Abstract Method)</td>
</tr>
<tr>
<td>intersection</td>
</tr>
<tr>
<td>at (Abstract Method)</td>
</tr>
<tr>
<td>disjoint</td>
</tr>
<tr>
<td>difference</td>
</tr>
<tr>
<td>(no class or instance methods)</td>
</tr>
</tbody>
</table>

5.3.6. Array Class

An Array is a possibly sparse collection with indexes that are positive whole numbers. You can reference Array items by using one or more indexes. The number of indexes is the same as the number of dimensions of the Array. This number is called the dimensionality of the Array.

Array items can be any valid Rexx object.

<table>
<thead>
<tr>
<th>Table 5.16. Array Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>Object</td>
</tr>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td>Class (MetaClass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>+ Collection (Mixin Class)</td>
</tr>
</tbody>
</table>
Array objects are variable-sized. The dimensionality of an array is fixed, but the size of each dimension is variable. When you create an array, you can specify a hint about how many elements you expect to put into the array or the array's dimensionality. However, you do not need to specify a size or dimensionality of an array when you are creating it. You can use any whole-number indexes to reference items in an array.

For any array method that takes an index, the index may be specified as either individual arguments or as an array of indexes. For example, the following are equivalent:

Example 5.161. Array class — [] method

```plaintext
x = myarray[1,2,3]   -- retrieves an item from a multi-dimensional Array
index = .array-(1,2,3)  -- create an index list
x = myarray[index]   -- also retrieves from "1,2,3"
```

Methods such as `index` that return index items will return a single numeric value for single-dimensional Arrays and an array of indexes for multi-dimensional Arrays.
5.3.6.1. new (Class Method)

```
new(size)
```

Returns a new empty array. If you specify any size arguments, the size is taken as a hint about how big each dimension should be. The Array class uses this only to allocate the initial array object. For multi-dimensional Arrays, you can also specify how much space is to be allocated initially for each dimension of the array.

Each size argument must a non-negative whole number. If it is 0, the corresponding dimension is initially empty. The dimensions may also be specified with one single-dimensional Array of sizes.

**Example 5.162. Array class — of method**

```
a = .array~new()        -- create a new, empty array
da = .array~new(3,3)     -- create a new 3x3 array
sizes = .array~of(4,4,4)
a = .array~new(sizes)   -- create new 4x4x4 array
```

5.3.6.2. of (Class Method)

```
of(item)
```

Returns a newly created single-dimensional Array containing the specified item objects. The first item has index 1, the second has index 2, and so on.

If you use the of method and omit any argument items, the returned array does not include the indexes corresponding to the omitted arguments.

**Example 5.163. Array class — of method**

```
a = .array-of("Fred", "Mike", "David")
do name over a
    say name  -- displays "Fred", "Mike", and "David"
end
```

5.3.6.3. []

```
[]
```

Returns the same value as the at method.

Note that the index argument may also be specified as an array of indexes.
5.3.6.4. `[]=`

This method is the same as the `put` method.

Note that the `index` argument may also be specified as an array of indexes.

5.3.6.5. allIndexes

Returns an array of all index positions in the array containing items. For multi-dimensional Arrays, each returned index will be an array of index values.

Example 5.164. Array class — allIndexes method

```plaintext
a = .array~of("Fred", "Mike", "David")
do name over a~allIndexes
   say name -- displays "1", "2", and "3"
end
a~remove(2) -- remove second item
do name over a~allIndexes
   say name -- displays "1" and "3"
end
```

5.3.6.6. allItems

Returns an array of all items contained in the array.

Example 5.165. Array class — allItems method

```plaintext
a = .array~of("Fred", "Mike", "David")
do name over a~allItems
   say name -- displays "Fred", "Mike", and "David"
end
a~remove(2) -- remove second item
do name over a~allItems
   say name -- displays "Fred" and "David"
end
```

5.3.6.7. append
Array Class

append(item)

Appends an item to the array after the last item (the item with the highest index). The return value is the index of the newly added item. The append method is only valid with single-dimensional Arrays.

Example 5.166. Array class — append method

```plaintext
a = .array-of("Mike", "Rick")
a-append("Fred")  -- a = .array-of("Mike", "Rick", "Fred")
```

5.3.6.8. at

at(index)

Returns the item associated with the specified index or indexes. If the array has no item associated with the specified index or indexes, this method returns .nil.

Note that the index argument may also be specified as an array of indexes.

Example 5.167. Array class — at method

```plaintext
a = .array-of("Mike", "Rick")
say a-at(2)  -- says: "Rick"
```

5.3.6.9. delete

delete(index)

Returns and deletes the member item with the specified index from the array. If there is no item with the specified index, .nil is returned and no item is deleted. All elements following the deleted item will be moved up in the array ordering and the item indexes will be adjusted for the deletion. The size of the array will be reduced by one element.

The delete method is only valid with single-dimensional Arrays. The index argument may also be specified as an array of a single index.

Example 5.168. Array class — delete method

```plaintext
a = .array-of("Fred", "Mike", "Rick", "David")
a-delete(2)  -- removes "Mike", resulting in the array
            -- ("Fred", "Rick", "David")
```

5.3.6.10. dimension

dimension(n)

Returns the current size (upper bound) of dimension n (a positive whole number). If you omit n, this method returns the dimensionality (number of dimensions) of the array. If the number of dimensions has not been determined, 0 is returned.

251
Examples 5.169. Array class — dimension method

```ruby
a = .array~of("Mike", "Rick")
say a~dimension -- says: 1 (number of dimensions in the array)
say a~dimension(1) -- says: 3 (upper bound of dimension one)

a = .array-new~put("Mike",1,1)~put("Rick",1,2)
say a~dimension -- says: 2 (number of dimensions in the array)
say a~dimension(1) -- says: 1 (upper bound of dimension one)
say a~dimension(2) -- says: 2 (upper bound of dimension two)
```

5.3.6.11. dimensions

Returns an array containing each of the array dimension sizes. A single-dimensional Array will return an array with a single size element.

5.3.6.12. empty

Returns the receiving Array with all items removed.

Example 5.170. Array class — empty method

```ruby
a = .array~of("Mike", "Rick", "Fred", "Rick")
a~empty    -- a~items now returns "0"
```

5.3.6.13. fill

Returns the receiving Array with all index locations set to `value`.

Example 5.171. Array class — fill method

```ruby
a = .array-new(3,3)
a~fill(0)    -- initialize the matrix to all zeroes.
```

5.3.6.14. first

Returns the index of the first item in the array or `.nil` if the array is empty. For multi-dimensional Arrays, the index is returned as an array of index values.
Example 5.172. Array class — first method

```lisp
a = .array-of("Mike", "Rick", "Fred", "Rick")
say a-first -- says: 1
a = .array-of("Mike", "Rick")
say a-first -- says: 2
```

5.3.6.15. firstItem

Returns the first item in the array or **.nil** if the array is empty.

Example 5.173. Array class — firstItem method

```lisp
musketeers=.array-of("Porthos","Athos","Aramis") /* Creates array MUSKETEERS */
item=musketeers-firstItem                         /* Gives first item in array */
/* (Assigns "Porthos" to item) */
```

5.3.6.16. hasIndex

Returns **.true** if the array contains an item associated with the specified index or indexes. Returns **.false** otherwise.

Note that the *index* argument may also be specified as an array of indexes.

Example 5.174. Array class — hasIndex method

```lisp
a=.array-of("Mike", "Rick", "Fred", "Rick")
say a-hasIndex(2) -- says: 1
say a-hasIndex(5) -- says: 0
```

5.3.6.17. hasItem

Returns **.true** if the array contains the specified item at any index location. Returns **.false** otherwise. Item equality is determined by using the **==** method of *item*.

Example 5.175. Array class — hasItem method

```lisp
a=.array-of("Mike", "Rick", "Fred", "Rick")
say a-hasItem("Rick") -- says: 1
say a-hasItem("Mark") -- says: 0
```
5.3.6.18. index

Returns the index of the specified item within the array. If the target item appears at more than one index, the first located index will be returned. For multi-dimensional Arrays, the index is returned as an array of index values. If the array does not contain the specified item, .nil is returned. Item equality is determined by using the == method of item.

Example 5.176. Array class — index method

```ruby
a = .array-of("Mike", "Rick", "Fred", "Rick")
say a-index("Rick")  -- says: 2
```

5.3.6.19. insert

Returns an Array-supplied index for item item, which is added to the Array. The inserted item follows an existing item with index index in the Array ordering. If index is .nil, item becomes the first item in the Array. If you omit index, the item becomes the last item in the Array.

Inserting an item in the Array at position index will cause the items in the Array after position index to have their indexes shifted by the Array object. The index values for any items in the Array are incremented by the insertion.

Example 5.177. Array class — insert method

```ruby
musketeers=.Array~of("Porthos","Athos","Aramis") /* Creates Array MUSKETEERS        */ /* consisting of: Porthos */ /* Athos */ /* Aramis */
musketeers~insert("D'Artagnan",1)    /* Adds D'Artagnan after Porthos  */ /* Array is now: Porthos */ /* D'Artagnan */ /* Athos */ /* Aramis */
```
Array Class

isEmtpy

Returns .true if the array is empty. Returns .false otherwise.

Example 5.178. Array class — isEmpty method

```plaintext
a = .array-new
say a-isEmpty  -- says: 1
a[1] = "1"
say a-isEmpty  -- says: 0
```

5.3.6.21. items

Returns the number of items in the array.

Example 5.179. Array class — items method

```plaintext
a = .array-of("Fred", , "Mike", , "David")
say a-items  -- says: 3
```

5.3.6.22. last

Returns the index of the last item in the array or .nil if the array is empty. For multi-dimensional Arrays, index is returned as an array of index items.

Example 5.180. Array class — last method

```plaintext
a = .array-of("Fred", , "Mike", , "David")
say a-last  -- says: 5
```

5.3.6.23. lastItem

Returns the last item in the array or .nil if the array is empty.

Example 5.181. Array class — lastItem method

```plaintext
musketeers=.array-of("Porthos","Athos","Aramis") /* Creates array MUSKETEERS */
item=musketeers-lastItem                          /* Gives last item in array */
/* (Assigns "Aramis" to item) */
```

5.3.6.24. makeArray
Array Class

**makeArray**

Returns a single-dimensional Array with the same number of items as the receiver object. Any index with no associated item is omitted from the new array. Items in the new array will have the same order as the source array. A multi-dimensional Array will be converted into a non-sparse single-dimensional Array.

**Example 5.182. Array class — makeArray method**

```plaintext
a = .array~of("Fred", , "Mike", , "David")
b = a~makeArray  -- b = .array~of("Fred", "Mike", "David")
```

### 5.3.6.25. makeString

**makeString**

Returns a string that contains the data of an array (one to n dimensional). The elements of the array are treated either in line or character format, starting at the first element in the array. The line format is the default. If the line format is used, a separator string can be specified. The separator will be used between concatenated elements instead of the default line end separator.

See method `toString` (which is a synonym for this method) for examples.

### 5.3.6.26. next

**next**

Returns the index of the item that follows the array item having index `index` or returns `.nil` if the item having that index is last in the array. For multi-dimensional Arrays, the same ordering as used by the `allItems` method is used to determine the next position and the index is returned as an array of index values.

Note that the `index` argument may also be specified as an array of indexes.

**Example 5.183. Array class — next method**

```plaintext
a = .array~of("Fred", , "Mike", , "David")
say a~next(3)  -- says: 5
```

### 5.3.6.27. previous

**previous**

Returns the index of the item that precedes the array item having index `index` or returns `.nil` if the item having that index is first in the array. For multi-dimensional Arrays, the same ordering as used by the `allItems` method is used to determine the previous position and the index is returned as an array of index values.
Array Class

Returns the index of the item that precedes the array item having index \textit{index} or \texttt{.nil} if the item having that index is first in the array. For multi-dimensional Arrays, the same ordering used by the \texttt{allItems} method is used to determine the previous position and the index is returned as an array of index values.

Note that the \textit{index} argument may also be specified as an array of indexes.

\begin{example}
\textbf{Example 5.184. Array class — previous method}
\begin{verbatim}
a = .array-of("Fred", , "Mike", , "David")
say a-previous(3) -- says: 1
\end{verbatim}
\end{example}

\subsection*{5.3.6.28. put}

\begin{tikzpicture}
    \node (query) {put};
    \node [right of=query] (item) {item};
    \node [right of=item] (index) {index};
    \draw [->] (query) to (item);
    \draw [->] (item) to (index);
    \draw [->] (index) to [out=180,in=90] (query);
\end{tikzpicture}

Makes the object \texttt{item} a member item of the array and associates it with the specified \textit{index} or indexes. This replaces any existing item associated with the specified \textit{index} or indexes with the new item. If the \textit{index} for a particular dimension is greater than the current size of that dimension, the array is expanded to the new dimension size.

Note that the \textit{index} argument may also be specified as an array of indexes.

\begin{example}
\textbf{Example 5.185. Array class — put method}
\begin{verbatim}
a = .array-new
a-put("Fred", 1) -- a = .array-of("Fred")
a-put("Mike", 2) -- a = .array-of("Fred", "Mike")
a-put("Mike", 1) -- a = .array-of("Mike", "Mike")

do name over a
    say name
end

/* Output would be: */
Mike
Mike
\end{verbatim}
\end{example}

\subsection*{5.3.6.29. remove}

\begin{tikzpicture}
    \node (query) {remove};
    \node [right of=query] (index) {index};
    \draw [->] (query) to (index);
\end{tikzpicture}

Returns and removes the member item with the specified \textit{index} or indexes from the array. If there is no item with the specified \textit{index} or indexes, \texttt{.nil} is returned and no item is removed. The index of the removed item becomes unused and the \texttt{hasIndex} method for the given index will now return \texttt{.false}. The size of the array is unchanged and no other indexes of the array are modified with the removal.

Note that the \textit{index} argument may also be specified as an array of indexes.
Example 5.186. Array class — remove method

```plaintext
a = .array-of("Fred", "Mike", "Mike", "David")
a~remove(2)  -- removes "Mike"
```

5.3.6.30. removeItem

Removes an item from the array. If the target item exists at more than one index, the first located item is removed. Item equality is determined by using the `==` method of `item`. The return value is the removed item.

Example 5.187. Array class — removeItem method

```plaintext
a = .array-of("Fred", "Mike", "Mike", "David")
a~removeItem("Mike")  -- removes the item at index "2"
```

5.3.6.31. section

Returns a new array (of the same class as the receiver) containing selected items from the receiver array. The first item in the new array is the item corresponding to index `start` in the receiver array. Subsequent items in the new array correspond to those in the receiver array (in the same sequence). If you specify the whole number `items`, the new array contains only this number of items (or the number of subsequent items in the receiver array, if this is less than `items`). If you do not specify `items`, the new array contains all subsequent items of the receiver array. The receiver array remains unchanged. The `section` method is valid only for single-dimensional Arrays.

Note that the index argument `start` may also be specified as an array of indexes.

Example 5.188. Array class — section method

```plaintext
a = .array-of(1,2,3,4)  -- Loads the array
b = a~section(2)   -- b = .array-of(2,3,4)
c = a~section(2,2) -- c = .array-of(2,3)
d = a~section(2,0) -- d = .array-new
```

5.3.6.32. size

Returns the number of items that can be placed in the array before it needs to be extended. This value is the same as the product of the sizes of the dimensions in the array.
5.3.6.33. sort

Sorts the Array of Comparable items into ascending order using an algorithm that is not guaranteed to be stable, and returns the sorted Array. See Section 5.3.19, “Sorting Ordered Collections” for details.

5.3.6.34. sortWith

Sorts the Array of items into ascending order using an algorithm that is not guaranteed to be stable, and returns the sorted Array. Ordering of elements is determined using the comparator argument. See Section 5.3.19, “Sorting Ordered Collections” for details.

5.3.6.35. stableSort

Sorts the Array of Comparable items into ascending order using a stable Mergesort algorithm, and returns the sorted Array. See Section 5.3.19, “Sorting Ordered Collections” for details.

5.3.6.36. stableSortWith

Sorts the Array of items into ascending order using a stable Mergesort algorithm, and returns the sorted Array. Ordering of elements is determined using the comparator argument. See Section 5.3.19, “Sorting Ordered Collections” for details.

5.3.6.37. supplier

Returns a Supplier object for the array. The supplier allows you to iterate over the index/item pairs of the array. The supplier enumerates the array items in their sequenced order. For multi-dimensional Arrays, the supplier index method will return the index values as an array of index numbers.

Example 5.189. Array class — supplier method

```
a = .array~of("Fred", "Mike", "David")
sup = a~supplier
da~append("Joe")
do while sup~available
  say sup~item  -- displays "Fred", "Mike", and "David"
sup~next
end
```

5.3.6.38. toString
Array Class

`toString()` returns a string that contains the data of an array (one to n dimensional). The elements of the array are treated either in line or character format, starting at the first element in the array. The line format is the default. If the line format is used, a `separator` string can be specified. The separator will be used between concatenated elements instead of the default line end separator.

See also method `makeString` for which this method is a synonym.

**Example 5.190. Array class — toString method**

```
a = .array~of(1,2,3,4)  -- Loads the array
say a~toString  -- Produces: 1
         --           2
         --           3
         --           4
say a~toString("c")  -- Produces: 1234
say a~toString(, ", ")  -- Produces: 1, 2, 3, 4
```

**5.3.6.39. Examples**

**Example 5.191. Array class — examples**

```
array1=.array~of(1,2,3,4)  /* Loads the array */
/* Alternative way to create and load an array */
array2=.array~new(4)  /* Creates array2, containing 4 items */
do i=1 to 4                   /* Loads the array */
   array2[i]=i
end
```

You can produce the elements loaded into an array, for example:

**Example 5.192. Array class — examples**

```
do i=1 to 4
   say array1[i]
end
```

If you omit any argument values before arguments you supply, the corresponding indexes are skipped in the returned array:

**Example 5.193. Array class — examples**

```
directions=.array~of("North","South", "West")
do i=1 to 4                                  /* Produces: North */
```
Here is an example using the `~`:

```plaintext
Example 5.194. Array class — examples

z=.array~of(1,2,3)~~put(4,4)
do i = 1 to z~size
   say z[i]              /* Produces:  1 2 3 4 */
end
```

### 5.3.7. Bag Class

A **Bag** is a non-sparse collection that restricts the elements to having an item that is the same as the index. Any object can be placed in a Bag, and the same object can be placed in a Bag several times.

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Object class</strong></td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
<tr>
<td>+ SetCollection (Mixin Class)</td>
</tr>
<tr>
<td>(no class or instance methods)</td>
</tr>
<tr>
<td>+ Collection (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>Collection</strong> class</td>
</tr>
<tr>
<td>⎕ (Abstract Method)</td>
</tr>
<tr>
<td>⎕= (Abstract Method)</td>
</tr>
<tr>
<td>allIndexes (Abstract Method)</td>
</tr>
<tr>
<td>allItems (Abstract Method)</td>
</tr>
<tr>
<td>at (Abstract Method)</td>
</tr>
<tr>
<td>difference</td>
</tr>
<tr>
<td>disjoint</td>
</tr>
<tr>
<td>+ MapCollection (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>MapCollection class</strong></td>
</tr>
<tr>
<td>of (Class Method)</td>
</tr>
<tr>
<td>makeArray</td>
</tr>
<tr>
<td>putAll</td>
</tr>
<tr>
<td>Bag</td>
</tr>
<tr>
<td>new (Class Method)</td>
</tr>
<tr>
<td>of (Class Method)</td>
</tr>
<tr>
<td>⎕</td>
</tr>
<tr>
<td>⎕=</td>
</tr>
<tr>
<td>allAt</td>
</tr>
<tr>
<td>allIndex</td>
</tr>
</tbody>
</table>
5.3.7.1. new (Class Method)

Returns a new empty Bag object.

If you specify size, a hint how large the new Bag object is expected to grow, this is used to optimize the initial allocation.

size must be a non-negative whole number.

5.3.7.2. of (Class Method)

Returns a newly created Bag containing the specified item objects.

5.3.7.3. []

Returns the item associated with index index. Returns .nil if the Bag has no item associated with index.

Index equality is determined by using the == method of index.

This method is the same as the at method.

5.3.7.4. allAt

Returns a single-dimensional Array containing all the items associated with index index.

Index equality is determined by using the == method of index.

This method has the same result as the allIndex method.

5.3.7.5. allIndex
Returns a single-dimensional Array containing all indexes for item `item`.

Item equality is determined by using the `==` method of `item`.

This method has the same result as the `allAt` method.

### 5.3.7.6. allIndexes

Returns an Array of all indexes contained in the Bag, in an unspecified order. The returned Array will have one index for every item stored in the Bag, including duplicates.

To retrieve the indexes without duplicates, use the `uniqueIndexes` method.

### 5.3.7.7. allItems

Returns an Array of all items contained in the Bag, in an unspecified order.

### 5.3.7.8. at

Returns the item associated with index `index`. Returns `.nil` if the Bag does not contain `index`.

Index equality is determined by using the `==` method of `index`.

This method is the same as the `[]` method.

### 5.3.7.9. [](index)=

Adds an item to the Bag. If specified, `index` must be the same object as `item`.

This method is the same as the `put` method.

### 5.3.7.10. difference

Returns a new Bag containing only those items from the receiver that the `argument` collection does not contain. The `argument` can be a `Collection` object or any other object that supports a `makeArray` method.
5.3.7.11. empty

Returns the receiving Bag with all items removed.

5.3.7.12. hasIndex

Returns .true if the Bag contains any item associated with index index, otherwise returns .false. Index equality is determined by using the == method of index.

5.3.7.13. hasItem

Returns .true if the Bag contains the member item item, otherwise returns .false. If index is specified, it should be the same object as index, otherwise hasItem will always return .false. Item and index equality is determined by using the == method.

5.3.7.14. index

Returns the index for item item if the Bag contains item, otherwise returns .false. Item equality is determined by using the == method of item.

5.3.7.15. intersection

Returns a new Bag containing only those items from the receiver that are also in the argument collection. The argument can be a Collection object or any other object that supports a makeArray method.

5.3.7.16. isEmpty

Returns .true if the Bag is empty. Returns .false otherwise.

5.3.7.17. items
Returns the number of Bag items with index *index*. If you specify no *index*, this method returns the total number of items in the Bag.

Index equality is determined by using the `==` method of *index*.

### 5.3.7.18. makeArray

Returns a single-dimensional Array containing all Bag items, in an unspecified order.

### 5.3.7.19. put

Adds an item to the Bag. If specified, *index* must be the same object as *item*.

This method is the same as the `[]=` method.

### 5.3.7.20. putAll

Returns the receiving Bag with all items in *collection* added to it. The *collection* argument can be any object that supports a `supplier` method. Items from *collection* are added using the item values returned by the `supplier`.

### 5.3.7.21. remove

Returns and removes from the Bag one member item with index *index*. Returns `.nil` if the Bag does not contain *index*.

Index equality is determined by using the `==` method of *index*.

### 5.3.7.22. removeAll

Returns and removes from the Bag all member items with index *index*. All removed items are returned in an Array containing each of the removed items. If the Bag does not contain *index*, an empty Array is returned.

Index equality is determined by using the `==` method of *index*. 
5.3.7.23. `removeItem`

```
removeItem(item, index)
```

Returns and removes from a Bag one member item `item`. Returns `.nil` if `item` is not a member of the Bag.

If `index` is specified, it should be the same object as `index`, otherwise `removeItem` will always return `.nil`

5.3.7.24. `subset`

```
subset(argument)
```

Returns `.true` if all indexes in the receiver Bag are also contained in the `argument` collection; returns `.false` otherwise. The `argument` can be a `Collection` object or any other object that supports a `makeArray` method.

5.3.7.25. `supplier`

```
supplier(index)
```

Returns a `Supplier` object for the Bag. The supplier allows you to iterate over all index/item pairs in the Bag at the time the supplier was created. The supplier enumerates the items in an unspecified order. If you specify `index`, the supplier contains all of the items with the specified index.

5.3.7.26. `union`

```
union(argument)
```

Returns a new Bag that contains all the items from the receiver Bag and selected items from the `argument` collection. The `argument` can be a `Collection` object or any other object that supports a `makeArray` method.

5.3.7.27. `uniqueIndexes`

```
uniqueIndexes
```

Returns an Array of all indexes contained in the Bag, with no duplicates.

5.3.7.28. `xor`

```
xor(argument)
```

Returns a new Bag that contains all items from the receiver Bag and the `argument` collection; items that appear in both collections are removed. The `argument` can be a `Collection` object or any other object that supports a `makeArray` method.
5.3.7.29. Examples

Example 5.195. Bag class — examples

```plaintext
/* Create a bag of fruit */
fruit = .bag~of("Apple", "Orange", "Apple", "Pear")
say fruit~items          /* How many pieces? (4) */
say fruit~items("Apple") /* How many apples? (2) */
fruit~remove("Apple")    /* Remove one of the apples. */
fruit~put("Banana")~put("Orange") /* Add a couple. */
say fruit~items          /* How many pieces? (5) */
```

5.3.8. CircularQueue Class

The **CircularQueue** class allows for storing objects in a circular queue of a predefined size. Once the end of the queue has been reached, new item objects are inserted from the beginning, replacing earlier entries. Any object can be placed in the queue and the same object can occupy more than one position in the queue.

The collected objects can be processed in FIFO (first-in, first-out) or in a stack-like LIFO (last-in, first-out) order.

Table 5.18. CircularQueue Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Object class</strong></td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
<tr>
<td>+ Collection (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>Collection class</strong></td>
</tr>
<tr>
<td>[] (Abstract Method)</td>
</tr>
<tr>
<td>[]= (Abstract Method)</td>
</tr>
<tr>
<td>allIndexes (Abstract Method)</td>
</tr>
<tr>
<td>allItems (Abstract Method)</td>
</tr>
<tr>
<td>at (Abstract Method)</td>
</tr>
<tr>
<td>difference</td>
</tr>
<tr>
<td>disjoint</td>
</tr>
<tr>
<td>+ OrderedCollection (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>OrderedCollection class</strong></td>
</tr>
<tr>
<td>append (Abstract Method)</td>
</tr>
<tr>
<td>appendAll</td>
</tr>
<tr>
<td>delete (Abstract Method)</td>
</tr>
<tr>
<td>difference</td>
</tr>
<tr>
<td>first (Abstract Method)</td>
</tr>
<tr>
<td>firstItem (Abstract Method)</td>
</tr>
<tr>
<td>insert (Abstract Method)</td>
</tr>
<tr>
<td>Queue</td>
</tr>
<tr>
<td>new (Class Method)</td>
</tr>
<tr>
<td>of (Class Method)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
5.3.8.1. of (Class Method)

Returns a newly created circular queue containing the specified *item* objects. The first *item* has index 1, the second has index 2, and so on. Any omitted item will be set to the null string. The total number of *item* objects (including omitted ones) determines the size of the circular queue.

5.3.8.2. init

Performs initialization of the circular queue. The required *size* argument, a non-negative whole number, specifies the initial size of the queue.

5.3.8.3. append

Append an item to the end of the collection ordering, returning the index of the added item. If the circular queue is full, then the first item will be deleted, before the insertion takes place.

5.3.8.4. insert
**CircularQueue Class**

```
CircularQueue insert(item, index)
```

Returns a queue-supplied index for `item`, which is added to the queue. The inserted item follows any existing item with index `index` in the queue ordering. If `index` is `nil`, `item` is inserted at the head of the queue. If you omit `index`, `item` becomes the last item in the queue.

Inserting an item in the queue at position `index` will cause the items in the queue after position `index` to have their indexes modified by the queue object. If inserting an object causes the queue to grow beyond the size, the last item in the queue will be removed.

### 5.3.8.5. intersection

```
CircularQueue intersection(argument)
```

Returns a new collection (of the same class as the receiver) containing only those items from the receiver that are in both the receiver collection and the `argument` collection. The `argument` can be a `Collection` object or any other object that supports a `makeArray` method. The resulting `CircularQueue` instance will be the same size as the receiver.

### 5.3.8.6. makeArray

```
CircularQueue makeArray(order)
```

Returns a single-dimensional Array containing the items of the circular queue in the specified `order`.

The following `order` can be used. (Only the capitalized letter is needed; all characters following it are ignored.)

- **Fifo**
  - First-in, first-out order. This is the default.

- **Lifo**
  - Last-in, first-out order (stack-like).

### 5.3.8.7. makeString

```
CircularQueue makeString(delimiter, order)
```

Returns a string object that concatenates the string values of the collected item objects, using the `delimiter` string to delimit them, in the specified `order`. If the `delimiter` is omitted, the comma character (", ") is used as the default delimiter string.

The following `order` can be used. (Only the capitalized letter is needed; all characters following it are ignored.)

- **Fifo**
  - First-in, first-out. This is the default
Lifo
Last-in, first-out (stack-like)

Example 5.196. CircularQueue class — makeArray method
```
-- reverse an array
a = .Array-of("one", "two", "three", "four", "five")
say .CircularQueue~new(a~size)~appendAll(a)~makeArray("lifo")~makeString(, ", ")
-- five, four, three, two, one
```

5.3.8.8. push

```
push( item, option )
```

Makes the object item a member item of the circular queue, inserting the item object in front of the first item in the queue. The pushed item object will be the new first item in the circular queue.

If the circular queue is full, then the last item stored in the circular queue will be deleted, before the insertion takes place. In this case the deleted item will be returned, otherwise .nil will be returned.

If option is specified, it may be "Normal" or "Unique". The default is "Normal". Only the capitalized letter is needed; all characters following it are ignored. If option is 'Unique', any matching items already in the queue will be removed before item is added to the queue. This allows you to maintain a list like the recent files list of an editor.

5.3.8.9. queue

```
queue( item, Normal, Unique )
```

Makes the object item a member item of the circular queue, inserting the item at the end of the circular queue.

If the circular queue is full, then the first item will be deleted, before the insertion takes place. In this case the deleted item will be returned, otherwise .nil will be returned.

If option is specified, it may be "Normal" or "Unique". The default is "Normal". Only the capitalized letter is needed; all characters following it are ignored. If option is 'Unique', any matching items already in the queue will be removed before item is added to the queue. This allows you to maintain a list like the recent files list of an editor.

5.3.8.10. resize

```
resize( newSize, order )
```

Resizes the circular queue object to be able to contain newSize items. If more than newSize items are in the queue, any extra items are removed in the specified order.
The following order can be used. (Only the capitalized letter is needed; all characters following it are ignored.)

**Fifo**

First-in, first-out. This keeps the items at the front of the queue. Item deletions occur at the end of the queue. This is the default action.

**Lifo**

Last-in, first-out (stack-like). This removes items from the front of the queue.

**Note:**

Resizing with a value of 0 removes all items from the circular queue.

### 5.3.8.11. section

```
section(start, items)
```

Returns a new queue (of the same class as the receiver) containing selected items from the receiver. The first item in the new queue is the item corresponding to index `start` in the receiver. Subsequent items in the new queue correspond to those in the receiver (in the same sequence). If you specify the whole number `items`, the new queue contains only this number of items (or the number of subsequent items in the receiver, if this is less than `items`). If you do not specify `items`, the new queue contains all subsequent items from the receiver. The receiver queue remains unchanged. The new queue item will be sized to the larger of the receiver queue size or `items`. The selection of the items will wrap from the end around to the beginning of the queue. For example,

```
q1 = circularqueue-of("Fred", "Mike", "David")
q2 = q1-section(2)       -- effectively rotates the queue

do name over a
   say name      -- displays "Mike", "David", and "Fred"
end
```

returns a new queue of three items, starting with the second item, effectively rotating the order of the contained items.

### 5.3.8.12. size

```
size
```

Returns the maximum number of objects that can be stored in the circular queue.

### 5.3.8.13. string

```
string(delimiter, order)
```


CircularQueue Class

Returns a string object that concatenates the string values of the collected item objects, using the delimiter string to delimit them, in the specified order. If the delimiter is omitted, the comma character ("," ) is used as the default delimiter string.

The following order can be used. (Only the capitalized letter is needed; all characters following it are ignored.)

Fifo
  First-in, first-out. This is the default

Lifo
  Last-in, first-out (stack-like)

5.3.8.14. supplier

Returns a Supplier object for the collection. The supplier allows you to iterate over the items that were in the queue at the time of the supplier's creation.

The supplier will iterate over the items in the specified order. (Only the capitalized letter is needed; all characters following it are ignored.)

Fifo
  First-in, first-out, default

Lifo
  Last-in, first-out (stack-like)

5.3.8.15. union

Returns a new collection of the same class as the receiver that contains all the items from the receiver collection and selected items from the argument collection. The resulting CircularQueue object will have a size that is the larger of the receiver's size or the union size of the two collections. This method includes an item from argument in the new collection only if there is no equivalent item in the receiver collection and the method has not already included. The order in which this method selects items in argument is unspecified (the program should not rely on any order.). The argument can be a Collection object or any other object that supports a makeArray method.

5.3.8.16. xor

Returns a new collection of the same class as the receiver that contains all items from the receiver collection and the argument collection; all items that appear in both collections are removed. The resulting CircularQueue object will have a size that is the larger of the receiver's size or the xor size of the two collections. The argument can be a Collection object or any other object that supports a makeArray method.
5.3.8.17. Example

Example 5.197. CircularQueue class — examples

```rexx
-- create a circular buffer with five items
u=circularQueue~of("a", "b", "c", "d", "e")
say "content: ", "content (LIFO): ", u~string("->","L")"
say
u~resize(4, "FIFO") -- resize fifo-style (keep newest)
say "after resizing to 4 items in FIFO style (keeping the newest):"
say "content: ", "content (LIFO): ", u~string("->","L")"
say
u~resize(2, "LIFO") -- resize lifo-style (keep oldest)
say "after resizing to 2 items in LIFO style (keeping the oldest):"
say "content: ", "content (LIFO): ", u~string("->","L")"
say
u~resize(0) -- resize lifo-style (keep oldest)
say "after resizing to 0 items, thereby deleting all items:"
say "content: ", "content (LIFO): ", u~string("->","L")"
say
u~resize(2) -- resize lifo-style (keep oldest)
say "after resizing to 2, size=", u~size "and items=", u~items
u~queue('x')~queue('y')~queue('z')
say "after queuing the three items 'x', 'y', 'z':"
say "content: ", "content (LIFO): ", u~string("->","L")"
say
u~push('1')~push('2')~push('3')
say "after pushing the three items '1', '2', '3':"
say "content: ", "content (LIFO): ", u~string("->","L")"
say
```

Output:

```
content: [a,b,c,d,e], content (LIFO): [e->d->c->b->a]
after resizing to 4 items in FIFO style (keeping the newest):
content: [b,c,d,e], content (LIFO): [e->d->c->b]
after resizing to 2 items in LIFO style (keeping the oldest):
content: [b,c], content (LIFO): [c->b]
after resizing to 0 items, thereby deleting all items:
content: [], content (LIFO): []
after resizing to 2, size=2 and items=0
after queuing the three items 'x', 'y', 'z':
content: [y,z], content (LIFO): [z->y]
after pushing the three items '1', '2', '3':
content: [3,2], content (LIFO): [2->3]
```

5.3.9. Directory Class

A Directory is a MapCollection using unique character string indexes. The items of a Directory can be any valid Rexx object.
See also StringTable Class, a MapCollection similar to Directory, but without methods setMethod and unsetMethod.

Table 5.19. Directory Class

<table>
<thead>
<tr>
<th>Methods inherited from the Object class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>+ Collection (Mixin Class)</td>
</tr>
</tbody>
</table>

Methods inherited from the Collection class

<table>
<thead>
<tr>
<th>Method</th>
<th>Equivalent</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>hasIndex</td>
<td>subset</td>
</tr>
<tr>
<td>[[]]</td>
<td>hasItem</td>
<td>supplier</td>
</tr>
<tr>
<td>allIndexes</td>
<td>index (Abstract Method)</td>
<td>union</td>
</tr>
<tr>
<td>allItems</td>
<td>items</td>
<td>xor</td>
</tr>
</tbody>
</table>

+ MapCollection (Mixin Class)

Methods inherited from the MapCollection class

<table>
<thead>
<tr>
<th>Method</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeArray</td>
<td></td>
</tr>
<tr>
<td>putAll</td>
<td></td>
</tr>
</tbody>
</table>

new (Class Method)

<table>
<thead>
<tr>
<th>Method</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>hasIndex</td>
</tr>
<tr>
<td>[[]]</td>
<td>hasItem</td>
</tr>
<tr>
<td>allIndexes</td>
<td>index</td>
</tr>
<tr>
<td>allItems</td>
<td>isEmpty</td>
</tr>
<tr>
<td>at</td>
<td>items</td>
</tr>
<tr>
<td>empty</td>
<td>makeArray</td>
</tr>
<tr>
<td>entry</td>
<td>put</td>
</tr>
<tr>
<td>hasEntry</td>
<td>remove</td>
</tr>
</tbody>
</table>

In addition to the standard put, [[]]=, at, and [] methods defined for Collections, Directory provides access to items using methods. For example:

```plaintext
mydir = .directory~new
mydir~name = "Mike"   -- same as mydir~put("Mike", "NAME")
say mydir~name        -- same as say mydir['NAME']
```

### 5.3.9.1. new (Class Method)

Returns an empty Directory object.

If you specify size, a hint how large the new Directory object is expected to grow, this is used to optimize the initial allocation.
size must be a non-negative whole number.

5.3.9.2. []

Returns the item corresponding to `name`. This method is the same as the `at` method.

5.3.9.3. []=

Adds or replaces the entry at index `name`. This method is the same as the `put` method.

5.3.9.4. allIndexes

Returns an array of all the directory indexes, including those of all the `setMethod` methods.

5.3.9.5. allItems

Returns an array of all items contained in the directory, including those returned by all `setMethod` methods.

5.3.9.6. at

Returns the item associated with index `name`. If a method defined using `setMethod` is associated with index `name`, the result of running this method is returned. If the Directory has no item or method associated with index `name`, `.nil` is returned.

Example 5.198. Directory class — at method

```ruby
say .environment-at("OBJECT")  /* Produces: "The Object class" */
```

5.3.9.7. empty

Returns the receiving Directory with all items removed. `empty` also removes all methods added using `setMethod`. 
5.3.9.8. entry

```
entry(name)
```

Returns the directory entry with index name (translated to uppercase). If there is no such entry, name returns the item for any method that setMethod supplied. If there is neither an entry nor a method for name, .nil is returned.

5.3.9.9. hasEntry

```
hasEntry(name)
```

Returns .true if the directory has an entry or a method for index name (translated to uppercase), or .false.

5.3.9.10. hasIndex

```
hasIndex(name)
```

Returns .true if the Directory contains any item associated with index name, or .false.

5.3.9.11. hasItem

```
hasItem(item)
```

Returns .true if the Directory contains the item at any index position or otherwise returns .false. Item equality is determined by using the == method of item.

5.3.9.12. index

```
index(item)
```

Returns the index of the specified item within the directory. If the target item appears at more than one index, the first located index will be returned. If the directory does not contain the specified item, .nil is returned. Item equality is determined by using the == method of item.

5.3.9.13. isEmpty

```
isEmpty
```

Returns .true if the directory is empty. Returns .false otherwise.

5.3.9.14. items

```
items
```

Returns the number of items in the collection.
5.3.9.15. makeArray

Returns a single-dimensional Array containing the index objects. The array indexes range from 1 to the number of items. The collection items appear in the array in an unspecified order. (The program should not rely on any order.)

5.3.9.16. put

Makes the object item a member item of the collection and associates it with index name. The new item replaces any existing item or method associated with index name.

5.3.9.17. remove

Removes and returns the member item with index name from the directory. If a method is associated with setMethod for index name, the method is removed and the result of running the method is returned. If there is no item or method with index name, .nil is returned.

5.3.9.18. removeEntry

Removes and returns the member item with index name (translated to uppercase) from the directory. If a method is associated with setMethod for index name, the method is removed and the result of running the method is returned. If there is no item or method with index name, .nil is returned.

See also setEntry, setMethod, hasEntry, and entry.

5.3.9.19. removeItem

Removes an item from the directory. If the target item exists at more than one index, the first located item is removed. The return value is the removed item. Item equality is determined by using the == method of item.

5.3.9.20. setEntry

Sets the member item with index name to entry.
Sets the directory entry with index name (translated to uppercase) to the object entry, replacing any existing entry or method for name. If you omit entry, this method removes any entry or method with this name.

5.3.9.21. setMethod

Associates index name (translated to uppercase) with method method. Thus, the object returns the result of running method when you access this entry. This occurs when you specify name on the [], at, entry, or remove method. This method replaces any existing item or method for name.

You can specify "UNKNOWN" as name. Doing so supplies a method that is run whenever an at or entry message specifies a name for which no item or method exists in the collection. This method's first argument is the specified directory index. This method has no effect on the action of any hasEntry, hasIndex, items, remove, or supplier message sent to the collection.

The method can be a string containing a method source line instead of a method object. Alternatively, an array of strings containing individual method lines can be passed. In either case, an equivalent method object is created.

If you omit method, setMethod removes the entry with the specified name.

5.3.9.22. supplier

Returns a Supplier object for the collection. The supplier allows you to iterate over the index/item pairs in the directory at the time the supplier was created. The supplier iterates the items in an unspecified order.

5.3.9.23. unknown

Runs either the entry or setEntry method, depending on whether messagename ends with an equal sign.

If messagename does not end with an equal sign, this method runs the entry method, passing messagename as its argument. The messageargs argument is ignored. The entry method is the return result.

If messagename does end with an equal sign, this method runs the setEntry method, passing the first part of messagename (up to, but not including, the final equal sign) as its first argument, and the first item in the array messageargs as its second argument. In this case, unknown returns no result.

5.3.9.24. unsetMethod

Removes the association between index name (translated to uppercase) and a method.
### 5.3.9.25. Examples

**Example 5.199. Directory class — examples**

```plaintext
/******************************************************************************/
/* A Phone Book Directory program                                            */
/* This program demonstrates use of the directory class.                      */
/******************************************************************************/

/* Define an UNKNOWN method that adds an abbreviation lookup feature.         */
/* Directories do not have to have an UNKNOWN method.                         */
book = .directory~new~~setMethod("UNKNOWN", .methods["UNKNOWN"])

book["ANN"] = "Ann B. ....... 555-6220"
book["ann"] = "Little annie . 555-1234"
book["JEFF"] = "Jeff G. ....... 555-5115"
book["MARK"] = "Mark C. ....... 555-5017"
book["MIKE"] = "Mike H. ....... 555-6123"
book~Rick = "Rick M. ....... 555-5110"          /* Same as book["RICK"] = ... */

Do i over book               /* Iterate over the collection */
  Say book[i]
end i

Say ""                       /* Index lookup is case sensitive... */
Say book-entry("Mike")      /* ENTRY method uppercases before lookup */
Say book["ANN"]            /* Exact match */
Say book~ann                 /* Message sends uppercase before lookup */
Say book["ann"]            /* Exact match with lowercase index */

Say ""
Say book["M"]             /* Uses UNKNOWN method for lookup */
Say book["Z"]
Exit

/******************************************************************************/
/* Define an unknown method to handle indexes not found.                      */
/* Check for abbreviations or indicate listing not found                      */
::Method unknown
  Parse arg at_index
  value = ""
  Do i over self
    If abbrev(i, at_index) then do
      If value <> "" then value = value", "
      value = value || self~at(i)
    end
  end i
  If value = "" then value = "No listing found for" at_index
  Return value

/* Define an unknown method that adds an abbreviation lookup feature.         */
/* Directories do not have to have an UNKNOWN method.                         */
******************************************************************************/
```

### 5.3.10. IdentityTable Class

An **IdentityTable** is a collection with indexes that can be any object. In an IdentityTable, each item is associated with a single index, and there can be only one item for each index. Index and item matches are made using an object identity comparison. That is, an index will only match if the same instance is used in the collection.

**Table 5.20. IdentityTable Class**

<table>
<thead>
<tr>
<th>Object</th>
<th>Methods inherited from the <strong>Object class</strong></th>
</tr>
</thead>
</table>
```
### IdentityTable Class

#### Methods inherited from the **Class class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### + Collection (Mixin Class)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Methods inherited from the **Collection class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>(Abstract Method) equivalent, put (Abstract Method)</td>
</tr>
<tr>
<td>[]=</td>
<td>(Abstract Method) hasIndex, subset</td>
</tr>
<tr>
<td>allIndexes</td>
<td>(Abstract Method) hasItem, supplier</td>
</tr>
<tr>
<td>allItems</td>
<td>(Abstract Method) index (Abstract Method), union</td>
</tr>
<tr>
<td>at</td>
<td>(Abstract Method) intersection, xor</td>
</tr>
<tr>
<td>difference</td>
<td>(Abstract Method) items, makeArray</td>
</tr>
<tr>
<td>disjoint</td>
<td>(Abstract Method)</td>
</tr>
</tbody>
</table>

#### + MapCollection (Mixin Class)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Methods inherited from the **MapCollection class of (Class Method)**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeArray</td>
<td>(Class Method)</td>
</tr>
<tr>
<td>putAll</td>
<td></td>
</tr>
</tbody>
</table>

### IdentityTable

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### new (Class Method)

```
new(size)
```

Returns an empty IdentityTable object.

If you specify `size`, a hint how large the new IdentityTable object is expected to grow, this is used to optimize the initial allocation.

`size` must be a non-negative whole number.

#### 5.3.10.2. []

```
[index]
```

Returns the item associated with `index`. This method is the same as the `at` method.

#### 5.3.10.3. []=

```
[index] = item
```

280
Adds item to the table at index index. This method is the same as the put method.

5.3.10.4. allIndexes

allIndexes

Returns an array of all indices contained in the table.

5.3.10.5. allItems

allItems

Returns an array of all items contained in the table.

5.3.10.6. at

at(index)

Returns the item associated with index index. Returns .nil if the IdentityTable has no item associated with index.

5.3.10.7. empty

empty

Returns the receiving IdentityTable with all items removed.

5.3.10.8. hasIndex

hasIndex(index)

Returns .true if the IdentityTable contains any item associated with index index, or .false.

5.3.10.9. hasItem

hasItem(item)

Returns .true if the IdentityTable contains the item at any index position or otherwise returns .false.

5.3.10.10. index

index(item)

Returns the index of the specified item within the table. If the target item appears at more than one index, the first located index will be returned. Returns .nil if the table does not contain the specified item.
5.3.10.11. isEmpty

Returns `.true` if the table is empty. Returns `.false` otherwise.

5.3.10.12. items

Returns the number of items in the collection.

5.3.10.13. makeArray

Returns a single-dimensional Array containing the index objects. The array indexes range from 1 to the number of items. The collection items appear in the array in an unspecified order.

5.3.10.14. put

Makes the object `item` a member item of the collection and associates it with index `index`. The new item replaces any existing item associated with index `index`.

5.3.10.15. remove

Returns and removes from a collection the member item with index `index`. Returns `.nil` if no item has index `index`.

5.3.10.16. removeItem

Removes an item from the table. If the target item exists at more than one index, the first located item is removed. The return value is the removed item.

5.3.10.17. supplier

Returns a `Supplier` object for the collection. The supplier allows you iterate over the index/item pairs contained in the table at the time the supplier was created. The supplier iterates over the items in an unspecified order.
5.3.11. List Class

A List is a non-sparse sequenced collection similar to the Array class to which you can add new items at any position in the sequence. The List creates a new index value whenever an item is added to the List and the associated index value remains valid for that item regardless of other additions or removals. Only indexes the List object generates are valid, i.e. the List is never a sparse list and the List object will not modify indexes for items in the list.

Table 5.21. List Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td>Class (Metaclass)</td>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>+ Collection (Mixin Class)</td>
<td>Methods inherited from the Collection class</td>
</tr>
<tr>
<td>+ OrderedCollection (Mixin Class)</td>
<td>Methods inherited from the OrderedCollection class</td>
</tr>
<tr>
<td>List</td>
<td>Methods inherited from the OrderedCollection class</td>
</tr>
<tr>
<td>new (Class Method)</td>
<td>of (Class Method)</td>
</tr>
</tbody>
</table>

5.3.11.1. new (Class Method)
5.3.11.2. of (Class Method)

Returns a newly created list containing the specified item objects in the order specified.

5.3.11.3. []

Returns the item located at index. This method is the same as the at method.

5.3.11.4. []=

Replaces the item at index with item. This method is the same as the put method.

5.3.11.5. allIndexes

Returns an array of all indexes contained in the list in the same order they are used in the list.

5.3.11.6. allItems

Returns an array of all items contained in the list in list iteration order.

5.3.11.7. append

Appends item to the end of the list, returning the index associated with item.
5.3.11.8. at

\texttt{at(index)}

Returns the item associated with index \textit{index}. Returns \texttt{nil} if the list has no item associated with \textit{index}.

5.3.11.9. delete

\texttt{delete(index)}

Returns and deletes the member item with the specified \textit{index} from the list. If there is no item with the specified \textit{index}, \texttt{nil} is returned and no item is deleted. All elements following the deleted item will be moved up in the list ordering, but the indexes associated with the moved items will not change. The size of the list will be reduced by one element. The delete method and the remove method produce the same result for the list class.

5.3.11.10. empty

\texttt{empty}

Returns the receiving List with all items removed.

5.3.11.11. first

\texttt{first}

Returns the index of the first item in the list or \texttt{nil} if the list is empty.

5.3.11.12. firstItem

\texttt{firstItem}

Returns the first item in the list or \texttt{nil} if the list is empty.

Example 5.200. List class — firstItem method

\begin{verbatim}
musketeers=.list-of("Porthos","Athos","Aramis") /* Creates list MUSKETEERS */ item=musketeers-firstItem /* Gives first item in list */ /* (Assigns "Porthos" to item) */
\end{verbatim}

5.3.11.13. hasIndex

\texttt{hasIndex(index)}

Returns \texttt{true} if the list contains any item associated with index \textit{index}, or \texttt{false}.
5.3.11.14. hasItem

Returns `.true` if the list contains the `item` at any index position or otherwise returns `.false`. Item equality is determined by using the `==` method of `item`.

5.3.11.15. index

Returns the index of the specified item within the list. If the target item appears at more than one index, the first located index will be returned. Returns `.nil` if the list does not contain the specified item. Item equality is determined by using the `==` method of `item`.

5.3.11.16. insert

Returns a list-supplied index for item `item`, which is added to the list. The inserted item follows an existing item with index `index` in the list ordering. If `index` is `.nil`, `item` becomes the first item in the list. If you omit `index`, the `item` becomes the last item in the list.

Inserting an item in the list at position `index` will cause the items in the list after position `index` to have their relative positions shifted by the list object. The index values for any items in the list are not modified by the insertion.

**Example 5.201. List class — insert method**

```clojure
musketeers=.list-of("Porthos","Athos","Aramis") ; Creates list MUSKETEERS
   ; consisting of: Porthos
   ;                Athos
   ;                Aramis
index=musketeers~first ; Gives index of first item
musketeers~insert("D'Artagnan",index) ; Adds D'Artagnan after Porthos
   ; List is now: Porthos
   ;              D'Artagnan
   ;              Athos
   ;              Aramis

/* Alternately, you could use */
musketeers~insert("D'Artagnan",.nil) ; Adds D'Artagnan before Porthos
   ; List is now:  D'Artagnan
   ;              Porthos
   ;              Athos
   ;              Aramis

/* Alternately, you could use */
musketeers~insert("D'Artagnan") ; Adds D'Artagnan after Aramis
   ; List is now:  Porthos
   ;              Athos
   ;              Aramis
   ;              D'Artagnan
```

286
5.3.11.17. isEmpty

```
isEmpty
```

Returns `.true` if the list is empty. Returns `.false` otherwise.

5.3.11.18. items

```
items
```

Returns the number of items in the list.

5.3.11.19. last

```
last
```

Returns the index of the last item in the list or `.nil` if the list is empty.

5.3.11.20. lastItem

```
lastItem
```

Returns the last item in the list or `.nil` if the list is empty.

5.3.11.21. makeArray

```
makeArray
```

Returns a single-dimensional Array containing the list collection items. The array indexes range from 1 to the number of items. The order in which the collection items appear in the array is the same as their sequence in the list collection.

5.3.11.22. next

```
next(index)
```

Returns the index of the item that follows the list item having index `index`. Returns `.nil` if `index` is the end of the list.

5.3.11.23. previous

```
previous(index)
```

Returns the index of the item that precedes the list item having index `index`. Returns `.nil` if `index` is the beginning of the list.
5.3.11.24. put

```
put(item, index)
```

Replaces any existing item associated with the specified index with the item. If index does not exist in the list, an error is raised.

5.3.11.25. remove

```
remove(index)
```

Returns and removes from a collection the member item with index index. If no item has index index, this method returns `nil` and removes no item.

Removing an item from the list at position index will shift the relative position of items after position index. The index values assigned to those items will not change.

5.3.11.26. removeItem

```
removeItem(item)
```

Removes an item from the list. If the target item exists at more than one index, the first located item is removed. The return value is the removed item. Item equality is determined by using the `==` method of item.

5.3.11.27. section

```
section(start, items)
```

Returns a new list (of the same class as the receiver) containing selected items from the receiver list. The first item in the new list is the item corresponding to index start in the receiver list. Subsequent items in the new list correspond to those in the receiver list (in the same sequence). If you specify the whole number items, the new list contains only this number of items (or the number of subsequent items in the receiver list, if this is less than items). If you do not specify items, the new list contains all subsequent items from the receiver list. The receiver list remains unchanged.

5.3.11.28. supplier

```
supplier
```

Returns a Supplier object for the list. The supplier allows you to iterate over the index/item pairs stored in the list at the time the supplier is created. The iteration is in the same order as the list sequence order.

5.3.12. Properties Class
A **Properties** object is a collection with unique indexes that are character strings representing names and items that are also restricted to character string values. Properties objects are useful for processing bundles of application option values.

### Table 5.22. Properties Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Object class</strong></td>
</tr>
<tr>
<td>Class (MetaClass)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
<tr>
<td>+ Collection (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>Collection class</strong></td>
</tr>
<tr>
<td>[] (Abstract Method) equivalent put (Abstract Method)</td>
</tr>
<tr>
<td>[]= (Abstract Method) hasIndex subset</td>
</tr>
<tr>
<td>allIndexes (Abstract Method) hasItem supplier</td>
</tr>
<tr>
<td>allItems (Abstract Method) index (Abstract Method) union</td>
</tr>
<tr>
<td>at (Abstract Method) intersection xor</td>
</tr>
<tr>
<td>difference items</td>
</tr>
<tr>
<td>disjoint makeArray</td>
</tr>
</tbody>
</table>

|+ MapCollection (Mixin Class)|
|Methods inherited from the **MapCollection class** of (Class Method) |
|makeArray |
|putAll |

<table>
<thead>
<tr>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Directory class</strong> new (Class Method)</td>
</tr>
<tr>
<td>[ ] hasIndex removeEntry</td>
</tr>
<tr>
<td>[ ]= hasItem removeItem</td>
</tr>
<tr>
<td>allIndexes index setEntry</td>
</tr>
<tr>
<td>allItems isEmpty setMethod</td>
</tr>
<tr>
<td>at items supplier</td>
</tr>
<tr>
<td>empty makeArray unknown</td>
</tr>
<tr>
<td>entry put unsetMethod</td>
</tr>
<tr>
<td>hasEntry remove</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Properties</strong> class</td>
</tr>
<tr>
<td>[ ]= load removeEntry</td>
</tr>
<tr>
<td>getLogical load setProperty</td>
</tr>
<tr>
<td>getProperty put setWhole</td>
</tr>
<tr>
<td>getWhole save</td>
</tr>
</tbody>
</table>

### 5.3.12.1. load (Class method)

```
Loads a set of properties from source and returns them as a new Properties object. The load source can be either the string name of a file, a File object, or a Stream object. Properties are read from source as individual lines using lineIn. Blank lines and lines with a Rexx line comment (“--”) as the first non-blank characters are ignored. Otherwise, the lines are assumed to be of the form “name=value” and are added to the receiver Properties value using name as the index for the value.

5.3.12.2. new (Class method)

```
new(size)
```

Returns an empty Properties object.

If you specify size, a hint how large the new Properties object is expected to grow, this is used to optimize the initial allocation.

size must be a non-negative whole number.

5.3.12.3. []=

```
{name} = item
```

Adds item using the index index. This method is the same as the put method.

5.3.12.4. getLogical

```
getLogical(name, default)
```

Returns the value of name as either .true or .false. The raw value of the name may be either the numeric values “0” or “1” or the string values “true” or “false”. Any other value will raise a syntax error. If the property name does not exist and default has been specified, the default value will be returned. If default has not been specified, a syntax error is raised for missing values.

5.3.12.5. getProperty

```
getProperty(name, default)
```

Returns the value of name. If property name does not exist and default has been specified, the default value will be returned. If default has not been specified, .nil is returned.

5.3.12.6. getWhole

```
getWhole(name, default)
```

290
Returns the value of name, validated as being a Rexx whole number. If property name does not exist and default has been specified, the default value will be returned. If default has not been specified, a syntax error is raised for missing values.

5.3.12.7. load

![load diagram]

Loads a set of properties into the receiving Properties object from source. The load source can be either the string name of a file, a File object, or a Stream object. Properties are read from the source as individual lines using lineIn. Blank lines and lines with a Rexx line comment ("--") as the first non-blank characters are ignored. Otherwise, the lines are assumed to be of the form "name=value" and are added to the receiver Properties value using name as the index for the value.

Properties loaded from source that have the same names as existing items will replace the current entries.

5.3.12.8. put

![put diagram]

Makes the object item a member item of the collection and associates it with index name. The item value must be a character string. The new item replaces any existing item or method associated with index name.

5.3.12.9. save

![save diagram]

Saves a set of properties into target. The save target can be either the string name of a file, a File object, or a Stream object. Properties are stored as individual lines using lineOut. The lines are written in the form "name=value". A saved Properties file can be reloaded using the Properties load method.

5.3.12.10. setLogical

![setLogical diagram]

Sets a logical value in the property bundle. The value argument must be either the numbers "0" or "1", or the logical values .true. or .false. The property value will be added with value converted in to the appropriate "true" or "false" string value.

5.3.12.11. setProperty

![setProperty diagram]

Sets a named property in the property bundle. The value argument must be a character string value.

5.3.12.12. setWhole
Queue Class

setWhole(name, value)

Sets a whole number value in the property bundle. The value argument must be a valid Rexx whole number.

5.3.13. Queue Class

A Queue is a non-sparse sequenced collection with whole-number indexes. The indexes specify the position of an item relative to the head (first item) of the queue. Adding or removing an item changes the association of an index to its queue item. You can add items at either the tail or the head of the queue.

Table 5.23. Queue Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>+ Collection (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the Collection class</td>
</tr>
<tr>
<td>+ OrderedCollection (Mixin Class)</td>
</tr>
<tr>
<td>Queue</td>
</tr>
</tbody>
</table>

292
5.3.13.1. new (Class Method)

```plaintext
new(size)
```

Returns a new empty Queue object.

If you specify `size`, a hint how large the new Queue object is expected to grow, this is used to optimize the initial allocation.

`size` must be a non-negative whole number.

5.3.13.2. of (Class Method)

```plaintext
of(item)
```

Returns a newly created queue containing the specified `item` objects in the order specified.

5.3.13.3. []

```plaintext
[index]
```

Returns the item located at index `index`. This method is the same as the `at` method.

5.3.13.4. []=

```plaintext
[index] = item
```

Replaces item at `index` with `item`. This method is the same as the `put` method.

5.3.13.5. allIndexes

```plaintext
allIndexes
```

Returns an array of all index values for the queue.

For the Queue class, the indices are integers from 1 to `items`.

5.3.13.6. allItems

```plaintext
allItems
```

Returns an array of all items contained in the queue, in queue order.
5.3.13.7. append

append(item)

Appends item to the end of the queue, returning the index of the inserted item.

5.3.13.8. at

at(index)

Returns the item associated with index index. Returns .nil if the Queue has no item associated with index.

5.3.13.9. delete

delete(index)

Returns and deletes the member item with the specified index from the queue. If there is no item with the specified index, .nil is returned and no item is deleted. All elements following the deleted item will be moved up in the queue ordering and the item indexes will be adjusted for the deletion. The size of the queue will be reduced by one element. The delete method and the remove method produce the same result for the queue class.

Example 5.202. Queue class — delete method

```
a = .queue-of("Fred", "Mike", "Rick", "David")
a~delete(2)  -- removes "Mike", resulting in the queue
-- ("Fred", "Rick", "David")
```

5.3.13.10. empty

empty

Returns the receiving Queue with all items removed.

5.3.13.11. first

first

Returns the index of the first item in the queue or .nil if the queue is empty. The index will always be 1 for non-empty queues.

5.3.13.12. firstItem

firstItem

Returns the first item in the queue or .nil if the queue is empty.
5.3.13.13. hasIndex

```
hasIndex(index)
```

Returns `.true.` if the queue contains any item associated with index `index`, or `.false.`.

5.3.13.14. hasItem

```
hasItem(item)
```

Returns `.true.` if the queue contains the `item` at any index position or otherwise returns `.false.`. Item equality is determined by using the `==` method of `item`.

5.3.13.15. index

```
index(item)
```

Returns the index of the specified item within the queue. If the target item appears at more than one index, the first located index will be returned. Returns `.nil` if the queue does not contain the specified item. Item equality is determined by using the `==` method of `item`.

5.3.13.16. insert

```
insert(item, index)
```

Returns a queue-supplied index for `item`, which is added to the queue. The inserted item follows any existing item with index `index` in the queue ordering. If `index` is `.nil`, `item` is inserted at the head of the queue. If you omit `index`, `item` becomes the last item in the queue.

Inserting an item in the queue at position `index` will cause the items in the queue after position `index` to have their indexes modified by the queue object.

Example 5.203. Queue class — insert method

```
musketeers=.queue-of("Porthos","Athos","Aramis") /* Creates queue MUSKETEERS       */ /* consisting of: Porthos         */ /* Athos           */ /* Aramis          */
index=musketeers~first                     /* Gives index of first item      */
musketeers~insert("D'Artagnan",index) /* Adds D'Artagnan after Porthos  */ /* List is now: Porthos          */ /* D'Artagnan      */ /* Athos           */ /* Aramis          */
/* Alternately, you could use */
musketeers~insert("D'Artagnan",.nil)  /* Adds D'Artagnan before Porthos */ /* List is now:  D'Artagnan     */ /* Porthos         */ /* Athos           */ /* Aramis          */
/* Alternately, you could use */
musketeers~insert("D'Artagnan")       /* Adds D'Artagnan after Aramis   */ /* List is now:  Porthos         */
```
5.3.13.17. isEmpty

Returns `.true` if the queue is empty. Returns `.false` otherwise.

5.3.13.18. items

Returns the number of items in the queue.

See also method `size`, which is a synonym for `items`.

5.3.13.19. last

Returns the index of the last item in the queue or `.nil` if the queue is empty.

5.3.13.20. lastItem

Returns the last item in the queue or `.nil` if the queue is empty.

5.3.13.21. makeArray

Returns a single-dimensional Array containing the receiver queue items. The array indexes range from 1 to the number of items. The order in which the queue items appear in the array is the same as their queuing order, with the head of the queue as index 1.

5.3.13.22. next

Returns the index of the item that follows the queue item having index `index` or returns `.nil` if the item having that index is last in the queue.

5.3.13.23. peek
Returns the item at the head of the queue. Returns `.nil` if the queue is empty. The collection remains unchanged.

### 5.3.13.24. previous

Returns the index of the item that precedes the queue item having index `index` or `.nil` if the item having that index is first in the queue.

### 5.3.13.25. pull

Returns and removes the item at the head of the queue. Returns `.nil` if the queue is empty.

### 5.3.13.26. push

Adds the object `item` to the head of the queue.

### 5.3.13.27. put

Replaces any existing item associated with the specified `index` with the new item. If the `index` does not exist in the queue, an error is raised.

### 5.3.13.28. queue

Adds the object `item` to the tail of the queue.

### 5.3.13.29. remove

Returns and removes from a collection the member item with index `index`. Returns `.nil` if no item has index `index`.

### 5.3.13.30. removeItem
Relation Class

Removes an item from the queue. If the target item exists at more than one index, the first located item is removed. The return value is the removed item. Item equality is determined by using the \texttt{==} method of \texttt{item}.

5.3.13.31. section

\begin{center}
\begin{tikzpicture}[node distance = 1.5cm, auto]
    
    
    
    

\end{tikzpicture}
\end{center}

Returns a new queue (of the same class as the receiver) containing selected items from the receiver. The first item in the new queue is the item corresponding to index \texttt{start} in the receiver. Subsequent items in the new queue correspond to those in the receiver (in the same sequence). If you specify the whole number \texttt{items}, the new queue contains only this number of items (or the number of subsequent items in the receiver, if this is less than \texttt{items}). If you do not specify \texttt{items}, the new queue contains all subsequent items from the receiver. The receiver queue remains unchanged.

5.3.13.32. size

\begin{center}
\begin{tikzpicture}[node distance = 1.5cm, auto]
    
    
    

\end{tikzpicture}
\end{center}

Returns the size, which is the number of items in the queue.

See also method \texttt{items} for which this method is a synonym.

5.3.13.33. supplier

\begin{center}
\begin{tikzpicture}[node distance = 1.5cm, auto]
    
    

\end{tikzpicture}
\end{center}

Returns a \texttt{Supplier} object for the queue. The supplier allows you to iterate over the index/item pair contained in the queue at the time the supplier was created. The supplier iterates the items in their queuing order, with the head of the queue first.

5.3.14. Relation Class

A Relation is a collection with indexes that can be any object. In a Relation, each item is associated with a single index, but there can be more than one item with the same index (unlike a Table, which can contain only one item for any index).

\begin{table}[h]
\centering
\caption{Relation Class}
\begin{tabular}{|l|l|l|}
\hline
\textbf{Object} & \textbf{Methods inherited from the Object class} & \\
\hline
\textbf{Class (Metaclass)} & & \\
\hline
\textbf{Methods inherited from the Class class} & & \\
\hline
\multicolumn{3}{|c|}{+ Collection (Mixin Class)} \\
\hline
\textbf{Methods inherited from the Collection class} & equivalent & put (Abstract Method) \\
\hline
\texttt{[]} (Abstract Method) & hasIndex & subset \\
\hline
\texttt{[]} = (Abstract Method) & hasItem & supplier \\
\hline
\texttt{allIndexes} (Abstract Method) & & \\
\hline
\end{tabular}
\end{table}
### 5.3.14.1. new (Class Method)

```
new(size)
```

Returns an empty Relation object.

If you specify `size`, a hint how large the new Relation object is expected to grow, this is used to optimize the initial allocation.

`size` must be a non-negative whole number.

### 5.3.14.2. []

```
[]
```

Returns an item associated with `index`. This method is the same as the `at` method.

### 5.3.14.3. []=

```
[]=item
```

Adds `item` to the relation associated with index `index`. This method is the same as the `put` method.

### 5.3.14.4. allAt
Relation Class

5.3.14.5. allIndex

Returns a single-dimensional Array containing all indexes for item item, in an unspecified order. Item equality is determined by using the == method of item.

5.3.14.6. allIndexes

Returns an array of all indexes contained in the Relation. The returned array will have one index for every item stored in the relation, including duplicates. To retrieve the indexes without duplicates, use the uniqueIndexes method.

5.3.14.7. allItems

Returns an array of all items contained in the relation.

5.3.14.8. at

Returns the item associated with index index. If the relation contains more than one item associated with index index, the item returned is unspecified. (The program should not rely on any particular item being returned.) Returns .nil if the relation has no item associated with index index. Index equality is determined by using the == method of index.

5.3.14.9. difference

Returns a new Relation containing only those items that the argument collection does not contain (with the same associated index). The argument can be a Collection object or any other object that supports a makeArray method.

5.3.14.10. empty

Returns the receiving Relation with all items removed.
5.3.14.11. hasIndex

Returns .true if the Relation contains any item associated with index index, or .false. Index equality is determined by using the == method of index.

5.3.14.12. hasItem

Returns .true if the relation contains the member item item, .false. If index is specified, hasItem will only return .true if the relation contains the pairing of item associated with index index. Item and index equality is determined by using the == method.

5.3.14.13. index

Returns the index for item item. If there is more than one index associated with item item, the one returned is not defined. Item equality is determined by using the == method of item.

5.3.14.14. intersection

Returns a new collection (of the same class as the receiver) containing only those items that are in both the receiver collection and the argument collection with the same associated index. The argument can be a Collection object or any other object that supports a makeArray method.

5.3.14.15. isEmpty

Returns .true if the relation is empty. Returns .false otherwise.

5.3.14.16. items

Returns the number of relation items with index index. If you specify no index, this method returns the total number of items associated with all indexes in the relation. Index equality is determined by using the == method of index.
5.3.14.17. makeArray

Returns a single-dimensional Array containing the index objects. The collection items appear in the array in an unspecified order.

5.3.14.18. put

Makes the object item a member item of the relation and associates it with index index. If the relation already contains any items with index index, this method adds a new member item item with the same index, without removing any existing member items.

5.3.14.19. remove

Returns and removes from a relation the member item with index index. If the relation contains more than one item associated with index index, the item returned and removed is unspecified. Returns .nil if no item has index index. Index equality is determined by using the == method of index.

5.3.14.20. removeAll

Returns and removes from a relation all member items with index index. All removed items are returned in an array containing each of the removed items. The order of the returned items is unspecified. If no items have the specified index, an empty array is returned. Index equality is determined by using the == method of index.

5.3.14.21. removeItem

If index is not specified, returns and removes from a relation the member item item. If the relation contains item associated with more than one index, the item returned and removed is unspecified.

If index is specified, returns and removes the member item item associated with index index. If item is the only member with index, then the index is also removed from the Relation.

Returns .nil if item is not a member item (associated with index index, if specified).

5.3.14.22. subset
Returns `true` if all items in the receiver Relation are also contained in the argument collection with the same associated index; returns `false` otherwise. The argument can be a `Collection` object or any other object that supports a `makeArray` method.

### 5.3.14.23. supplier

Returns a `Supplier` object for the relation. The supplier allows you to iterate over all index/item pairs in the relation at the time the supplier was created. The supplier enumerates the items in an unspecified order. If you specify `index`, the supplier contains all of the items with the specified index.

### 5.3.14.24. union

Returns a new collection containing all items from the receiver collection and the argument collection. The argument can be a `Collection` object or any other object that supports a `makeArray` method.

### 5.3.14.25. uniqueIndexes

Returns an array of all indexes contained in the Relation, with no duplicates.

### 5.3.14.26. xor

Returns a new collection of the same class as the receiver that contains all items from the receiver collection and the argument collection. All index-item pairs that appear in both collections are removed. The argument can be a `Collection` object or any other object that supports a `makeArray` method.

### 5.3.14.27. Examples

Example 5.204. Relation class — examples

```plaintext
/* Use a relation to express parent-child relationships */
family = .relation~new
family["Henry"] = "Peter" /* Peter is Henry's child */
family["Peter"] = "Bridget" /* Bridget is Peter's child */
family["Henry"] = "Jane" /* Jane is Henry's child */

/* Show all children of Henry recorded in the family relation */
henrys_kids = family~allAt("Henry")
Say "Here are all the listed children of Henry:
Do kid Over henrys_kids
```
Set Class

5.3.15. Set Class

A **Set** is a collection containing member items where the index is the same as the item (similar to a **Bag** collection). Any object can be placed in a Set. There can be only one occurrence of any object in a Set (unlike a **Bag** collection). Item equality is determined by using the `==` method.

Table 5.25. Set Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Object class</strong></td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
<tr>
<td>+ <strong>SetCollection (Mixin Class)</strong></td>
</tr>
<tr>
<td>Methods inherited from the <strong>SetCollection class</strong></td>
</tr>
<tr>
<td>(no class or instance methods)</td>
</tr>
<tr>
<td>+ <strong>Collection (Mixin Class)</strong></td>
</tr>
<tr>
<td>Methods inherited from the <strong>Collection class</strong></td>
</tr>
<tr>
<td>equivalent</td>
</tr>
<tr>
<td>hasIndex</td>
</tr>
<tr>
<td>hasItem</td>
</tr>
<tr>
<td>index (Abstract Method)</td>
</tr>
<tr>
<td>items</td>
</tr>
<tr>
<td>makeArray</td>
</tr>
<tr>
<td>put (Abstract Method)</td>
</tr>
<tr>
<td>subset</td>
</tr>
<tr>
<td>supplier</td>
</tr>
<tr>
<td>union</td>
</tr>
<tr>
<td>xor</td>
</tr>
<tr>
<td>+ <strong>MapCollection (Mixin Class)</strong></td>
</tr>
<tr>
<td>Methods inherited from the <strong>MapCollection class</strong></td>
</tr>
<tr>
<td>of (Class Method)</td>
</tr>
<tr>
<td>makeArray</td>
</tr>
<tr>
<td>putAll</td>
</tr>
</tbody>
</table>
### 5.3.15.1. of (Class Method)

```
of(item)
```

Returns a newly created set containing the specified `item` objects.

### 5.3.15.2. new (Class Method)

```
new(size)
```

Returns an empty Set object.

If you specify `size`, a hint how large the new Set object is expected to grow, this is used to optimize the initial allocation.

`size` must be a non-negative whole number.

### 5.3.15.3. []

```
[]
```

Returns the item associated with `index`. This method is the same as the `at` method.

### 5.3.15.4. []=  

```
[] = item
```

Adds an item to the Set. If specified, `index` must be the same object as `item`.

This method is the same as the `put` method.

### 5.3.15.5. allIndexes
Set Class

- **allIndexes**
  Returns an array of all indexes contained in the set.

- **allItems**
  Returns an array of all items contained in the set.

- **at(index)**
  Returns the item associated with index `index`. Returns `.nil` if the Set has no item associated with `index`.

- **empty**
  Returns the receiving Set with all items removed.

- **hasIndex(index)**
  Returns `.true` if the Relation contains any item associated with index `index`, or `.false`.

- **hasItem(value)**
  Returns `.true` if the Relation contains the specified item at any index location. Returns `.false` otherwise.

- **index(item)**
  Returns the index of the specified item within the set. If the target item appears at more than one index, the first located index will be returned. Returns `.nil` if the set does not contain the specified item. Item equality is determined by using the `==` method of `item`.

- **intersection(argument)**
Returns a new collection (of the same class as the receiver) containing only those items from the receiver whose indexes are in both the receiver collection and the argument collection. The argument can be a `Collection` object or any other object that supports a `makeArray` method. The argument must also allow all of the index values in the receiver collection.

### 5.3.15.13. `isEmpty`

`isEmpty` returns `.true` if the set is empty. Returns `.false` otherwise.

### 5.3.15.14. `items`

Returns the number of items in the collection.

### 5.3.15.15. `makeArray`

Returns a single-dimensional Array containing the index objects. The array indexes range from `1` to the number of items. The collection items appear in the array in an unspecified order.

### 5.3.15.16. `put`

Adds `item` to the Set. If there is an equivalent item in the Set, the exiting item will be replaced by the new instance. Item equality is determined by using the `==` method of `item`. If `index` is specified, it must be the same as `item`.

### 5.3.15.17. `putAll`

Returns the receiving Set with all items in `collection` added to it. The `collection` argument can be any object that supports a `supplier` method. Items from `collection` are added using the item values returned by the supplier. If duplicate items exist in `collection`, the last item provided by the supplier will overwrite previous items with the same index.

### 5.3.15.18. `remove`

Returns and removes the set item with index `index`. Returns `.nil` if no item has index `index`. 
5.3.15.19. removeItem

RemoveItem(item)

Removes an item from the set. If the target item exists at more than one index, the first located item is removed. The return value is the removed item. Item equality is determined by using the == method of item.

5.3.15.20. subset

subset(argument)

Returns .true if all items in the receiver collection are also contained in the argument collection; returns .false otherwise. The argument can be a Collection object or any other object that supports a makeArray method. The argument must also allow all of the index values in the receiver collection.

5.3.15.21. supplier

Returns a Supplier object for the collection. The supplier allows you iterate over the index/item pairs contained in the table at the time the supplier was created. The supplier iterates over the items in an unspecified order.

5.3.15.22. union

union(argument)

Returns a new Set contains all the items from the receiver collection and selected items from the argument collection. This method includes an item from argument in the new collection only if there is no item already in the in the receiver collection and the method has not already included a matching item. The order in which this method selects items in argument is unspecified. The argument can be a Collection object or any other object that supports a makeArray method. The argument must also allow all of the index values in the receiver collection.

5.3.15.23. xor

xor(argument)

Returns a new Set that contains all items from the receiver collection and the argument collection; all items that appear in both collections are removed. The argument can be a Collection object or any other object that supports a makeArray method. The argument must also allow all of the index values in the receiver collection.

5.3.16. Stem Class

A Stem object is a collection with unique indexes that are character strings.
### Table 5.26. Stem Class

<table>
<thead>
<tr>
<th>Methods inherited from the <strong>Object class</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class (Metaclass)</strong></td>
</tr>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
<tr>
<td>+ Collection (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>Collection class</strong></td>
</tr>
<tr>
<td>[] (Abstract Method)</td>
</tr>
<tr>
<td>[]= (Abstract Method)</td>
</tr>
<tr>
<td>allIndexes (Abstract Method)</td>
</tr>
<tr>
<td>allItems (Abstract Method)</td>
</tr>
<tr>
<td>at (Abstract Method)</td>
</tr>
<tr>
<td>difference</td>
</tr>
<tr>
<td>disjoint</td>
</tr>
<tr>
<td>+ MapCollection (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>MapCollection class of (Class Method)</strong></td>
</tr>
<tr>
<td>makeArray</td>
</tr>
<tr>
<td>putAll</td>
</tr>
<tr>
<td>Stem</td>
</tr>
<tr>
<td>new (Class Method)</td>
</tr>
<tr>
<td>[]</td>
</tr>
<tr>
<td>[]=</td>
</tr>
<tr>
<td>allIndexes</td>
</tr>
<tr>
<td>allItems</td>
</tr>
<tr>
<td>at</td>
</tr>
<tr>
<td>empty</td>
</tr>
<tr>
<td>hasIndex</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Stems are automatically created whenever a Rexx stem variable or Rexx compound variable is used. For example:

**Example 5.205. Stem class — examples**

```
a.1 = 2
```

creates a new stem collection with the name `A.` and assigns it to the Rexx variable `A.;` it also assigns the value 2 to entry 1 in the collection.

The value of an uninitialized stem index is the stem object NAME concatenated with the derived stem index.

**Example 5.206. Stem class — examples**

```
say a.[1,2] -- implicitly creates stem object with name "A."
-- displays "A.1.2"
a = .stem-new("B.")
```
In addition to the items explicitly assigned to the collection indexes, a value may be assigned to all possible stem indexes. The \[\]= method (with no index argument) will assign the target value to all possible stem indexes. Following assignment, a reference to any index will return the new value until another value is assigned or the index is dropped.

The [] method (with no index specified) will retrieve any globally assigned value. By default, this returns the stem NAME value.

In addition to the methods defined in the following, the Stem class removes the methods =, ==, \=, \\=, <>, and >= using the DEFINE method.

### 5.3.16.1. new (Class Method)

```
new(name)
```

Returns a new stem object. If you specify a string name, this value is used to create the derived name of compound variables. The default stem name is a null string (""").

### 5.3.16.2. []

```
[]
```

Returns the item associated with the specified indexes. Each index is an expression; use commas to separate the expressions. The Stem object concatenates the index string values, separating them with a period (.), to create a derived index. A null string (""") is used for any omitted expressions. The resulting string is the index of the target stem item. If the stem has no item associated with the specified final index, the stem default value is returned. If a default value has not been set, the stem name concatenated with the final index string is returned.

If you do not specify index, the stem default value is returned. If no default value has been assigned, the stem name is returned.

**Note**

You cannot use the [] method in a DROP or PROCEDURE instruction.

### 5.3.16.3. []=

```
[] =
```


Stem Class

Makes `value` a member item of the stem collection and associates it with the specified index. The final index is derived by concatenation of each of the `index` arguments together with a "." separator. If you specify no `index` arguments, a new default stem value is assigned. Assigning a new default value will re-initialize the stem and remove all existing assigned indexes.

5.3.16.4. allIndexes

```
| allIndexes |
```

Returns an array of all the stem tail names used in the stem.

5.3.16.5. allItems

```
| allItems |
```

Returns an array of all items contained in the stem.

5.3.16.6. at

```
| at( | tail ) |
```

Returns the item associated with the specified `tail`. `.nil` is returned if the stem has no item associated with the specified `tail`.

5.3.16.7. empty

```
| empty |
```

Returns the receiving Stem object with all items removed.

5.3.16.8. hasIndex

```
| hasIndex( | tail ) |
```

Returns `.true` if the Stem contains any item associated with a stem tail `tail`, or `.false`.

5.3.16.9. hasItem

```
| hasItem( | value ) |
```

Returns `.true` if the Stem contains the `value` at any tail position or otherwise returns `.false`. Item equality is determined by using the `==` method of `item`.

5.3.16.10. index

```
| index( | item ) |
```
Returns the index of the specified item within the stem. Returns `.nil` if the stem does not contain the specified item. Item equality is determined by using the `==` method of `item`.

### 5.3.16.11. `isEmpty`

Returns `.true` if the stem is empty. Returns `.false` otherwise.

### 5.3.16.12. `items`

Returns the number of items in the collection.

### 5.3.16.13. `makeArray`

Returns an array of all stem indexes that currently have an associated value. The items appear in the array in an unspecified order.

### 5.3.16.14. `put`

Replaces any existing item associated with the specified `tail` with the new item `item`.

### 5.3.16.15. `remove`

Returns and removes from the stem the member item with index `tail`. Returns `.nil` if no item has index `tail`.

### 5.3.16.16. `removeItem`

Removes an item from the stem. If the target item exists at more than one tail, the first located item is removed. Item equality is determined by using the `==` method of `item`. The return value is the removed item.

### 5.3.16.17. `request`


This method requests conversion to a specific class. All conversion requests except Array are forwarded to the stem's current stem default value. Returns the result of the Stem class makeArray method, if the requested class is Array. For all other classes, request forwards the message to the stem object's default value.

5.3.16.18. supplier

Returns a Supplier object for the stem. The supplier allows you to iterate though the index/item pairs contained in the Stem object at the time the supplier was created. The supplier iterates the items in an unspecified order.

5.3.16.19. toDirectory

Returns a Directory object for the stem. The directory will contain a name/value pair for each stem index with a directly assigned value.

5.3.16.20. unknown

Reissues or forwards all unknown messages to the stem's current default value. For additional information, see Section 4.2.6, "Defining an UNKNOWN Method".

5.3.17. StringTable Class

A StringTable is a MapCollection using unique character string indexes. The items of a StringTable can be any valid Rexx object.

See also Directory Class, a MapCollection similar to StringTable, but with additional methods setMethod and unsetMethod.

Table 5.27. StringTable Class

<table>
<thead>
<tr>
<th>Methods inherited from the Object class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>+ Collection (Mixin Class)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods inherited from the Collection class</th>
</tr>
</thead>
<tbody>
<tr>
<td>equivalent</td>
</tr>
<tr>
<td>hasIndex</td>
</tr>
<tr>
<td>hasItem</td>
</tr>
<tr>
<td>index (Abstract Method)</td>
</tr>
<tr>
<td>intersection</td>
</tr>
<tr>
<td>items</td>
</tr>
<tr>
<td>put (Abstract Method)</td>
</tr>
<tr>
<td>subset</td>
</tr>
<tr>
<td>supplier</td>
</tr>
<tr>
<td>union</td>
</tr>
<tr>
<td>xor</td>
</tr>
</tbody>
</table>
In addition to the standard put, [],=, at, and [] methods defined for Collections, StringTable provides access to items using methods. For example:

```
sTable = .StringTable~new
sTable~name = "Mike"   -- same as sTable~put("Mike", "NAME")
say sTable~name        -- same as say sTable["NAME"]
```

Note that StringTable does not provide methods setMethod and unsetMethod as the Directory class does.

### 5.3.17.1. new (Class Method)

Returns an empty StringTable object.

If you specify size, a hint how large the new StringTable object is expected to grow, this is used to optimize the initial allocation.

`size` must be a non-negative whole number.

### 5.3.17.2. []

Returns the item corresponding to `name`. This method is the same as the `at` method.

### 5.3.17.3. []=

RETURNS the item corresponding to `name`. This method is the same as the `at` method.
Adds or replaces the entry at index name. This method is the same as the `put` method.

**5.3.17.4. allIndexes**

Returns an array of all the StringTable indexes.

**5.3.17.5. allItems**

Returns an array of all items contained in the StringTable.

**5.3.17.6. at**

Returns the item associated with index name. If the StringTable has no item associated with index name, `nil` is returned.

Example 5.207. StringTable class — at method

```plaintext
say .environment~at("OBJECT") /* Produces: "The Object class" */
```

**5.3.17.7. empty**

Returns the receiving StringTable with all items removed.

**5.3.17.8. entry**

Returns the StringTable entry with index name (translated to uppercase). If there is no entry for name, `nil` is returned.

**5.3.17.9. hasEntry**

Returns `true` if the StringTable has an entry for index name (translated to uppercase), or `false`.

**5.3.17.10. hasIndex**
5.3.17.10. hasIndex

Returns \texttt{true} if the StringTable contains any item associated with index \texttt{name}, or \texttt{false}.

5.3.17.11. hasItem

Returns \texttt{true} if the StringTable contains the \texttt{item} at any index position or otherwise returns \texttt{false}. Item equality is determined by using the \texttt{==} method of \texttt{item}.

5.3.17.12. index

Returns the index of the specified item within the StringTable. If the target \texttt{item} appears at more than one index, the first located index will be returned. If the StringTable does not contain the specified \texttt{item}, \texttt{nil} is returned. Item equality is determined by using the \texttt{==} method of \texttt{item}.

5.3.17.13. isEmpty

Returns \texttt{true} if the StringTable is empty. Returns \texttt{false} otherwise.

5.3.17.14. items

Returns the number of items in the collection.

5.3.17.15. makeArray

Returns a single-dimensional Array containing the index objects. The array indexes range from \texttt{1} to the number of \texttt{items}. The collection \texttt{items} appear in the array in an unspecified order. (The program should not rely on any order.)

5.3.17.16. put

Makes the object \texttt{item} a member \texttt{item} of the collection and associates it with index \texttt{name}. The new \texttt{item} replaces any existing \texttt{item} or method associated with index \texttt{name}.

5.3.17.17. remove
5.3.17.18. removeEntry

Removes and returns the member item with index name (translated to uppercase) from the StringTable. If there is no item with index name, \texttt{.nil} is returned.

See also
• method \texttt{setEntry},
• method \texttt{hasEntry}, and
• method \texttt{entry}.

5.3.17.19. removeItem

Removes an item from the StringTable. If the target item exists at more than one index, the first located item is removed. The return value is the removed item. Item equality is determined by using the \texttt{\==} method of \texttt{item}.

5.3.17.20. setEntry

Sets the StringTable entry with index name (translated to uppercase) to the object entry, replacing any existing entry or method for name. If you omit entry, this method removes any entry or method with this name.

5.3.17.21. supplier

Returns a \texttt{Supplier} object for the collection. The supplier allows you to iterate over the index/item pairs in the StringTable at the time the supplier was created. The supplier iterates the items in an unspecified order.

5.3.17.22. unknown

Runs either the entry or setEntry method, depending on whether messagename ends with an equal sign.
If `messagename` does not end with an equal sign, this method runs the `entry` method, passing `messagename` as its argument. The `messageargs` argument is ignored. The `entry` method is the return result.

If `messagename` does end with an equal sign, this method runs the `setEntry` method, passing the first part of `messagename` (up to, but not including, the final equal sign) as its first argument, and the first item in the array `messageargs` as its second argument. In this case, `unknown` returns no result.

## 5.3.18. Table Class

A **Table** is a collection with indexes that can be any object. In a Table, each item is associated with a single index, and there can be only one item for each index (unlike a `Relation`, which can contain more than one item with the same index). Index equality is determined by using the `==` method.

### Table 5.28. Table Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Object class</strong></td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
<tr>
<td>+ Collection (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the <strong>Collection class</strong></td>
</tr>
<tr>
<td><code>[]</code> (Abstract Method)</td>
</tr>
<tr>
<td><code>[=]</code> (Abstract Method)</td>
</tr>
<tr>
<td>allIndexes (Abstract Method)</td>
</tr>
<tr>
<td>allItems (Abstract Method)</td>
</tr>
<tr>
<td>at (Abstract Method)</td>
</tr>
<tr>
<td>difference</td>
</tr>
<tr>
<td>disjoint</td>
</tr>
</tbody>
</table>

| + MapCollection (Mixin Class) |
| Methods inherited from the **MapCollection class** of (Class Method) |
| `makeArray` |
| `putAll` |

<table>
<thead>
<tr>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Table</strong></td>
</tr>
<tr>
<td><code>[]</code></td>
</tr>
<tr>
<td><code>[=]</code></td>
</tr>
<tr>
<td>allIndexes</td>
</tr>
<tr>
<td>allItems</td>
</tr>
<tr>
<td>at</td>
</tr>
<tr>
<td>empty</td>
</tr>
</tbody>
</table>

### 5.3.18.1. new (Class Method)

```ruby
def new(size)
  # Implementation of new method
end
```
Table Class

Returns an empty Table object.

If you specify size, a hint how large the new Table object is expected to grow, this is used to optimize the initial allocation.

size must be a non-negative whole number.

5.3.18.2. []

Returns the item associated with index. This method is the same as the at method.

5.3.18.3. []=

Adds item to the table at index index. This method is the same as the put method.

5.3.18.4. allIndexes

Returns an array of all indexes contained in the table.

5.3.18.5. allItems

Returns an array of all items contained in the table.

5.3.18.6. at

Returns the item associated with index index. Returns .nil if the Table has no item associated with index.

5.3.18.7. empty

Returns the receiving Table with all items removed.

5.3.18.8. hasIndex
Table Class

Returns `.true` if the Table contains any item associated with index `index`, or `.false`.

### 5.3.18.9. hasItem

![hasItem](image)

Returns `.true` if the Table contains the `value` at any index position or otherwise returns `.false`. Item equality is determined by using the `==` method of `item`.

### 5.3.18.10. index

![index](image)

Returns the index of the specified item within the table. If the target item appears at more than one index, the first located index will be returned. Returns `.nil` if the table does not contain the specified item. Item equality is determined by using the `==` method of `item`.

### 5.3.18.11. isEmpty

![isEmpty](image)

Returns `.true` if the table is empty. Returns `.false` otherwise.

### 5.3.18.12. items

![items](image)

Returns the number of items in the collection.

### 5.3.18.13. makeArray

![makeArray](image)

Returns a single-dimensional Array containing the index objects. The array indexes range from `1` to the number of items. The collection items appear in the array in an unspecified order.

### 5.3.18.14. put

![put](image)

Makes the object `item` a member item of the collection and associates it with index `index`. The new item replaces any existing items associated with index `index`.

### 5.3.18.15. remove

![remove](image)

320
Returns and removes the table item with index \textit{index}. Returns \texttt{.nil} if no item has index \textit{index}.

\subsection{5.3.18.16. \texttt{removeItem}}

\begin{syntax}
\texttt{removeItem(item)}
\end{syntax}

Removes an item from the table. If the target item exists at more than one index, the first located item is removed. The return value is the removed item. Item equality is determined by using the \texttt{==} method of \texttt{item}.

\subsection{5.3.18.17. \texttt{supplier}}

\begin{syntax}
\texttt{supplier}
\end{syntax}

Returns a \texttt{Supplier} object for the collection. The supplier allows you iterate over the index/item pairs contained in the table at the time the supplier was created. The supplier iterates over the items in an unspecified order.

\section{5.3.19. Sorting Ordered Collections}

Any ordered collection, such as non-sparse Arrays or Lists can have its elements placed into sorted order using the \texttt{sort} method. The simplest sort is sorting an array of strings.

\begin{example}{5.208. Array class — sorting}
myArray = .array-of("Zoe", "Fred", "Xavier", "Andy")
myArray~sort
do name over myArray
  say name
end
\end{example}

will display the names in the order "Andy", "Fred", "Xavier", "Zoe".

The \texttt{sort} method orders the strings by using the \texttt{compareTo} method of the \texttt{String} class. The \texttt{compareTo} method knows how to compare one string to another, and returns the values -1 (less than), 0 (equal), or 1 (greater than) to indicate the relative ordering of the two strings.

\subsection{5.3.19.1. Sorting non-strings}

Sorting is not limited to string values. Any object that inherits the \texttt{Comparable} MIXIN class and implements a \texttt{compareTo} method can be sorted. The \texttt{DateTime} class and \texttt{TimeSpan} class are examples of built-in Rexx classes that can be sorted. Any user-created class may also implement a \texttt{compareTo} method to enable sorting. For example, consider the following simple class:

\begin{example}{5.209. Non-string sorting}
::class Employee inherit Comparable
::attribute id
::attribute name
\end{example}
The Employee class implements its sort order using the employee identification number. When the sort method needs to compare two Employee instances, it will call the compareTo method on one of the instances, passing the second instance as an argument. The compareTo method tells the sort method which of the two instances should be first.

Example 5.210. Comparison during sorting

```plaintext
a = .array-new
a[1] = .Employee-new(654321, "Fred")
a[2] = .Employee-new(123456, "George")

a~sort

do employee over a
   say employee    -- sorted order is "George", "William", "Fred"
end
```

5.3.19.2. Sorting with more than one order

The String class compareTo method only implements a sort ordering for an ascending sort using a strict comparison. Frequently it is desirable to override a class-defined sort order or even to sort items that do not implement a compareTo method. To change the sorting criteria, use the sortWith method. The sortWith method takes a single argument, which is a Comparator object that implements a compare method. The compare method performs comparisons between pairs of items. Different comparators can be customized for different comparison purposes. For example, the Rexx language provides a DescendingComparator class that will sort items into descending order:

Example 5.211. Multi-order sorting

```plaintext
::CLASS 'DescendingComparator' MIXINCLASS Comparator
::METHOD compare
   use strict arg left, right
   return -left~compareTo(right)
```

The DescendingComparator merely inverts the result returned by the item compareTo method. Our previous example

Example 5.212. Descending compare sorting

```plaintext
myArray = .array-of("Zoe", "Fred", "Xavier", "Andy")
myArray~sortWith(.DescendingComparator~new)
```
Custom Comparators are simple to create for any sorting purpose. The only requirement is implementing a `compare` method that knows how to compare pairs of items in some particular manner. For example, to sort our Employee class by name instead of the default employee id, we can use the following simple comparator class:

```plaintext
::CLASS EmployeeNameSorter MIXINCLASS Comparator
::METHOD compare
  use strict arg left, right
  return left~name~compareTo(right~name)  -- do the comparison using the names
```

### 5.3.19.3. Builtin Comparators

Rexx includes a number of built-in Comparators for common sorting operations.

**Comparator Class**

Base comparator. The Comparator class just uses the `compareTo` method of the first argument to generate the result. Using `sortWith` and a Comparator instance is equivalent to using the `sort` method and no comparator.

**CaselessComparator Class**

Like the base comparator, but uses the `caselessCompareTo` method to determine order. The String class implements `caselessCompareTo`, so the CaselessComparator can be used to sort arrays of strings independent of case.

**ColumnComparator Class**

The ColumnComparator will sort string items using specific substrings within each string item. If sorting is performed on multiple column positions, the `stableSortWith` method is recommended to ensure the results of previous sort operations are retained.

**CaselessColumnComparator Class**

Like the ColumnComparator, but the substring comparisons are done independently of case.

**DescendingComparator Class**

The reverse of the Comparator class. The DescendingComparator can be used to sort items in descending order.

**CaselessDescendingComparator Class**

The reverse of the CaselessComparator class. The CaselessDescendingComparator can be used to sort items in descending order with comparisons done independently of case.

**InvertingComparator Class**

The InvertingComparator will invert the result returned by another Comparator instance. This comparator can be combined with another comparator instance to reverse the sort order.

**NumericComparator Class**

Performs comparisons of strings using numeric comparison rules. Use the NumericComparator to sort collections of numbers.
5.3.19.4. Stable and Unstable Sorts

Note

The current implementation is using the same stable sort algorithm for both the sort and the stableSort method of the Array class.

As such, below discussion of the difference between a stable and an unstable sort does not apply currently.

The default sorting algorithm is an unstable sort. In an unstable sort, items are not guaranteed to maintain their original positions if they compare equal during the sort. Consider the following simple example:

Example 5.214. Unstable sort

```
a = .array-of("Fred", "George", "FRED", "Mike", "fred")
a~sortwith(.caselesscomparator~new)
do name over a
   say name
end
```

This example displays the 3 occurrences of Fred in the order "Fred", "fred", "FRED", even though they compare equal using a caseless comparison.

The Array class implements a second sort algorithm that is available using the stableSort and stableSortWith methods. These methods use a Mergesort algorithm, which is less efficient than the default Quicksort and requires additional memory. The Mergesort is a stable algorithm that maintains the original relative ordering of equivalent items. Our example above, sorted with stableSortWith, would display "Fred", "FRED", "fred".

5.3.20. Concept of Set Operations

The following sections describe the concept of set operations to help you work with set operators, in particular if the receiver collection class differs from the argument collection class.

Rexx provides the following set-operator methods:

- difference
- intersection
- subset
- union
- xor

These methods are available to instances of the following collection classes:

- The OrderedCollections Array, List, Queue, and CircularQueue
Concept of Set Operations

- The MapCollections Directory, Stem, Table, IdentityTable, and Relation
- The SetCollections Set and Bag.

\[
\text{result} = \text{receiver}\text{-setoperator}(\text{argument})
\]

where:

- \text{receiver}
  
  is the collection object receiving the set-operator message.

- \text{setoperator}
  
  is the set-operator method used.

- \text{argument}
  
  is the argument collection supplier supplied to the method. It can be an instance of one of the Rexx collection classes or any object that implements a makearray method or supplier method, depending on class of \text{receiver}.

The result object is of the same class as the \text{receiver} collection object.

5.3.20.1. Principles of Operation

A set operation is performed by iterating over the elements of the \text{receiver} collection to compare each element of the \text{receiver} with each element of the \text{argument} collection. The element is defined as the tuple (\text{index}, \text{item}) (see Section 5.3.20.4, “Determining the Identity of an Item”). Depending on the set-operator method and the result of the comparison, an element of the receiver collection is, or is not, included in the resulting collection. A receiver collection that allows for duplicate elements can, depending on the set-operator method, also accept elements of the argument collection after they have been coerced to the type of the receiver collection.

The following examples are to help you understand the semantics of set operations. The collections are represented as a list of elements enclosed in curly brackets. The list elements are separated by a comma.

5.3.20.2. Set Operations on Collections without Duplicates

Assume that the example sets are \(A = \{a, b\}\) and \(B = \{b, c, d\}\). Except for subset, equivalent, and disjoint, the result of a set operation is another set. Using the collection \(A\) and \(B\), the different set operators produce the following:

**UNION operation**

All elements of \(A\) and \(B\) are united:

\[
A \text{ UNION } B = \{a, b, c, d\}
\]

**DIFFERENCE operation**

The resulting collection contains all elements of the first set except for those that also appear in the second set. The system iterates over the elements of the second set and removes them from the first set one by one.

\[
A \text{ DIFFERENCE } B = \{a\}
\]
Concept of Set Operations

B DIFFERENCE A = \{c, d\}

XOR operation
The resulting collection contains all elements of the first set that are not in the second set and all elements of the second set that are not in the first set:

A XOR B = \{a, c, d\}

INTERSECTION operation
The resulting collection contains all elements that appear in both sets:

A INTERSECTION B = \{b\}

SUBSET operation
Returns \texttt{true} if the first set contains only elements that also appear in the second set, otherwise it returns \texttt{false}:

A SUBSET B = \texttt{false}
B SUBSET A = \texttt{false}

EQUIVALENT operation
Returns \texttt{true} if the first set contains only elements that also appear in the second set and the two sets have the same number of elements, otherwise it returns \texttt{false}:

A EQUIVALENT B = \texttt{false}
B EQUIVALENT A = \texttt{false}

DISJOINT operation
Returns \texttt{true} if there are no elements that appear in both sets, otherwise it returns \texttt{false}:

A DISJOINT B = \texttt{false}
B DISJOINT A = \texttt{false}

5.3.20.3. Set-Like Operations on Collections with Duplicates

Assume that the example bags are A=\{a, b, b\} and B=\{b, b, c, c, d\}. Except for subset, equivalent, and disjoint, the result of any set-like operation is a collection, in this case a bag. Using the collections A and B, the different set-like operators produce the following:

UNION operation
All elements of A and B are united:

A UNION B = \{a, b, b, b, b, c, c, d\}

DIFFERENCE operation
The resulting collection contains all elements of the first bag except for those that also appear in the second bag. The system iterates over the elements of the second bag and removes them from the first bag one by one.

A DIFFERENCE B = \{a\}
B DIFFERENCE A = \{c, c, d\}
XOR operation
The resulting collection contains all elements of the first bag that are not in the second bag and all elements of the second bag that are not in the second bag:

\[ A \text{ XOR } B = \{a, c, c, d\} \]

INTERSECTION operation
The resulting collection contains all elements that appear in both bags:

\[ A \text{ INTERSECTION } B = \{b, b\} \]

SUBSET operation
Returns \texttt{true} if the first set contains only elements that also appear in the second set, otherwise it returns \texttt{false}:

\[ A \text{ SUBSET } B = \texttt{false} \]
\[ B \text{ SUBSET } A = \texttt{false} \]

EQUIVALENT operation
Returns \texttt{true} if the first set contains only elements that also appear in the second set and the two sets have the same number of elements, otherwise it returns \texttt{false}:

\[ A \text{ EQUIVALENT } B = \texttt{false} \]
\[ B \text{ EQUIVALENT } A = \texttt{false} \]

DISJOINT operation
Returns \texttt{true} if there are no elements that appear in both sets, otherwise it returns \texttt{false}:

\[ A \text{ DISJOINT } B = \texttt{false} \]
\[ B \text{ DISJOINT } A = \texttt{false} \]

### 5.3.20.4. Determining the Identity of an Item

Set operations require the definition of the identity of an element to determine whether a certain element exists in the receiver collection. The element of a collection is conceived as the tuple \((index, item)\). The \textit{index} is used as the identification tag associated with the item. Depending on the collection class, the index is an instance of a particular class, for example, the string class for a directory element, an integer for an array, or any arbitrary class for a relation. The Array class is an exception because it can be multi-dimensional having more than one index. However, as a collection, it is conceptionally linearized by the set operator.

For collection classes that require unique indexes, namely the Set, IdentityTable, Table, Directory, and Stem, an item is identified by its \textit{index}. For collections of collection classes that allow several items to have the same index, namely the Relation class, an item is identified by both its \textit{index} and its \textit{item}. For the Bag and the Set subclasses, where several items can have the same index but \textit{index} and \textit{item} must be identical, the item is identified by its \textit{index}. For Array, List, and Queue classes, the index is derived from an object's position within the collection's order. Items are identified using only \textit{item}.

When collections with different index semantics are used in set operations, the argument collection is coerced into a collection of the same type as the receiver, and the operation is then performed using the converted collection. The coercion process differs based on the types of both the receiver and the argument collection. According to this concept, an item of a collection is identified for the different \textit{receiver} categories as follows:
Map Collection

If argument is a MapCollection, then index values are used to determine membership, and items are inserted into the result using the index and item pairs.

If argument is an OrderedCollection or SetCollection, argument is converted into a MapCollection using the collection items as both index and item values. Since the argument collection may contain duplicate items, the converted collection is effectively a Relation instance.

For all other argument objects, the makearray method is used to obtain a set of values which are used as if argument was an OrderedCollection.

Ordered Collection and Set Collection

If argument is an instance of Collection, the matching set is obtained from the allItems method. For any other class of object, the makearray method is used. The hasItem method is used to perform the matching operations between the two collections.

Relation

If argument is a MapCollection, then index values are used to determine membership, and items are inserted into the result using the index and item pairs.

If argument is an OrderedCollection or SetCollection, argument is converted into a MapCollection using the collection items as both index and item values. Since the argument collection may contain duplicate items, the converted collection is effectively a Relation instance.

For all other argument objects, the makearray method is used to obtain a set of values which are used as if argument was an OrderedCollection. All tests for result membership are made using both the index and item values.

5.4. Utility Classes

This section describes

- MutableBuffer Class,
- File Class,
- Date-, time-, and timing-related classes DateTime, TimeSpan, Alarm, Ticker, and the notification classes AlarmNotification and MessageNotification,
- the synchronization classes EventSemaphore and MutexSemaphore,
- Comparable class and Orderable class,
- eight Comparator classes used for sorting (Caseless)Comparator, (Caseless)ColumnComparator, (Caseless)DescendingComparator, InvertingComparator, and NumericComparator, and
- other miscellaneous classes Buffer, Monitor, Pointer, RegularExpression, RexxContext, RexxInfo, RexxQueue, StackFrame, StreamSupplier, Supplier, Validate, VariableReference, and WeakReference.

5.4.1. Alarm Class

An Alarm object provides timing and notification capability by sending a notification message to a notification target at the trigger time.

An Alarm can be cancelled before it triggers. If cancelled, an Alarm will also send a notification message to the notification target.

<table>
<thead>
<tr>
<th>Table 5.29. Alarm Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
</tr>
</tbody>
</table>

328
Methods inherited from the \textit{Object class}

\textbf{Class (Metaclass)}

Methods inherited from the \textit{Class class}

\begin{tabular}{llll}
\textbf{Alarm} & \textbf{attachment} & \textbf{canceled/cancelled} & \textbf{scheduledTime} \\
\textbf{cancel} & \textbf{init} & & \textbf{triggered} \\
\end{tabular}

\section*{5.4.1.1. attachment}

\begin{itemize}
\item Returns the object that has been attached to the \texttt{Alarm} instance upon creation. Returns \texttt{.nil} if no object is attached.
\item See also method \texttt{init}.
\end{itemize}

\begin{example}
\textbf{Example 5.215. Alarm class — attachment method}
\begin{verbatim}
oneSecond = .Alarm~new(1, .Target~new, "a second has passed")
call SysSleep 1.5                      -- give Alarm time to trigger
oneSecond~cancel
::class Target inherit AlarmNotification
::method triggered                     -- called when Alarm triggers
use arg alarm
say alarm~attachment                 -- displays "a second has passed"
\end{verbatim}
\end{example}

\section*{5.4.1.2. cancel}

\begin{itemize}
\item Cancels the pending \texttt{Alarm} request represented by the receiver. Takes no action if the specified time has already been reached.
\end{itemize}

\section*{5.4.1.3. canceled/cancelled}

\begin{itemize}
\item Returns \texttt{.true} if the alarm was cancelled before triggering. Returns \texttt{.false} otherwise.
\item See also method \texttt{cancel}.
\end{itemize}

\begin{example}
\textbf{Example 5.216. Alarm class — canceled/cancelled method}
\begin{verbatim}
alarm = .Alarm-new(1, .message~new(.stdout, "SAY", "I", "Alarm went off"))
call SysSleep 0.5
alarm~cancel
\end{verbatim}
\end{example}
**5.4.1.4. init**

Sets up an `Alarm` for a future time `atime`. At this time, the `Alarm` sends message `triggered` to the specified notification `target`.

The `target` must be an object that implements the `AlarmNotification` interface. It must inherit from or be a subclass of the `AlarmNotification` class, or a `Message` object (as the `Message` class inherits from `AlarmNotification`). If `target` is a `Message` object, the `triggered` method of the `Message` class will respond by simply sending the specified message.

The `atime` can be a `DateTime`, a `TimeSpan`, or a `String` object. If it is
• a `DateTime` object, it specifies the time when the alarm will be triggered. The specified time must be in the future.
• a `TimeSpan`, the `Alarm` will be set to the current time plus the specified time span. The time span must not be a negative interval.
• a `String`, you can specify this as a date and time (`hh:mm:ss`) or as a number of seconds starting at the present time. If you use the date and time format, you can specify a date in the default format (`dd MMM yyyy`) after the time with a single blank separating the time and date. Leading and trailing whitespace characters are not allowed in the `atime`. If you do not specify a date, the `Alarm` uses the first future occurrence of the specified time.

If specified, `attachment` can be an arbitrary object that will be attached to the alarm instance, and can later be retrieved in the event handler. See method `attachment`.

You can use the `cancel` method to cancel a pending alarm. If cancelled, the alarm sends message `cancel` to the specified notification `target`.

The following code sets up an alarm at 5:10 p.m. on December 15, 2017. (Assume today's date/time is prior to December 15, 2017.)

```
/* Alarm Examples */

PersonalMessage = .MyMessageClass~new("Call the Bank")
msg = .Message~new(PersonalMessage, "RemindMe")
time = .DateTime~fromIsoDate("2017-12-15T17:10:00.000000")
a = .Alarm~new(time, msg)
exit

::class MyMessageClass public
::method init
  expose inmsg
  use arg inmsg
::method RemindMe
  expose inmsg
  say "It is now" time("C")". Please" inmsg
```
/* On the specified date and time, displays the following message: */
/* "It is now 5:10pm. Please Call the Bank" */

For the following example, the user uses the same code as in the preceding example to define msg, a message object to run at the specified time. The following code sets up an alarm to run the msg message object in 30 seconds from the current time:

```
Example 5.218. Alarm class

a = .Alarm~new(30, msg)
```

### 5.4.1.5. scheduledTime

Returns a `DateTime` object representing the time for which the alarm has initially been scheduled.

### 5.4.1.6. triggered

Returns `.true` if the alarm has triggered. Returns `.false` if the alarm has been cancelled or hasn't yet triggered.

### 5.4.2. AlarmNotification Class

The `AlarmNotification` class implements the notification interface for the `Alarm` and the `Ticker` class.

For any `Alarm` or `Ticker` object, notifications of alarm triggering and alarm cancellation are sent to the notification target specified when creating the `Alarm` or `Ticker` instance. The notification target must implement this alarm notification interface. Upon alarm triggering a `triggered` message is sent to the notification target, and upon alarm cancellation a `cancel` message is sent to the notification target.

This class is defined as a MIXIN class.

#### Table 5.30. AlarmNotification Class

<table>
<thead>
<tr>
<th>Object</th>
<th>Methods inherited from the <code>Object class</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Class (Metaclass)</td>
<td>Methods inherited from the <code>Class class</code></td>
</tr>
<tr>
<td><code>cancel</code></td>
<td><code>AlarmNotification (Mixin Class)</code></td>
</tr>
<tr>
<td><code>triggered</code> (Abstract method)</td>
<td></td>
</tr>
</tbody>
</table>

### 5.4.2.1. cancel

```
cancel(source)
```
Whenever an Alarm or Ticker is cancelled, this method of the alarm notification target will be called. The Alarm or Ticker object that was cancelled will be provided as argument source.

It is defined as a no-op and doesn't necessarily have to be implemented by an inheriting class.

### 5.4.2.2. triggered (Abstract method)

Whenever an Alarm or Ticker triggers, this method of the alarm notification target will be called. The Alarm or Ticker object which triggered will be provided as argument source.

It is an abstract method and must be implemented by an inheriting class.

For an example see Alarm class attachment method example.

### 5.4.3. Buffer Class

A Buffer instance is a Rexx interpreter managed block of storage. This class is designed primarily for writing methods and functions in native code and can only be created using the native code application programming interfaces. The new (Class Method) method will raise an error if invoked.

<table>
<thead>
<tr>
<th>Table 5.31. Buffer Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
</tr>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td><strong>Class (Metaclass)</strong></td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td><strong>Buffer</strong></td>
</tr>
<tr>
<td>new (Class Method)</td>
</tr>
</tbody>
</table>

### 5.4.3.1. new (Class Method)

Creating Buffer object instances directly from Rexx code is not supported. Method new will raise an error if invoked.

### 5.4.4. Comparable Class

This class is defined as a MIXIN class.

<table>
<thead>
<tr>
<th>Table 5.32. Comparable Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
</tr>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td><strong>Class (Metaclass)</strong></td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td><strong>Comparable (Mixin Class)</strong></td>
</tr>
</tbody>
</table>
5.4.4.1. compareTo

This method compares the receiving object to the object supplied in the `other` argument. This is a default implementation which compares two items based on their `identityHash`.

```
compareTo( other )
```

This method returns \(-1\) if the `other` is larger than the receiving object, \(0\) if the two objects are equal, and \(1\) if `other` is smaller than the receiving object.

Note

Classes inheriting from `Comparable` (like built-in classes `File`, `DateTime`, or `TimeSpan`) are strongly encouraged to forward to this default implementation for comparison cases not covered by their class-specific `compareTo` implementation.

5.4.5. Comparator Classes

This section describes eight Comparator classes used for sorting, `(Caseless)Comparator`, `(Caseless)ColumnComparator`, `(Caseless)DescendingComparator`, `InvertingComparator`, and `NumericComparator`.

5.4.5.1. Comparator Class

The `Comparator` class is the base class for implementing `Comparator` objects that can be used with the `sortWith` or `stableSortWith` methods. The `compare` method implements some form of comparison that determines the relative ordering of two objects. Many `Comparator` implementations are specific to particular object types.

It is defined as a MIXIN class.

Table 5.33. Comparator Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Object</code></td>
<td>Methods inherited from the <code>Object class</code></td>
</tr>
<tr>
<td><code>Class (Metaclass)</code></td>
<td>Methods inherited from the <code>Class class</code></td>
</tr>
<tr>
<td><code>Comparator (Mixin Class)</code></td>
<td></td>
</tr>
<tr>
<td><code>compare</code></td>
<td></td>
</tr>
</tbody>
</table>

5.4.5.1.1. compare

```
compare( first, second )
```

Returns \(-1\) if `second` is larger than `first`, \(0\) if the two objects are equal, and \(1\) if `second` is smaller than `first`. 
Comparator Classes

The default Comparator compares method assumes that first is an object that implements the Comparable compareTo method. Subclasses may override this to implement more specific comparisons.

Example 5.219. Comparator class

```plaintext
wine = .Array-of("Strawberries", "cherries", "angel's kiss")
wine-sortWith(.Comparator-new)         -- Strawberries, angel's kiss, cherries
```

5.4.5.2. CaselessComparator Class

The CaselessComparator class performs caseless orderings of string objects.

It is defined as a MIXIN class.

Table 5.34. CaselessComparator Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>+ Comparator (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the Comparator class</td>
</tr>
<tr>
<td>compare</td>
</tr>
<tr>
<td>CaselessComparator (Mixin Class)</td>
</tr>
<tr>
<td>compare</td>
</tr>
</tbody>
</table>

5.4.5.2.1. compare

```plaintext
compare(first, second)
```

Returns -1 if second is larger than first, 0 if the two objects are equal, and 1 if second is smaller than first.

The two strings are compared using a caseless comparison.

Example 5.220. CaselessComparator class

```plaintext
wine = .Array-of("Strawberries", "cherries", "angel's kiss")
wine-sortWith(.CaselessComparator-new) -- angel's kiss, cherries, Strawberries
```

5.4.5.3. ColumnComparator Class

The ColumnComparator class performs orderings based on specific substrings of string objects.

It is defined as a MIXIN class.

Table 5.35. ColumnComparator Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
</table>
Comparator Classes

Methods inherited from the \textit{Object class}

\textbf{Class (Metaclass)}

Methods inherited from the \textit{Class class}

+ \textbf{Comparator (Mixin Class)}

Methods inherited from the \textbf{Comparator class}

\texttt{compare}

\textbf{ColumnComparator (Mixin Class)}

\texttt{compare, init}

\subsection*{5.4.5.3.1. compare}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{compare.png}
\end{figure}

Returns -1 if \textit{second} is larger than \textit{first}, 0 if the two objects are equal, and 1 if \textit{second} is smaller than \textit{first}.

Only the defined columns of the strings are compared.

\subsection*{5.4.5.3.2. init}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{init.png}
\end{figure}

Initializes a \textbf{ColumnComparator} to sort strings starting at position \textit{start} for \textit{length} characters.

\begin{example}[ColumnComparator class]
\begin{verbatim}
wine = .Array-of("1. Strawberries", "2. cherries", "3. angel's kiss")
wine-sortWith(.ColumnComparator-new(3, 100))
-- 1. Strawberries, 3. angel's kiss, 2. cherries
\end{verbatim}
\end{example}

\subsection*{5.4.5.4. CaselessColumnComparator Class}

The \textbf{CaselessColumnComparator} class performs caseless orderings of specific substrings of string objects.

It is defined as a MIXIN class.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Object} & \textbf{Methods} & \textbf{Class (Metaclass)} \\
\hline
\textbf{Methods inherited from the Object class} & \textbf{Class (Metaclass)} & \textbf{+ Comparator (Mixin Class)} \\
\hline
\textbf{Methods inherited from the Class class} & \textbf{+ Comparator (Mixin Class)} & \textbf{Methods inherited from the Comparator class} \\
\hline
\texttt{compare} & \textbf{CaselessColumnComparator (Mixin Class)} & \\
\hline
\end{tabular}
\end{table}

335
5.4.5.4.1. compare

Returns -1 if second is larger than first, 0 if the two objects are equal, and 1 if second is smaller than first.

Only the defined columns of the strings are compared, using a caseless comparison.

5.4.5.4.2. init

Initializes a CaselessColumnComparator to sort strings starting at position start for length characters.

Example 5.222. CaselessColumnComparator class

wine = .Array-of("1. Strawberries", "2. cherries", "3. angel's kiss")
wine-sortWith(.CaselessColumnComparator-new(3, 100))
   -- 3. angel's kiss, 2. cherries, 1. Strawberries

5.4.5.5. DescendingComparator Class

The DescendingComparator class performs string sort orderings in descending order. This is the inverse of a Comparator sort order.

This class is defined as a MIXIN class.

Table 5.37. DescendingComparator Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>+ Comparator (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the Comparator class</td>
</tr>
<tr>
<td>compare</td>
</tr>
<tr>
<td>DescendingComparator (Mixin Class)</td>
</tr>
</tbody>
</table>

5.4.5.5.1. compare
Comparator Classes

Returns 1 if second is larger than first, 0 if the two objects are equal, and -1 if second is smaller than first, resulting in a descending sort sequence.

The DescendingComparator assumes the first object implements the Comparable compareTo method.

Example 5.223. DescendingComparator class

```
wine = .Array~of("Strawberries", "cherries", "angel's kiss")
wine~sortWith(.DescendingComparator~new)    -- cherries, angel's kiss, Strawberries
```

### 5.4.5.6. CaselessDescendingComparator Class

The CaselessDescendingComparator class performs caseless string sort orderings in descending order. This is the inverse of a CaselessComparator sort order.

This class is defined as a MIXIN class.

<table>
<thead>
<tr>
<th>Table 5.38. CaselessDescendingComparator Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
</tr>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>+ Comparator (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the Comparator class</td>
</tr>
<tr>
<td>compare</td>
</tr>
<tr>
<td>CaselessDescendingComparator (Mixin Class)</td>
</tr>
</tbody>
</table>

5.4.5.6.1. compare

```
compare( first, second )
```

Returns 1 if second is larger than first, 0 if the two objects are equal, and -1 if second is smaller than first, resulting in a descending sort sequence.

The two strings are compared using a caseless comparison.

Example 5.224. CaselessDescendingComparator class

```
wine = .Array~of("Strawberries", "cherries", "angel's kiss")
wine~sortWith(.CaselessDescendingComparator~new) -- Strawberries, cherries, angel's kiss
```

### 5.4.5.7. InvertingComparator Class

The InvertingComparator class inverts the comparison results of another Comparator object to reverse the resulting sort order.

This class is defined as a MIXIN class.
Table 5.39. InvertingComparator Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from</td>
</tr>
<tr>
<td>the Object class</td>
</tr>
<tr>
<td>+ Comparator (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from</td>
</tr>
<tr>
<td>the Class class</td>
</tr>
<tr>
<td>Methods inherited from</td>
</tr>
<tr>
<td>the Comparator class</td>
</tr>
<tr>
<td>InvertingComparator (Mixin Class)</td>
</tr>
</tbody>
</table>

5.4.5.7.1. compare

The InvertingComparator will invert the ordering returned by the comparator it was initialized with.

5.4.5.7.2. init

Initializes an InvertingComparator to sort strings using an inversion of the result from the comparator compare method.

Example 5.225. InvertingComparator class

```perl
wine = Array~of("1. Strawberries", "2. cherries", "3. angel's kiss")
wine~sortWith(.InvertingComparator~new(.LengthComparator~new))
-- 3. angel's kiss, 1. Strawberries, 2. cherries
```

5.4.5.8. NumericComparator Class

The NumericComparator class compares strings using numeric comparison rules rather than string comparison rules.

It is defined as a MIXIN class.

Table 5.40. NumericComparator Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
</table>

338
5.4.5.8.1. compare

\[ \text{compare(} \text{first, second}\text{)} \]

Returns -1 if second is larger than first, 0 if the two objects are equal, and 1 if second is smaller than first.

Comparisons are performed using numeric comparison rules, so the collection strings must be valid numeric values.

5.4.5.8.2. init

\[ \text{init(} \text{digits}\text{)} \]

Initializes a NumericComparator to sort strings using numeric comparison rules using the Comparator compare method. If digits is specified, the comparisons will be performed using the provided precision. The default precision is 9.

Example 5.226. NumericComparator class

```plaintext
primes = .Array~of(23, 19, 17, 13, 11, 7, 5, 3, 2)
primes~sortWith(.NumericComparator~new)  --  2, 3, 5, 7, 11, 13, 17, 19, 23
```

5.4.6. DateTime Class

A DateTime instance represents a timestamp between 1 January 0001 at 00:00.000000 and 31 December 9999 at 23:59:59.999999. It has methods to allow formatting a date or time in various formats, as well as allowing arithmetic operations between timestamps.

**Note**

DateTime does not support leap seconds like December 31, 2016 at 23:59:60 UTC.
Table 5.41. DateTime Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Object class</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class (Metaclass)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>+ Orderable (Mixin Class)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Orderable class</strong></td>
</tr>
</tbody>
</table>

*Comparison Methods* $= == < <= << <= <> > >= >== |== |== |<= |<= |<= |== |<= |== |<= |== |

| compareTo |

<table>
<thead>
<tr>
<th>+ Comparable (Mixin Class)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Comparable class</strong></td>
</tr>
</tbody>
</table>

| compareTo |

<table>
<thead>
<tr>
<th>DateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>new (Inherited Class Method)</td>
</tr>
<tr>
<td>fromBaseDate (Class Method)</td>
</tr>
<tr>
<td>fromCivilTime (Class Method)</td>
</tr>
<tr>
<td>fromEuropeanDate (Class Method)</td>
</tr>
<tr>
<td>fromIsoDate (Class Method)</td>
</tr>
<tr>
<td>fromLongTime (Class Method)</td>
</tr>
<tr>
<td>fromNormalDate (Class Method)</td>
</tr>
<tr>
<td>fromNormalTime (Class Method)</td>
</tr>
<tr>
<td>fromOrderedDate (Class Method)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arithmetic Methods + -</th>
</tr>
</thead>
<tbody>
<tr>
<td>addDays</td>
</tr>
<tr>
<td>addHours</td>
</tr>
<tr>
<td>addMicroseconds</td>
</tr>
<tr>
<td>addMinutes</td>
</tr>
<tr>
<td>addSeconds</td>
</tr>
<tr>
<td>addWeeks</td>
</tr>
<tr>
<td>addYears</td>
</tr>
<tr>
<td>baseDate</td>
</tr>
<tr>
<td>civilTime</td>
</tr>
<tr>
<td>compareTo</td>
</tr>
<tr>
<td>date</td>
</tr>
<tr>
<td>day</td>
</tr>
<tr>
<td>dayMicroseconds</td>
</tr>
<tr>
<td>dayMinutes</td>
</tr>
<tr>
<td>dayName</td>
</tr>
<tr>
<td>daySeconds</td>
</tr>
<tr>
<td>daysInMonth</td>
</tr>
<tr>
<td>daysInYear</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>elasped</th>
</tr>
</thead>
<tbody>
<tr>
<td>europeanDate</td>
</tr>
<tr>
<td>fullDate</td>
</tr>
<tr>
<td>hashCode</td>
</tr>
<tr>
<td>hours</td>
</tr>
<tr>
<td>init</td>
</tr>
<tr>
<td>isLeapYear</td>
</tr>
<tr>
<td>isoDate</td>
</tr>
<tr>
<td>isoTime</td>
</tr>
<tr>
<td>makeString</td>
</tr>
<tr>
<td>minutes</td>
</tr>
<tr>
<td>microseconds</td>
</tr>
<tr>
<td>month</td>
</tr>
<tr>
<td>monthName</td>
</tr>
<tr>
<td>monthYear</td>
</tr>
<tr>
<td>normalDate</td>
</tr>
<tr>
<td>normalTime</td>
</tr>
<tr>
<td>offset</td>
</tr>
<tr>
<td>orderedDate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ordinalDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>seconds</td>
</tr>
<tr>
<td>standardDate</td>
</tr>
<tr>
<td>string</td>
</tr>
<tr>
<td>ticks</td>
</tr>
<tr>
<td>timeOfDay</td>
</tr>
<tr>
<td>toLocalTime</td>
</tr>
<tr>
<td>toTimeZone</td>
</tr>
<tr>
<td>toUtcTime</td>
</tr>
<tr>
<td>usaDate</td>
</tr>
<tr>
<td>utcDate</td>
</tr>
<tr>
<td>utcIsoDate</td>
</tr>
<tr>
<td>weekDay</td>
</tr>
<tr>
<td>weekNumber</td>
</tr>
<tr>
<td>weekNumberDate</td>
</tr>
<tr>
<td>weekNumberYear</td>
</tr>
<tr>
<td>weekInYear</td>
</tr>
<tr>
<td>year</td>
</tr>
<tr>
<td>yearDay</td>
</tr>
</tbody>
</table>

5.4.6.1. minDate (Class Method)

**minDate**

Returns a **DateTime** instance representing the minimum supported Rexx date, 1 January 0001 at 00:00:00.000000.
5.4.6.2. maxDate (Class Method)

Returns a DateTime instance representing the maximum supported Rexx date, 31 December 9999 at 23:59:59.999999.

5.4.6.3. today (Class Method)

Returns a DateTime instance for the current day, with a time value of 00:00:00.000000.

If specified, offset is the offset from UTC, in minutes. The offset must be a valid whole number between -900 and 900 or an equivalent TimeSpan instance. The default offset is the current system timezone offset.

5.4.6.4. fromBaseDate (Class Method)

Returns a new DateTime object, created from a string in the format returned by the Base option of the DATE built-in function (ddddddd). The time component will be set to 00:00:00.000000.

If specified, offset is the offset from UTC, in minutes. The offset must be a valid whole number between -900 and 900 or an equivalent TimeSpan instance. The default offset is the current system timezone offset.

5.4.6.5. fromEuropeanDate (Class Method)

Returns a new DateTime object, created from a string in the format returned by the European option of the DATE built-in function (dd/mm/yy). The time component will be set to 00:00:00.000000.

If specified, separator identifies the field separator character used in the string. The separator must be a single character or the null string (""). A slash ("/") is the default separator. The time component will be set to 00:00:00.000000.

If specified, offset is the offset from UTC, in minutes. The offset must be a valid whole number between -900 and 900 or an equivalent TimeSpan instance. The default offset is the current system timezone offset.

5.4.6.6. fromNormalDate (Class Method)
5.4.6.7. fromOrderedDate (Class Method)

Returns a new DateTime object, created from a string in the format returned by the Ordered option of the DATE built-in function (yy/mm/dd). The time component will be set to 00:00:00.000000.

If specified, separator identifies the field separator character used in the string. The separator must be a single character or the null string ("'"). A slash ("/") is the default separator. The time component will be set to 00:00:00.000000.

If specified, offset is the offset from UTC, in minutes. The offset must be a valid whole number between -900 and 900 or an equivalent TimeSpan instance. The default offset is the current system timezone offset.

5.4.6.8. fromOrdinalDate (Class Method)

Returns a new DateTime object, created from a string in ISO ordinal date format. The time component will be set to 00:00:00.000000.

Both the basic format yyyyddd and the extended format yyyy-ddd are accepted, where ddd is the ordinal number of a day within year yyyy. The allowed range for ddd is 1 through the number of days in year yyyy, which is either 365 or 366. Leading zeros are required for both yyyy and ddd.

If specified, offset is the offset from UTC, in minutes. The offset must be a valid whole number between -900 and 900 or an equivalent TimeSpan instance. The default offset is the current system timezone offset.

See also method ordinalDate.

Example 5.227. DateTime class — fromOrdinalDate method

```
say .DateTime~fromOrdinalDate("2020-041")    -- 2020-02-10T00:00:00.000000
```
5.4.6.9. fromStandardDate (Class Method)

Returns a new DateTime object, created from a string in the format returned by the Standard option of the DATE built-in function (yyyyymmdd). The time component will be set to 00:00:00.000000.

If specified, separator identifies the field separator character used in the string. The separator must be a single character or the null string (""). A null string (""") is the default separator.

If specified, offset is the offset from UTC, in minutes. The offset must be a valid whole number between -900 and 900 or an equivalent TimeSpan instance. The default offset is the current system timezone offset.

5.4.6.10. fromUsaDate (Class Method)

Returns a new DateTime object, created from a string in the format returned by the Usa option of the DATE built-in function (mm/dd/yy). The time component will be set to 00:00:00.000000.

If specified, separator identifies the field separator character used in the string. The separator must be a single character or the null string (""). A slash ("/") is the default separator.

If specified, offset is the offset from UTC, in minutes. The offset must be a valid whole number between -900 and 900 or an equivalent TimeSpan instance. The default offset is the current system timezone offset.

5.4.6.11. fromWeekNumberDate (Class Method)

Returns a new DateTime object, created from a string in ISO week date format. The time component will be set to 00:00:00.000000.

Both the basic format yyyywwd and the extended format yyyy-ww-d are accepted, where ww is the ordinal number of an ISO week within year yyyy, and d is the ordinal number of the calender day within the week. The allowed range for ww is 1 through the number of ISO weeks in year yyyy, which is either 52 or 53. Leading zeros are required for both yyyy and ww. The allowed range for d is 1 through 7, where Monday is 1, Tuesday is 2, running through 7 for Sunday.

If specified, offset is the offset from UTC, in minutes. The offset must be a valid whole number between -900 and 900 or an equivalent TimeSpan instance. The default offset is the current system timezone offset.

See also methods weekNumberDate, weekNumberYear, and weekDay.
Example 5.228. DateTime class — fromWeekNumberDate method

```plaintext
say .DateTime~fromWeekNumberDate("2020-W07-1")    -- 2020-02-10T00:00:00.000000
```

5.4.6.12. fromNormalTime (Class Method)

Returns a new `DateTime` object, created from a string in the format returned by the `Normal` option of the `TIME` built-in function (`hh:mm:ss`). The date component will be set to 1 January 0001.

If specified, `offset` is the offset from UTC, in minutes. The `offset` must be a valid whole number between -900 and 900 or an equivalent `TimeSpan` instance. The default `offset` is the current system timezone offset.

5.4.6.13. fromCivilTime (Class Method)

Returns a new `DateTime` object, created from a string in the format returned by the `Civil` option of the `TIME` built-in function (`hh:mm:xx`). The date component will be set to 1 January 0001.

If specified, `offset` is the offset from UTC, in minutes. The `offset` must be a valid whole number between -900 and 900 or an equivalent `TimeSpan` instance. The default `offset` is the current system timezone offset.

5.4.6.14. fromLongTime (Class Method)

Returns a new `DateTime` object, created from a string in the format returned by the `Long` option of the `TIME` built-in function (`hh:mm:ss.uuuuuu`). The date component will be set to 1 January 0001.

If specified, `offset` is the offset from UTC, in minutes. The `offset` must be a valid whole number between -900 and 900 or an equivalent `TimeSpan` instance. The default `offset` is the current system timezone offset.

5.4.6.15. fromTicks (Class Method)

Returns a new `DateTime` object, created from a string in the format returned by the `Ticks` option of the `DATE` or `TIME` built-in functions (`dddddddddddd`).
If specified, offset is the offset from UTC, in minutes. The offset must be a valid whole number between -900 and 900 or an equivalent TimeSpan instance. The default offset is the current system timezone offset.

5.4.6.16. fromIsoDate (Class Method)

\[ \text{fromIsoDate(date, offset)} \]

Returns a new DateTime object, created from a string in extended ISO format yyyy-mm-ddThh:mm:ss.uuuuuu.

The isoDate and the string methods return a string in extended ISO format as the string form of a DateTime instance.

If specified, offset is the offset from UTC, in minutes. The offset must be a valid whole number between -900 and 900 or an equivalent TimeSpan instance. The default offset is the current system timezone offset.

5.4.6.17. fromUtcIsoDate (Class Method)

\[ \text{fromUtcIsoDate(date)} \]

Returns a new DateTime object, created from a string in timezone-qualified extended ISO format yyyy-mm-ddThh:mm:ss.uuuuuu+hh:mm.

5.4.6.18. init

\[ \text{init(fulldate, offset)} \]
\[ \text{init(year, month, day, offset)} \]
\[ \text{init(year, month, day, hours, minutes, offset)} \]
\[ \text{init(year, month, day, hours, minutes, seconds, offset)} \]
\[ \text{init(year, month, day, hours, minutes, seconds, microseconds, offset)} \]

Initializes a new DateTime instance. If no arguments are specified, the instance is set to the current date and time. If the single fulldate argument is used, the instance is initialized to the date and time calculated by adding fulldate microseconds to 0001-01-01T00:00:00.000000. If the year, month, day, form is used, the instance is initialized to 00:00:00.000000 on the indicated date. Otherwise, the instance is initialized to the year, month, day, hours, minutes, seconds, and microseconds components. Each of these components must be a valid whole number within the acceptable range.
for the given component. For example, year must be in the range 1-9999, while minutes must be in the range 0-59.

If specified, offset is the offset from UTC, in minutes. The offset must be a valid whole number between -900 and 900 or an equivalent TimeSpan instance. The default offset is the current system timezone offset.

Example 5.229. DateTime class

<table>
<thead>
<tr>
<th>today = .DateTime~new</th>
<th>-- current date and time</th>
</tr>
</thead>
<tbody>
<tr>
<td>day = .DateTime~new(date('F', &quot;2017\0630&quot;, &quot;S&quot;))</td>
<td>2017-06-30T00:00:00.000000</td>
</tr>
<tr>
<td>day = .DateTime~new(2017, 6, 30)</td>
<td>2017-06-30T00:00:00.000000</td>
</tr>
<tr>
<td>day = .DateTime~new(2017, 6, 30, 11, 8, 50)</td>
<td>2017-06-30T11:08:50.000000</td>
</tr>
</tbody>
</table>

5.4.6.19. Arithmetic Methods

Returns the result of performing the specified arithmetic operation on the receiver DateTime instance. Depending on the operation, the argument may be either a TimeSpan object or a DateTime instance. See the description of the individual operations for details. The arithmetic_operator can be:

- **+**
  - Addition. Adds a TimeSpan to the DateTime object, returning a new DateTime instance. The receiver DateTime object is not changed. The TimeSpan may be either positive or negative.

- **-**
  - Subtraction. If argument is a DateTime object, the two times are subtracted, and a TimeSpan object representing the interval between the two times is returned. If the receiver DateTime is less than the argument argument DateTime, a negative TimeSpan interval is returned. The receiver DateTime object is not changed.

  If argument is a TimeSpan object, subtracts the TimeSpan from the DateTime object, returning a new DateTime instance. The receiver DateTime object is not changed. The TimeSpan may be either positive or negative.

Prefix -
- A prefix - operation on a DateTime object will raise a SYNTAX error condition.

Prefix +
- Returns a new DateTime object with the same time value.
When adding or subtracting `DateTime` and `TimeSpan` objects, leap seconds (like the one which happened on December 31, 2016 at 23:59:60 UTC) are not taken into account.

Example 5.230. DateTime class — arithmetic

```plaintext
let t = .dateTime~new~timeOfDay -- returns TimeSpan for current time
say t

let d = .dateTime~new(2016, 12, 31) -- creates new date
let future = d + t  -- adds timespan to d
say future

-- does not take leap second into account
say d + .TimeSpan~new(24, 0, 1) -- 2017-01-01T00:00:01.000000

let nextCentury = .dateTime~new(2101, 1, 1) -- "real" start of next century
say "The next century starts in" (nextCentury - .dateTime~new) "days"
```

5.4.6.20. compareTo

`compareTo(other)`

Returns -1 if the `other` is larger than the receiving object, 0 if the two objects are equal, and 1 if `other` is smaller than the receiving object.

5.4.6.21. weekNumber

`weekNumber`

Returns the timestamp’s ISO week number.

Any given year will have week numbers either between 1 and 52, or between 1 and 53.

See also methods `weekNumberYear`, `weekNumberDate`, and `weeksInYear`.

Example 5.231. DateTime class — weekNumber method

```plaintext
say weekNumber(2018, 12, 31) -- 1 2019-W01-1
say weekNumber(2019, 1, 1)  -- 1 2019-W01-2

say weekNumber(2020, 12, 31) -- 53 2020-W53-4
say weekNumber(2021, 1, 1)  -- 53 2020-W53-5
return

weekNumber: procedure
let date = .DateTime~new(arg(1), arg(2), arg(3))
return date~weekNumber date~weekNumberDate
```
5.4.6.22. `weekNumberDate`

Returns the timestamp formatted as a string in extended ISO week number date format `yyyy-Www-d` with leading zeros as required.

See also methods `fromWeekNumberDate (Class Method)` and `weekNumberYear`, and `weekDay`.

Example 5.232. DateTime class — weekNumberDate method

```prolog
say .DateTime~new(2018, 12, 31)~weekNumberDate    -- 2019-W01-1
say .DateTime~new(2019, 1, 1)~weekNumberDate     -- 2019-W01-2
say .DateTime~new(2020, 12, 31)~weekNumberDate  -- 2020-W53-4
say .DateTime~new(2021, 1, 1)~weekNumberDate    -- 2020-W53-5
```

5.4.6.23. `weekNumberYear`

Returns the year associated with this date's ISO week number. The returned year may be the next or the previous year for dates at the beginning or end of the year.

See also methods `weekNumber`, `weekNumberDate`, and `weeksInYear`.

Example 5.233. DateTime class — weekNumberYear method

```prolog
say weekNumber(2018, 12, 31)    -- 2019 2019-W01-1
say weekNumber(2019, 1, 1)      -- 2019 2019-W01-2
say weekNumber(2020, 12, 31)    -- 2020 2020-W53-4
say weekNumber(2021, 1, 1)      -- 2020 2020-W53-5
return
weekNumber: procedure
  date = .DateTime~new(arg(1), arg(2), arg(3))
  return date~weekNumberYear date~weekNumberDate
```

5.4.6.24. `weeksInYear`

Returns the number of weeks in this date's year according to the ISO week number rules.

There are either 52 or 53 ISO weeks in any given year.

See also methods `weekNumber`, `weekNumberYear`, and `weekNumberDate`.

Example 5.234. DateTime class — weeksInYear method

```prolog
do year = 2010 to 2030
```

348
5.4.6.25. year

Returns the timestamp year.

5.4.6.26. month

Returns the timestamp month.

5.4.6.27. day

Returns the timestamp day.

5.4.6.28. hours

Returns number of whole hours in the timestamp since midnight.

5.4.6.29. minutes

Returns the minutes portion of the timestamp time-of-day.

5.4.6.30. ordinalDate

Returns the timestamp formatted as a string in extended ISO ordinal date format `yyyy-ddd` with leading zeros as required.

See also methods `fromOrdinalDate (Class Method)` and `yearDay`.

Example 5.235. DateTime class — ordinalDate method

```
say .DateTime~new(2020, 2, 10)~ordinalDate    -- 2020-041
```
5.4.6.31. seconds

Seconds

Returns the seconds portion of the timestamp time-of-day.

5.4.6.32. makeString

MakeString

Returns the timestamp formatted as a string in extended ISO format `yyyy-mm-ddThh:mm:ss.uuuuuu`. This is an alias of the `string` method.

5.4.6.33. microseconds

Microseconds

Returns the microseconds portion of the timestamp time-of-day.

5.4.6.34. dayMinutes

DayMinutes

Returns the number of minutes since midnight in the timestamp time-of-day.

5.4.6.35. daySeconds

DaySeconds

Returns the number of seconds since midnight in the timestamp time-of-day.

5.4.6.36. dayMicroseconds

DayMicroseconds

Returns the number of microseconds since midnight in the timestamp time-of-day.

5.4.6.37. hashCode

HashCode

Returns a string value for the timestamp that is used as a hash value for a `MapCollection` class.

5.4.6.38. addYears

AddYears

Returns the timestamp with `years` added to it.
Returns a new DateTime instance with a number of years added to the timestamp. The receiving instance is unchanged. The years value must be a valid whole number. Negative values result in years being subtracted.

The addYear's method will take leap years into account. If the addition result would fall on February 29th of a non-leap year, the day will be rolled back to the 28th.

Example 5.236. DateTime class — addYears method

date = .DateTime~new(2016, 6, 30)
say date              -- displays "2016-06-30T00:00:00.000000"
say date~addYears(1)  -- displays "2017-06-30T00:00:00.000000"

5.4.6.39. addWeeks

Returns a new DateTime instance with a number of weeks added to the timestamp. The receiving instance is unchanged. The weeks value must be a valid number, including fractional values. Negative values result in weeks being subtracted.

5.4.6.40. addDays

Returns a new DateTime instance with a number of days added to the timestamp. The receiving instance is unchanged. The days value must be a valid number, including fractional values. Negative values result in days being subtracted.

Example 5.237. DateTime class — addDays method

date = .DateTime~new(2016, 6, 30)
say date              -- displays "2016-06-30T00:00:00.000000"
say date~addDays(1.5) -- displays "2016-07-01T12:00:00.000000"

5.4.6.41. addHours

Returns a new DateTime instance with a number of hours added to the timestamp. The receiving instance is unchanged. The hours value must be a valid number, including fractional values. Negative values result in hours being subtracted.

5.4.6.42. addMinutes

Returns a new DateTime instance with a number of minutes added to the timestamp. The receiving instance is unchanged. The minutes value must be a valid number, including fractional values. Negative values result in minutes being subtracted.
5.4.6.43. addSeconds

Returns a new DateTime instance with a number of seconds added to the timestamp. The receiving instance is unchanged. The seconds value must be a valid number, including fractional values. Negative values result in seconds being subtracted.

5.4.6.44. addMicroseconds

Returns a new DateTime instance with a number of microseconds added to the timestamp. The receiving instance is unchanged. The microseconds value must be a valid whole number. Negative values result in microseconds being subtracted.

5.4.6.45. isoDate

Returns the timestamp formatted as a string in extended ISO format yyyy-mm-ddThh:mm:ss.uuuuuu.

The string method will also return this value.

5.4.6.46. utcIsoDate

Returns the timestamp formatted as a string in extended timezone-qualified ISO format. If the timezone offset is 0, the format is yyyy-mm-ddThh:mm:ss.uuuuuuZ. If the offset is positive, the string is formatted as yyyy-mm-ddThh:mm:ss.uuuuuu+hh:mm. If the offset is negative, the result will be in the format yyyy-mm-ddThh:mm:ss.uuuuuu-hh:mm.

5.4.6.47. baseDate

Returns the number of complete days (that is, not including the timestamp day) since and including the base date, 1 January 0001, in the format dddddd (no leading zeros or whitespace characters).

The base date of 1 January 0001 is determined by extending the current Gregorian calendar backward (365 days each year, with an extra day every year that is divisible by 4 except century years that are not divisible by 400). It does not take into account any errors in the calendar system that created the Gregorian calendar originally.

5.4.6.48. yearDay

Returns the number of complete days (that is, not including the timestamp day) since and including the base date, 1 January 0001, in the format dddddd (no leading zeros or whitespace characters).
Returns the number of days, including the timestamp day, that have passed in the year the timestamp represents in the format **ddd** (no leading zeros or blanks).

See also method **ordinalDate**.

**Example 5.238. DateTime class — yearDay method**

```rexx
say .DateTime~new(2020, 2, 10)~yearDay    -- 41
```

### 5.4.6.49. weekDay

**weekDay**

Returns the timestamp weekday as an integer. The values returned use the ISO convention for day numbering. Monday is 1, Tuesday is 2, running through 7 for Sunday.

### 5.4.6.50. europeanDate

**europeanDate( separator)**

Returns the timestamp date formatted as a string in the format **dd/mm/yy**. If specified, **separator** identifies the field separator character used in the returned string. The separator must be a single character or the null string (""). A slash ("/") is the default separator.

### 5.4.6.51. languageDate

**languageDate**

Returns the timestamp date formatted as a string in an implementation- and language-dependent, or local, date format. The format is **dd month yyyy**. The name of the month is according to the national language installed on the system. If no local date format is available, the default format is returned.

**Note**

This format is intended to be used as a whole. Rexx programs must not make any assumptions about the form or content of the returned string.

### 5.4.6.52. monthName

**monthName**

Returns the name of the timestamp month, in English.

### 5.4.6.53. dayName
5.4.6.54. normalDate

Returns the timestamp date formatted as a string in the format `dd mon yyyy`. If specified, `separator` identifies the field separator character used in the returned date. The separator must be a single character or the null string (""). A blank (" ") is the default separator.

5.4.6.55. orderedDate

Returns the timestamp date formatted as a string in the format `yy/mm/dd`. If specified, `separator` identifies the field separator character used in the returned date. The separator must be a single character or the null string (""). A slash ("/") is the default separator.

5.4.6.56. standardDate

Returns the timestamp date formatted as a string in the format `yyyymmdd`. If specified, `separator` identifies the field separator character used in the returned date. The separator must be a single character or the null string (""). A null string (""") is the default separator.

5.4.6.57. usaDate

Returns the timestamp date formatted as a string in the format `mm/dd/yy`. If specified, `separator` identifies the field separator character used in the returned date. The separator must be a single character or the null string (""). A slash ("/") is the default separator.

5.4.6.58. civilTime

Returns the timestamp time formatted as a string in Civil format `hh:mmxx`. The hours can take the values 1 through 12, and the minutes the values 00 through 59. The minutes are followed immediately by the letters `am` or `pm`. This distinguishes times in the morning (12 midnight through
11:59 a.m.—appearing as **12:00am** through **11:59am** from noon and afternoon (12 noon through 11:59 p.m.—appearing as **12:00pm** through **11:59pm**). The hour has no leading zero. The minute field shows the current minute (rather than the nearest minute) for consistency with other TIME results.

### 5.4.6.59. normalTime

Returns the timestamp time formatted as a string in the default format **hh:mm:ss**. The hours can have the values **00** through **23**, and minutes and seconds, **00** through **59**. There are always two digits. Any fractions of seconds are ignored (times are never rounded).

### 5.4.6.60. longTime

Returns the timestamp time formatted as a string in the format **hh:mm:ss.uuuuuu** (where **uuuuuu** are microseconds).

### 5.4.6.61. fullDate

Returns the timestamp’s number of microseconds since 00:00:00.000000 on 1 January 0001, in the format **dddddddddddddddd** (no leading zeros or blanks).

### 5.4.6.62. utcDate

Returns the timestamp converted to UTC time as the number of microseconds since 00:00:00.000000 on 1 January 0001, in the format **dddddddddddddddd** (no leading zeros or blanks).

### 5.4.6.63. toLocalTime

Returns a new **DateTime** instance representing the time for the local timezone.

### 5.4.6.64. toUtcTime

Returns a new **DateTime** instance representing the time for the UTC timezone (offset 0).

### 5.4.6.65. toTimezone

---

355
Returns a new `DateTime` instance representing the time for the timezone indicated by `offset`.

If specified, `offset` is the offset from UTC, in minutes. The `offset` must be a valid whole number between -900 and 900 or an equivalent `TimeSpan` instance. The default `offset` is 0, which creates a `DateTime` object for UTC.

### 5.4.6.66. ticks

Returns the timestamp's number of seconds since 00:00:00.000000 on 1 January 1970, in the format `dddddddddddd` (no leading zeros or blanks). Times prior to 1 January 1970 are returned as a negative value.

### 5.4.6.67. offset

Returns the timestamp timezone as an offset in minutes from UTC. Timezones east of UTC will return a positive offset. Timezones west of UTC will return a negative offset.

### 5.4.6.68. date

Returns a new `DateTime` instance for the timestamp date, with the time component set to 00:00:00.000000.

### 5.4.6.69. timeOfDay

Returns the interval since 00:00:00.000000 of the timestamp day as a `TimeSpan` object.

### 5.4.6.70. elapsed

Returns the difference between the current date/time and the timestamp date/time as a `TimeSpan` object. The time span will be negative if the receiving instance represents a time in the future.

### 5.4.6.71. isLeapYear

Returns the number of seconds since 00:00:00.000000 on 1 January 1970, in the format `dddddddddddd` (no leading zeros or blanks). Times prior to 1 January 1970 are returned as a negative value.
Returns `.true` if the timestamp year is a leap year. Returns `.false` otherwise.

### 5.4.6.72. daysInMonth

Returns the number of days in the timestamp month. For example, for dates in January, 31 is returned. The `daysInMonth` method takes leap years into account, returning 28 for February in non-leap years, and 29 for leap years.

### 5.4.6.73. daysInYear

Returns the number of days in the timestamp year. For leap years, 366 is returned, and 365 for non-leap years.

### 5.4.6.74. string

Returns the timestamp formatted as a string in extended ISO format `yyyy-mm-ddThh:mm:ss.uuuuu`. The `isoDate` method will also return this value.

### 5.4.7. EventSemaphore Class

An event semaphore is a synchronization mechanism that can be used to indicate to activities when a particular condition—the event—has become true.

An event can be set by posting the event semaphore, or cleared by resetting it. An activity, a concurrent chain of execution, can choose to get suspended while waiting for the event to become true, at which point the activity is released and continues to execute.

#### Table 5.42. EventSemaphore Class

<table>
<thead>
<tr>
<th>Object</th>
<th>Methods inherited from the <code>Object class</code></th>
<th>Class (Metaclass)</th>
<th>Methods inherited from the <code>Class class</code></th>
<th>EventSemaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>new (Class Method)</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>isPosted</code></td>
<td><code>reset</code></td>
<td><code>wait</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>post</code></td>
<td><code>uninit</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
EventSemaphore Class

new

Returns a new EventSemaphore instance. Initially the semaphore is in the cleared state.

5.4.7.2. isPosted

isPosted

Returns .true if the event semaphore is in the posted state, .false if it is currently cleared.

See also methods post and reset.

Example 5.239. EventSemaphore class — isPosted method

| sem = .EventSemaphore~new |
| say sem~isPosted          -- 0 |
| sem~post                  |
| say sem~isPosted          -- 1 |

5.4.7.3. post

post

Sets the event semaphore to the posted state. All suspended activities waiting for this event are released. Activities calling the wait method while the event semapahore is already in the posted state will continue executing without getting suspended.

See also method reset.

5.4.7.4. reset

reset

Clears the event semaphore. During the time the semaphore is cleared, any activities starting a blocking wait get suspended.

See also method post.

5.4.7.5. uninit

uninit

This method cleans up the event semaphore when it is garbage collected.
uninit should not be invoked directly except via an uninit method of a subclass of the EventSemaphore class. Any such subclassed uninit method must forward to the superclass uninit method.

5.4.7.6. wait

Returns .true if waiting for the event semaphore to get posted has been successful or the semaphore is already in the posted state, and .false if a timeout occurred while waiting.

If timeout is specified it must be a TimeSpan instance or a valid Rexx number. If the value is negative or if timeout is omitted, wait suspends the current activity until the semaphore gets posted.

If timeout is zero, wait immediately returns with a return value as if isPosted had been called.

If the timeout period is positive, wait suspends the current activity for timeout seconds or until the semaphore gets posted, whatever comes first.

Any number of activities can wait for an event semaphore. When the semaphore is posted, all waiting activities are released. The exact order in which released activities resume execution is unspecified and should not be relied upon.

Example 5.240. EventSemaphore class — wait method

```rexx
event = .EventSemaphore~new
say "main starts tasks"
do nr = 1 to 5
   .task~new~waitFor(event, "task" nr)
end
call SysSleep 0.1
say "main posts"
event~post
say "main ends"
::class Task
::method waitFor
   reply
   use strict arg event, name
   say name "waits"
   event~wait
   say name "runs"
```

may output

```plaintext
main starts tasks
task 2 waits
task 5 waits
```
5.4.8. File Class

The File class provides services which are common to all Rexx-supported filesystems. A File object represents a path to a file or directory. The path can be relative or absolute.

If you create a File object with a relative path, the absolute path will be calculated using the current default directory. This absolute path is memorized on the File object, and will not change if you change the default directory.

<table>
<thead>
<tr>
<th>Table 5.43. File Class</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Methods inherited from the Object class</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Class (Metaclass)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Methods inherited from the Class class</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>+ Orderable (Mixin Class)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Methods inherited from the Orderable class</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Comparison Methods</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>compareTo</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>+ Comparable (Mixin Class)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Methods inherited from the Comparable class</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>compareTo</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>File</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>isCaseSensitive (Class Method)</th>
<th>searchPath (Class Method)</th>
</tr>
</thead>
<tbody>
<tr>
<td>listRoots (Class Method)</td>
<td>separator (Class Method)</td>
</tr>
<tr>
<td>pathSeparator (Class Method)</td>
<td>temporaryPath (Class Method)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>absoluteFile</th>
<th>isDirectory</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolutePath</td>
<td>isFile</td>
</tr>
<tr>
<td>canRead</td>
<td>isHidden</td>
</tr>
<tr>
<td>canWrite</td>
<td>lastAccessed (Attribute)</td>
</tr>
<tr>
<td>compareTo</td>
<td>lastModified (Attribute)</td>
</tr>
<tr>
<td>delete</td>
<td>length</td>
</tr>
<tr>
<td>exists</td>
<td>list</td>
</tr>
<tr>
<td>extension</td>
<td>listFiles</td>
</tr>
<tr>
<td>hashcode</td>
<td>makeDir</td>
</tr>
<tr>
<td>init</td>
<td>makeDirs</td>
</tr>
<tr>
<td>isCaseSensitive</td>
<td>makeString</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent</td>
</tr>
<tr>
<td>parentFile</td>
</tr>
<tr>
<td>pathSeparator</td>
</tr>
<tr>
<td>separator</td>
</tr>
<tr>
<td>setWritable</td>
</tr>
</tbody>
</table>

5.4.8.1. isCaseSensitive (Class Method)
**isCaseSensitive**

Returns the case-sensitivity, `.true` or `.false`, of root (`/`) on Unix-like systems. On Windows, returns the case-sensitivity of the Windows system directory.

See also instance method `isCaseSensitive`.

### 5.4.8.2. listRoots (Class Method)

Returns the file system root elements, as an array of strings. On Windows, each of the drives is a root element in the format `d:\`. On Unix-like systems, there is just one root (`/`).

**Example 5.241. File class**

```
say .File~listRoots~toString(, " ")  -- C:\ D:\ E:\ R:\ (e. g. on Windows)
say .File~listRoots~toString(, " ")  -- / (Unix-like system)
```

### 5.4.8.3. pathSeparator (Class Method)

Returns the separator used for file search paths, `";"` on Windows and `":"` on Unix-like systems.

See also instance method `pathSeparator`.

### 5.4.8.4. searchPath (Class Method)

Returns a new `File` instance of the file `name`, if it can be located along `path`. Returns `.nil` otherwise.

The `name` must be a valid file name without wildcard characters and may optionally include a relative path. The `path` must be a String, a Collection, or any object that provides a `makeArray` method. Specified `path` items must be valid directories separated by the platform's `path separator`. If omitted, `path defaults` to the value of the `PATH` environment variable.

See also RexxUtil function `SysSearchPath`.

**Example 5.242. File class — searchPath method**

```
say .File~searchPath("rexx.exe")     -- e. g. C:\Program Files\ooRexx\rexx.exe
say .File~searchPath("rexx")         -- e. g. /usr/local/bin/rexx
```

### 5.4.8.5. separator (Class Method)
Returns the file name separator used by the file system ("\" on Windows, "/" on Unix-like systems).

This query method is available as both an instance and class method.

**Example 5.243. File class — separator method**

```rexx
file = .File~new("dir1" || .File~separator || "dir2" || .File~separator || "file")
-- "dir1/dir2/file" on Unix-like systems, "dir1\dir2\file" on Windows
```

### 5.4.8.6. temporaryPath (Class Method)

Returns the full path to the user’s temporary directory as a new instance of File.

On Windows this method returns the value of the environment variable **TMP**, **TEMP** or **USERPROFILE**, whichever is defined in this sequence. If none of them are defined it returns the current directory. On Unix-like systems it returns the value of the environment variable **TMPDIR**, or returns `/tmp` if **TMPDIR** is undefined.

See also RexxUtil function **SysTempFileName**.

**Example 5.244. File class — temporaryPath method**

```rexx
say .File~temporaryPath    -- (Windows e.g.) C:\Users\USER~1\AppData\Local\Temp
say .File~temporaryPath    -- (Unix-like system e.g.) /tmp
```

### 5.4.8.7. absoluteFile

Returns the fully qualified path as a new instance of File.

**Example 5.245. File class — absoluteFile method**

```rexx
/* On Windows */
'cd c:\program files\oorexx'
say .File-new("my file")~absoluteFile~class        -- The File class
say .File-new("my file")~absoluteFile              -- c:\program files\oorexx\my file
say .File-new("..\my file")~absoluteFile          -- c:\program files\my file
say .File-new("..\.\my file")~absoluteFile        -- c:\my file
say .File-new("..\..\my dir\my file")~absoluteFile -- c:\my dir\my file

/* On Linux */
'cd /opt/oorexx'
say .File-new("my file")~absoluteFile              -- /opt/oorexx/my file
say .File-new("..\my file")~absoluteFile          -- /opt/my file
```

### 5.4.8.8. absolutePath


Returns the fully qualified path as a string.

**Example 5.246. File class — absolutePath method**

```
/* On Windows */
'cd c:\program files\oorexx'
say .File~new("my file")~absolutePath~class -- The String class
say .File~new("my file")~absolutePath -- c:\program files\oorexx\my file
say .File~new("..\my file")~absolutePath -- c:\program files\my file
say .File~new("..\.my file")~absolutePath -- c:\my file
say .File~new("..\.\dir\my file")~absolutePath -- c:\my dir\my file

/* On Linux */
'cd /opt/ooRexx'
say .File~new("my file")~absolutePath -- /opt/ooRexx/my file
say .File~new("../my file")~absolutePath -- /opt/my file
```

### 5.4.8.9. canRead

**canRead**

Returns **true** if the file exists and is readable. Otherwise returns **false**.

See also methods **canWrite, setReadOnly, setWritable**.

### 5.4.8.10. canWrite

**canWrite**

Returns **true** if the file exists and is writable. Otherwise returns **false**.

See also methods **canRead, and setReadOnly, setWritable**.

### 5.4.8.11. compareTo

**compareTo**(other)

Performs a sorting comparison of the target File object to the other File object. The comparison is made on the absolute paths (strings) of both File objects. If the filesystem is case-sensitive then the paths comparison is case-sensitive, otherwise the comparison is caseless. If the two paths are equal, 0 is returned. If the target path is larger, 1 is returned. -1 if the other argument is the larger path.

**Example 5.247. File class — compareTo method**

```
call directory .File~listRoots[1]
file1 = .File~new("file", "dir")
file2 = .File~new("FILE", "DIR")
file1~compareTo(file2) -- 0 on Windows (both Files denote the same path)
file1~compareTo(file2) -- 1 on Unix-like system ("/dir/file" is greater than "/DIR/FILE")
```

### 5.4.8.12. delete
5.4.8.13. exists

Returns .true. if the file or directory (denoted by the absolute path of the target File object) exists. Otherwise returns .false.

5.4.8.14. extension

Returns the file's extension, the portion of the file name after the last dot.

See also method name.

Example 5.248. File class — extension method

```plaintext
say .File-new("/usr/local/lib/rexx.img")-extension    -- img
say .File-new("/")-name                               -- ""
```

5.4.8.15. hashCode

Returns a string value that is used as a hash value for MapCollection such as Table, Relation, Set, Bag, and Directory.

5.4.8.16. init

Initializes a new File instance with the path path (after normalization).

If specified, dir is a parent path that is prepended to path. If dir is a File object then the absolute path of dir is prepended, otherwise dir is prepended as-is (after normalization). The normalization consists in adjusting the separators to the platform's convention and removing the final separator (if any).

Example 5.249. File class — init method

```plaintext
/* Windows */
file = .File-new("file")                           -- file
file = .File-new("c:\program files\")            -- c:\program files
```
5.4.8.17. isCaseSensitive

Returns the case-sensitivity, `.true` or `.false`, of the file or directory represented by the absolute path of the target File object. If the referenced file or directory does not exist, the case-sensitivity of the final existing folder along the referenced absolute path is returned.

See also class method `isCaseSensitive (Class Method)`.

5.4.8.18. isDirectory

Returns `.true` if the absolute path of the target File object references a directory. Otherwise returns `.false`.

5.4.8.19. isFile

Returns `.true` if the absolute path of the target File object references a file. Otherwise returns `.false`.

5.4.8.20. isHidden

Returns `.true` if the absolute path of the target File object references an existing file or directory which is hidden. Otherwise returns `.false`.

On Windows, a file or directory is hidden when its attribute FILE_ATTRIBUTE_HIDDEN is set.

On Unix-like systems, a file or directory is hidden when its name starts with a period character ("."), or when one of its parent directories has a name starting with a period character.

Example 5.250. File class — isHidden method

```
say .File-new("/tmp/.file")-isHidden -- 1
say .File-new("/tmp/.dir/file")-isHidden -- 1

5.4.8.21. lastAccessed (Attribute)

lastAccessed get:
Returns the last access date of the file or directory denoted by the absolute path of the receiver object. The result is a DateTime object, or .nil if the file or directory doesn't exist or the last access time stamp cannot be retrieved.

lastAccessed set:
If the file or directory denoted by the absolute path of the receiver object exists, this sets the last access date of the file or directory. Otherwise it does nothing.

The date parameter must be a DateTime object.

See also method lastModified (Attribute).

Note
File last access time stamps may not be available on all file systems. Also, on Windows last access time is not updated for NTFS volumes by default.

Example 5.251. File class — lastAccessed attribute
say .File-new(".")-lastAccessed -- e. g. 2020-02-06T13:03:42.143095

5.4.8.22. lastModified (Attribute)

lastModified get:
Returns the last modified date of the file or directory denoted by the absolute path of the receiver object. The result is a DateTime object, or .nil if the file or directory doesn't exist or the last modified time stamp cannot be retrieved.

lastModified set:
If the file or directory denoted by the absolute path of the receiver object exists, this sets the last modified date of the file or directory. Otherwise it does nothing.

The date parameter must be a DateTime object.

See also method lastAccessed (Attribute).
Example 5.252. File class — lastModified attribute

```/* On Windows */
say .File~new("C:\Program Files")~lastModified~class  -- The DateTime class
say .File~new("C:\Program Files")~lastModified        -- e.g. 2018-06-18T11:20:17.000000
say .File~new("dummy")~lastModified                   -- e.g. The NIL object

/* A possible implementation of : touch -c -m -r referenceFile file
   -c, --no-create             do not create any files
   -m                          change only the modification time
   -r, --reference=FILE        use this file's time instead of current time
*/
parse arg referenceFilePath filePath .
file = .File~new(filePath)
if \file~exists then
  return 0         -- OK, not an error
referenceFile = .File~new(referenceFilePath)
referenceDate = referenceFile~lastModified
if referenceDate == .nil then
  return 1         -- KO
file~lastModified = referenceDate
return 0           -- OK```

5.4.8.23. length

Returns the size in bytes of the file/directory denoted by the absolute path of the receiver object.

5.4.8.24. list

Returns an array of files/directories names which are immediate children of the directory denoted by the absolute path of the receiver object. The order in which the names are returned is dependent on the file system (not necessarily alphabetic order). The special names "." and "." are not returned.

The result is an array of strings. If the receiver object is not a directory then the result is .nil.

Example 5.253. File class — list method

```names = .File-new("c:\program files\oorexx\samples")~list
say names~toString
/* Possible output */
api
ccreply.rex
complex.rex
drives.rex
factor.rex
(etc...)```

5.4.8.25. listFiles
listFiles

Returns an array of files/directories which are immediate children of the directory denoted by the absolute path of the receiver object. The order in which the names are returned is dependent on the file system (not necessarily alphabetic order). The special names "." and ".." are not returned.

The result is an array of File objects. If the receiver object is not a directory then the result is 

Example 5.254. File class — listFiles method

do file over deepListFiles("c:\program files\oorexx\samples")  
say file  
end

-- Depth first iteration
::routine deepListFiles
use strict arg directory, accumulator=(.List~new)
files = .File-new(directory)-listFiles
if files == .nil then return accumulator
do file over files
   accumulator~append(file)
   if file~isDirectory then call deepListFiles file~absolutePath, accumulator
end
return accumulator

/* Possible output */
c:\program files\oorexx\samples\api
c:\program files\oorexx\samples\api\callrxnt
c:\program files\oorexx\samples\api\callrxnt\backward.fnc
c:\program files\oorexx\samples\api\callrxnt\callrxnt.c
c:\program files\oorexx\samples\api\callrxnt\callrxnt.exe
c:\program files\oorexx\samples\api\callrxnt\callrxnt.ico
c:\program files\oorexx\samples\api\callrxnt\callrxnt.mak
c:\program files\oorexx\samples\api\callrxw

c:\program files\oorexx\samples\api\callrxwn\backward.fnc
(etc...)

5.4.8.26. makeDir

makeDir

Makes just the directory represented by the last name portion of the receiver object's absolute path. Does not create any parent directories, which must all exist for a successful creation of the leaf directory.

Returns .true if the creation was successful, otherwise returns .false. If the directory already exists then the result is .false.

5.4.8.27. makeDirs

makeDirs

Creates the entire directory hierarchy represented by the absolute path of the receiver object.

Returns .true if the creation was successful, otherwise returns .false. If the directory already exists then the result is .false.
5.4.8.28. makeString

Returns the fully qualified path as a string. This is an alias of the `absolutePath` method.

5.4.8.29. name

Returns the name portion of the receiver object's absolute path. This is everything after the last path separator. The file's extension is part of the name.

See also method `extension`.

Example 5.255. File class — name method

/* On Windows */
say .File~new("c:\program files\oorexx\rexx.exe")~name   -- rexx.exe
say .File~new("c:\")~name                                -- empty string

/* On Unix-like system */
say .File~new("/usr/local/lib/rexx.img")~name            -- rexx.img
say .File~new("/")~name                                  -- empty string

5.4.8.30. parent

Returns the parent directory portion of the receiver object's absolute path. If no separator is found or the absolute path ends with a separator (which means this is a root path) then returns `.nil`.

Example 5.256. File class — parent method

/* On Windows */
say .File~new("c:\program files\oorexx\rexx.exe")~parent   -- c:\program files\oorexx
say .File~new("c:\")~parent                                -- The NIL object
say .File~new("c:\")~parent                                 -- The NIL object

/* On Unix-like system */
say .File~new("/opt/ooRexx/bin/rexx.img")~parent            -- /opt/ooRexx/bin
say .File~new("/")~parent                                  -- The NIL object

5.4.8.31. parentFile

Returns the `parent` directory portion as a File object. If no separator is found or the absolute path ends with a separator (which means this is a root path) then returns `.nil`.
5.4.8.32. path

Returns the original path (after normalization) used to create the File object. The normalization consists in adjusting the separators to the platform's convention and removing the final separator (if any).

5.4.8.33. pathSeparator

Returns the separator used for file search paths, ";' on Windows and ";:" on Unix-like systems.
See also class method pathSeparator (Class Method).

5.4.8.34. renameTo
dest

Changes the name of the file/directory denoted by the absolute path of the target object. The new name is dest.
Returns true if the renaming was successful, otherwise returns false.
On Windows, this method calls the MoveFile API to perform the action.
On Unix-like systems, this method calls the rename API to perform the action.

5.4.8.35. separator

Returns the file name separator used by the file system ("\" on Windows, "/" on Unix-like systems).
This query method is available as both an instance and class method.

5.4.8.36. setReadOnly

Sets the read-only flag of the file or directory denoted by the absolute path of the target object.
See also methods setWritable, canRead, and canWrite.

5.4.8.37. setWritable

Clears the read-only flag of the file or directory denoted by the absolute path of the target object.
See also methods `setReadOnly`, `canRead`, and `canWrite`.

### 5.4.8.38. string

Returns a string that indicates the path used to create the File object.

### 5.4.9. MessageNotification Class

The `MessageNotification` class implements the notification interface for the `Message` class.

For any message, notification of completion of the message can be requested by using the `notify` method of the Message class. This method requires as its argument a notification target, that implements this message notification interface. If notification was requested, upon message completion a `messageComplete` message is sent to the notification target.

This class is defined as a MIXIN class.

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the</td>
</tr>
<tr>
<td><code>Object class</code></td>
</tr>
<tr>
<td><code>Class (Metaclass)</code></td>
</tr>
<tr>
<td>Methods inherited from the</td>
</tr>
<tr>
<td><code>Class class</code></td>
</tr>
<tr>
<td><code>MessageNotification (Mixin Class)</code></td>
</tr>
</tbody>
</table>

#### 5.4.9.1. messageComplete (Abstract Method)

Whenever a message completes processing, for which notification was requested by using the `notify` method of the Message class, this method of the message notification target will be called. The Message object which completed processing will be provided as argument `source`.

It is an abstract method and must be implemented in a subclass.

### 5.4.10. Monitor Class

The `Monitor` class acts as a proxy for other objects. Messages sent to the Monitor object are forwarded to a different target object. The message target can be changed dynamically.

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the</td>
</tr>
<tr>
<td><code>Object class</code></td>
</tr>
<tr>
<td><code>Class (Metaclass)</code></td>
</tr>
<tr>
<td>Methods inherited from the</td>
</tr>
<tr>
<td><code>Class class</code></td>
</tr>
<tr>
<td>Monitor</td>
</tr>
</tbody>
</table>
5.4.10.1. current

Returns the current destination object.

5.4.10.2. destination

Returns a new destination object. If you specify `destination`, this becomes the new destination for any forwarded messages. If you omit `destination`, the previous destination object becomes the new destination for any forwarded messages.

5.4.10.3. init

Initializes the newly created monitor object.

5.4.10.4. unknown

Reissues or forwards to the current monitor destination all unknown messages sent to a monitor object. For additional information, see Section 4.2.6, "Defining an UNKNOWN Method".

5.4.10.5. Examples

Example 5.257. Class MONITOR

```plaintext
.local-setentry("output",.monitor-new(.stream-new("my.new")~~command("open nobuffer")))

/* The following sets the destination */
previous_destination=.output-destination(.stream-new("my.out")~~command("open write"))

/* The following resets the destination */
.output-destination

.output-destination(.Stdout)
current_output_destination_stream_object=.output-current
```
5.4.11. MutableBuffer Class

The MutableBuffer class is a buffer on which certain string operations such as concatenation can be performed very efficiently. Unlike String objects, MutableBuffers can be altered without requiring a new object allocation. A MutableBuffer object can provide better performance for algorithms that involve frequent concatenations to build up longer string objects because it creates fewer intermediate objects.

Table 5.46. MutableBuffer Class

<table>
<thead>
<tr>
<th>Object methods inherited from the <strong>Object class</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <strong>Class (Metaclass)</strong></td>
</tr>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
<tr>
<td><strong>MutableBuffer</strong> methods</td>
</tr>
</tbody>
</table>

5.4.11.1. new (Class Method)

Initialize the buffer, optionally assign a buffer content and a starting buffer size. The default size is 256; the buffer size increases to the length of string if the string does not fit into the buffer.

5.4.11.2. []

...
MutableBuffer Class

Returns a substring of the receiving buffer that begins at the \( n \)'th character and is of length \( length \). The \( n \) must be a positive whole number. If \( length \) is omitted, it defaults to 1. If \( n \) is greater than the length of the receiving string, a null string is returned.

See also methods \( substr \) and \( subChar \).

Example 5.258. MutableBuffer class — \( [] \) method

```haskell
s = .MutableBuffer~new(xrange("a", "z"))
say s[5] -- "e"
say s[18, 3] -- "rst"
say s[25,10] -- "yz"
say s[30] -- ""
```

5.4.11.3. \( [] = \)

Returns the receiving buffer with the characters from the \( n \)th character for \( length \) characters replaced with \( new \). The replacement position and length can extend beyond the end of the receiving string. The starting position, \( n \), is required and must be a positive whole number. The \( length \) is optional and must be a positive whole number or zero. If \( length \) is omitted, it defaults to the length of \( new \).

If \( n \) is greater than the length of the receiving string, blanks are added before the \( new \) string.

See also the similar MutableBuffer method \( replaceAt \).

Example 5.259. MutableBuffer class — \( [] = \) method

```haskell
s = .MutableBuffer~new(xrange("a", "z"))
s[5] = "E" -- "abcdEfghijklmnopqrstuvwxyz"
s[5] = "XXXX" -- "abcdXXXXijklmnopqrstuvwxyz"
s[5] = "" -- "abcdXXXXijklmnopqrstuvwxyz"
s[5, 1] = ":" -- "abcd=ijklmnopqrstuvwxyz"
s[5, 4] = ";" -- "abcd=ijklmnopqrstuvwxyz"
s[10, 2] = "REPLACE" -- "abcd=ijklREPLACEijklmnopqrstuvwxyz"
s[4, 99] = "" -- "abc"
```

5.4.11.4. append

Returns the receiving buffer with all \( string \)s appended to the buffer content. The buffer size is increased if necessary.

5.4.11.5. caselessChangeStr
MutableBuffer Class

5.4.11.6. caselessContains

`caselessContains(other, start, length)`

Returns `.true` if the receiving buffer contains the `other` string. It returns `.false` if `other` is the null string or is not found within the receiving buffer. The search is performed using caseless comparisons.

By default, the search starts at the first character of the receiving buffer and continues to the end of the buffer. You can override this by specifying `start`, the point at which the search starts, and `length`, the bounding limit for the search. If specified, `start` must be a positive whole number and `length` must be a non-negative whole number.

See also methods `contains, caselessStartsWith, caselessEndsWith, and caselessPos`.

Example 5.260. MutableBuffer class — caselessContains method

```plaintext
say .mutablebuffer~new('-abcdef-')~caselessContains('EF')         -- 1
say .mutablebuffer~new('-abcdef-')~caselessContains('-', 2, 6)    -- 0
```

5.4.11.7. caselessContainsWord

`caselessContainsWord(phrase, start)`

Returns `.true` if `phrase` is found in the receiving buffer. Returns `.false` if `phrase` contains no words or if `phrase` is not found. Word matches are made independent of case. Multiple whitespace characters between words in either `phrase` or the receiving buffer are treated as a single blank for the comparison, but, otherwise, the words must match, except for case.

By default the search starts at the first word in the receiving buffer. You can override this by specifying `start` (which must be a positive whole number), the word at which the search is to be started.

See also methods `containsWord` and `caselessWordPos` (caselessContainsWord returns `.false` exactly if caselessWordPos would have returned `.false`.)

Example 5.261. MutableBuffer class — caselessContainsWord method

```plaintext
good = .MutableBuffer~new("Now is the time for all good men")
good~caselessContainsWord("the")          -- .true
good~caselessContainsWord("The")          -- .true
good~caselessContainsWord("is the")       -- .true
good~caselessContainsWord("is the ")      -- .true
```

375
MutableBuffer Class

| good-caselessContainsWord("is time") | -- .false |
| good-caselessContainsWord("time") | -- .true |
| good-caselessContainsWord("time", 5) | -- .false |

5.4.11.8. **caselessCountStr**

```
caselessCountStr(needle)
```

Returns a count of the occurrences of *needle* in the receiving MutableBuffer that do not overlap. All matches are made using caseless comparisons.

5.4.11.9. **caselessEndsWith**

```
caselessEndsWith(other)
```

Returns `.true` if the characters of the *other* match the characters at the end of the target buffer. Returns `.false` if the characters are not a match, or if *other* is the null string. The match is made using caseless comparisons.

The caselessEndsWith method is useful for efficient string parsing as it does not require new string objects be extracted from the target buffer.

See also methods `caselessStartsWith`, `endsWith`, and `caselessMatch`.

5.4.11.10. **caselessLastPos**

```
caselessLastPos(needle, start, length)
```

Returns the position of the last occurrence of a string, *needle*, in the receiving buffer. It returns 0 if *needle* is the null string or not found. By default, the search starts at the last character of the receiving buffer and scans backward to the beginning of the string. You can override this by specifying *start*, the point at which the backward scan starts and *length*, the range of characters to scan. The *start* must be a positive whole number and defaults to `receiving_buffer-length` if larger than that value or omitted. The *length* must be a non-negative whole number and defaults to *start*. The search is performed using caseless comparisons.

See also methods `lastPos` and `caselessPos`.

5.4.11.11. **caselessMatch**

```
caselessMatch(start, other, n, length)
```

Returns `.true` if the characters of the *other* match the characters of the target buffer beginning at position *start*. Returns `.false` if the characters are not a match. The matching is performed using caseless comparisons. *start* must be a positive whole number.

If *n* is specified, the match will be performed starting with character *n* of *other*. The default value for *n* is “1”. *n* must be a positive whole number less than or equal to the length of *other*. 
If `length` is specified, it defines a substring of `other` that is used for the match. `length` must be a positive whole number and the combination of `n` and `length` must be a valid substring within the bounds of `other`.

The `caselessMatch` method is useful for efficient string parsing as it does not require new string objects be extracted from the target string.

### 5.4.11.12. `caselessMatchChar`

```plaintext
| caselessMatchChar | n | chars |
```

Returns `.true` if the character at position `n` matches any character of the string `chars`. Returns `.false` if the character does not match any of the characters in the reference set. The match is made using caseless comparisons. The argument `n` must be a positive whole number.

### 5.4.11.13. `caselessPos`

```plaintext
| caselessPos | needle | start | length |
```

Returns the position in the receiving buffer of a `needle` string. It returns `0` if `needle` is the null string or is not found or if `start` is greater than the length of the receiving buffer. The search is performed using caseless comparisons. By default, the search starts at the first character of the receiving buffer (that is, the value of `start` is `1`), and continues to the end of the buffer. You can override this by specifying `start`, the point at which the search starts, and `length`, the bounding limit for the search. If specified, `start` must be a positive whole number and `length` must be a non-negative whole number.

See also method `lastPos`.

### 5.4.11.14. `caselessStartsWith`

```plaintext
| caselessStartsWith | other |
```

Returns `.true` if the characters of the `other` match the characters at the start of the target buffer. Returns `.false` if the characters are not a match, or if `other` is the null string. The match is made using caseless comparisons.

The `caselessStartsWith` method is useful for efficient string parsing as it does not require new string objects be extracted from the target buffer.

See also methods `startsWith`, `caselessEndsWith`, and `caselessMatch`.

### 5.4.11.15. `caselessWordPos`

```plaintext
| caselessWordPos | phrase | start |
```

Returns the word number of the first word of `phrase` found in the receiving buffer, or `0` if `phrase` contains no words or if `phrase` is not found. Word matches are made independent of case. Multiple
whitespace characters between words in either phrase or the receiving buffer are treated as a single blank for the comparison, but, otherwise, the words must match exactly.

By default the search starts at the first word in the receiving string. You can override this by specifying start (which must be positive), the word at which the search is to be started.

### 5.4.11.16. changeStr

![changeStr](changeStr.png)

Returns the receiver MutableBuffer with newneedle replacing occurrences of needle.

If count is not specified, all occurrences of needle are replaced. If count is specified, it must be a non-negative, whole number that gives the maximum number of occurrences to be replaced.

### 5.4.11.17. contains

![contains](contains.png)

Returns .true if the receiving buffer contains the other string. It returns .false if other is the null string or is not found within the receiving buffer.

By default, the search starts at the first character of the receiving buffer and continues to the end of the buffer. You can override this by specifying start, the point at which the search starts, and length, the bounding limit for the search. If specified, start must be a positive whole number and length must be a non-negative whole number.

See also methods caselessContains, startsWith, endsWith, and pos.

**Example 5.262. MutableBuffer class — caselessContains method**

```pascal
say .mutablebuffer~new('-abcdef-')~contains('ef')         -- 1
say .mutablebuffer~new('-abcdef-')~contains('-', 2, 6)    -- 0
```

### 5.4.11.18. containsWord

![containsWord](containsWord.png)

Returns .true if phrase is found in the receiving buffer. Returns .false if phrase contains no words or if phrase is not found. Multiple whitespace characters between words in either phrase or the receiving buffer are treated as a single blank for the comparison, but, otherwise, the words must match exactly.

By default the search starts at the first word in the receiving buffer. You can override this by specifying start (which must be positive whole number), the word at which the search is to be started.
See also methods \texttt{caselessContainsWord} and \texttt{wordPos} (\texttt{containsWord} returns \texttt{false} exactly if \texttt{wordPos} would have returned 0.)

\begin{example}
\textbf{Example 5.263.} 
\textbf{MutableBuffer class — containsWord method}
\begin{verbatim}
good = .MutableBuffer~new("Now is the time for all good men")
good~containsWord("the")  -- .true
good~containsWord("The")   -- .false
good~containsWord("is the")-- .true
good~containsWord("is the ")-- .true
good~containsWord("is time")-- .false
good~containsWord("time")  -- .true
good~containsWord("time", 5)-- .false
\end{verbatim}
\end{example}

\subsection{countStr}

\texttt{countStr(needle)}

Returns a count of the occurrences of \texttt{needle} in the receiving buffer that do not overlap.

\subsection{delete / delStr}

\texttt{delete(\texttt{delStr(n, length})}

Returns the receiver \texttt{MutableBuffer} with \texttt{length} characters deleted from the buffer beginning at the \texttt{n}'th character. If \texttt{n} is omitted, it defaults to 1. If \texttt{length} is omitted, or if \texttt{length} is greater than the number of characters from \texttt{n} to the end of the buffer, the method deletes the remaining buffer contents (including the \texttt{n}'th character). The \texttt{length} must be a positive integer or zero. The \texttt{n} must be a positive integer. If \texttt{n} is greater than the length of the buffer or \texttt{length} is zero, the method does not modify the buffer content.

\textbf{Note}

The \texttt{delete} method and the \texttt{delStr} method are identical. \texttt{delStr} is provided for polymorphism with the \texttt{String} class.

\subsection{delWord}

\texttt{delWord(n, length)}

Deletess a substring from the \texttt{MutableBuffer} that starts at the \texttt{n}th word and is of \texttt{length} whitespace-delimited words. If you omit \texttt{length}, or if \texttt{length} is greater than the number of words from \texttt{n} to the end of the receiving buffer, the method deletes the remaining words in the receiving buffer (including the \texttt{n}th word). The \texttt{length} must be a positive whole number or zero. The \texttt{n} must be a positive whole number. If \texttt{n} is greater than the number of words in the receiving buffer, the method returns the receiving buffer unchanged. The portion deleted includes any whitespace characters following the final word involved but none of the whitespace characters preceding the first word involved.
5.4.11.22. `endsWith`

```
endsWith(other)
```

Returns `.true.` if the characters of the `other` match the characters at the end of the target buffer. Returns `.false.` if the characters are not a match, or if `other` is the null string.

The `endsWith` method is useful for efficient string parsing as it does not require new string objects be extracted from the target buffer.

See also methods `startsWith`, `caselessEndsWith`, and `match`.

5.4.11.23. `getBufferSize`

```
getBufferSize
```

Retrieves the current buffer size.

See also method `setBufferSize`.

5.4.11.24. `insert`

```
insert(new, n, length, pad)
```

Returns the receiving buffer with the string `new`, padded or truncated to length `length`, inserted into the MutableBuffer after the `n`'th character. The default value for `n` is 0, which means insertion at the beginning of the string. If specified, `n` and `length` must be positive integers or zeros. If `n` is greater than the length of the buffer contents, the string `new` is padded at the beginning. The default value for `length` is the length of `new`. If `length` is less than the length of string `new`, `insert` truncates `new` to length `length`. The default `pad` character is a blank.

5.4.11.25. `lastPos`

```
lastPos(needle, start, length)
```

Returns the position of the last occurrence of a string, `needle`, in the receiving buffer. It returns 0 if `needle` is the null string or not found. By default, the search starts at the last character of the receiving buffer and scans backward to the beginning of the string. You can override this by specifying `start`, the point at which the backward scan starts and `length`, the range of characters to scan. The `start` must be a positive whole number and defaults to `receiving_buffer~length` if larger than that value or omitted. The `length` must be a non-negative whole number and defaults to `start`.

See also methods `caselessLastPos` and `pos`.

---

**Example 5.264. MutableBuffer class — lastPos method**

```plaintext
x1 = .mutablebuffer~new("abc def ghi")
```
MutableBuffer Class

| x1.lastPos(" ") | -> | 8 |
| x1 = .mutablebuffer~new("abcdefghi") |
| x1.lastPos(" ") | -> | 0 |
| x1 = .mutablebuffer~new("efgxyz") |
| x1.lastPos("xy") | -> | 4 |
| x1 = .mutablebuffer~new("abc def ghi") |
| x1.lastPos(" ",7) | -> | 4 |

5.4.11.26. length

\[ \text{length} \]

Returns length of data in buffer.

5.4.11.27. lower

\[ \text{lower}(n, \text{length}) \]

Returns the receiving buffer with the characters of the target string beginning with character \( n \) for \( \text{length} \) characters converted to lowercase. If \( n \) is specified, it must be a positive whole number. If \( n \) is not specified, the case conversion will start with the first character. If \( \text{length} \) is specified, it must be a non-negative whole number. If \( \text{length} \) is not specified, the default is to convert the remainder of the buffer.

5.4.11.28. makeArray

\[ \text{makeArray}(\text{separator}) \]

This method returns an Array of the receiving MutableBuffer's substrings that were separated by the \( \text{separator} \) string. \( \text{separator} \) may be any string, including the null string. If the null string is used, an Array containing each character of the MutableBuffer is returned. If the target MutableBuffer starts with the separator, the first Array item will be a null string. If the MutableBuffer ends with a separator, no extra null string item will be added. If \( \text{separator} \) isn't specified, any \text{line-end} indicator is honored.

5.4.11.29. makeString

\[ \text{makeString} \]

Returns the content of the buffer as the string representation of the receiving buffer.

See also method \text{string}.

5.4.11.30. match

\[ \text{match}(\text{start}, \text{other}, n, \text{length}) \]

381
MutableBuffer Class

Returns .true if the characters of the other match the characters of the target buffer beginning at position start. Returns .false if the characters are not a match. start must be a positive whole number.

If n is specified, the match will be performed starting with character n of other. The default value for n is "1". n must be a positive whole number less than or equal to the length of other.

If length is specified, it defines a substring of other that is used for the match. length must be a positive whole number and the combination of n and length must be a valid substring within the bounds of other.

The match method is useful for efficient string parsing as it does not require new string objects be extracted from the target buffer.

5.4.11.31. matchChar

matchChar(n, chars)

Returns .true if the character at position n matches any character of the string chars. Returns .false if the character does not match any of the characters in the reference set. The argument n must be a positive whole number.

5.4.11.32. overlay

overlay(new, n, length, pad)

Returns the receiving buffer after overlaying it, starting at the n'th character, with the string new, padded or truncated to length length. The overlay can extend beyond the end of the buffer. In this case the buffer size will be increased. If you specify length, it must be a positive integer or zero. The default value for length is the length of new. If n is greater than the length of the buffer content, padding is added before the new string. The default pad character is a blank, and the default value for n is 1. If you specify n, it must be a positive integer.

5.4.11.33. pos

pos(needle, start, length)

Returns the position in the receiving buffer of another string, needle. It returns 0 if needle is the null string or is not found or if start is greater than the length of the receiving buffer. By default, the search starts at the first character of the receiving buffer (that is, the value of start is 1), and continues to the end of the string. You can override this by specifying start, the point at which the search starts, and length, the bounding limit for the search. If specified, start must be a positive whole number and length must be a non-negative whole number.

See also method lastPos.
Example 5.26. MutableBuffer class — pos method

|x1 = .mutablebuffer~new("Saturday")
|x1~pos("day")       ->    6
|x1 = .mutablebuffer~new("abc def ghi")
|x1~pos("x")         ->    0
|x1~pos(" ")         ->    4
|x1~pos(" ",5)       ->    8

5.4.11.34. replaceAt

replaceAt(
new,
length,
pad)

Returns the receiving buffer with the characters from the $n$th character for $length$ characters replaced with $new$. The replacement position and length can extend beyond the end of the receiving string. The starting position, $n$, is required and must be a positive whole number. The $length$ is optional and must be a positive whole number or zero. If $length$ is omitted, it defaults to the length of $new$.

If $n$ is greater than the length of the receiving string, padding is added before the new string. The default $pad$ character is a blank.

See also the similar MutableBuffer method $[]=$.

5.4.11.35. setBufferSize

setBufferSize(
$n$
)

Returns the receiving buffer with buffer size set to $n$. If $n$ is less than the length of buffer content, the content is truncated. If $n$ is 0, the entire content is erased and the new buffer size is the value given in the init method.

See also method $getBufferSize$.

5.4.11.36. setText

setText(
string
)

Returns the receiving buffer, with the buffer contents set to $string$.

5.4.11.37. space

space(
n, pad
)

Returns the target MutableBuffer, with $n$ pad characters between each whitespace-delimited word. If you specify $n$, it must be a positive whole number or zero. If it is 0, all whitespace characters are removed. Leading and trailing whitespace characters are always removed. The default for $n$ is 1, and the default $pad$ character is a blank.
5.4.11.38. startsWith

Returns `.true.` if the characters of the `other` match the characters at the start of the target buffer. Returns `.false.` if the characters are not a match, or if `other` is the null string.

The startsWith method is useful for efficient string parsing as it does not require new string objects be extracted from the target buffer.

See also methods `caselessStartsWith`, `endsWith`, and `match`.

5.4.11.39. string

Retrieves the content of the buffer as a string.

See also method `makeString`.

5.4.11.40. subChar

Returns the `n`th character of the receiving buffer. `n` must be a positive whole number. If `n` is greater than the length of the receiving buffer then a zero-length string is returned.

5.4.11.41. substr

Returns a substring from the buffer content that begins at the `n`th character and is of length `length`, padded with `pad` if necessary. The `n` must be a positive integer. If `n` is greater than the length of the receiving buffer, only `pad` characters are returned. If you omit `length`, the remaining buffer content is returned. The default `pad` character is a blank.

5.4.11.42. subWord

Returns the substring of the receiving buffer that starts at the `n`th word and is up to `length` whitespace-delimited words. The `n` must be a positive whole number. If you omit `length`, it defaults to the number of remaining words in the receiving buffer. The returned string never has leading or trailing whitespace characters, but includes all whitespace characters between the selected words.

5.4.11.43. subWords
MutableBuffer Class

5.4.11.44. translate

Returns the receiving buffer with each character translated to another character or unchanged.

The output table is \texttt{tableo} and the input translation table is \texttt{tablei}. \texttt{translate} searches \texttt{tablei} for each character in the receiving buffer. If the character is found, the corresponding character in \texttt{tableo} is replaces the character in the buffer. If there are duplicates in \texttt{tablei}, the first (leftmost) occurrence is used. If the character is not found, the original character in the receiving buffer is unchanged.

The tables can be of any length. If you specify neither translation table and omit \texttt{pad}, the receiving string is translated to uppercase (that is, lowercase \texttt{a-z} to uppercase \texttt{A-Z}), but if you include \texttt{pad} the buffer translates the entire string to \texttt{pad} characters. \texttt{tablei} defaults to \texttt{XRange('00'x, 'ff'x)}, and \texttt{tableo} defaults to the null string and is padded with \texttt{pad} or truncated as necessary. The default \texttt{pad} is a blank.

\( n \) is the position of the first character of the translated range. The default starting position is \( 1 \). \texttt{length} is the range of characters to be translated. If \texttt{length} is omitted, the remainder of the buffer from the starting position to the end is used.

5.4.11.45. upper

Returns the receiving buffer with the characters of the target string beginning with character \( n \) for \texttt{length} characters converted to uppercase. If \( n \) is specified, it must be a positive whole number. If \( n \) is not specified, the case conversion will start with the first character. If \texttt{length} is specified, it must be a non-negative whole number. If \texttt{length} is not specified, the default is to convert the remainder of the buffer.

5.4.11.46. verify
Returns a number that, by default, indicates whether the receiving buffer is composed only of characters from \textit{reference}. It returns 0 if all characters in the receiving buffer are in \textit{reference} or returns the position of the first character in the receiving buffer not in \textit{reference}.

The \textit{option} can be either \textbf{Nomatch} (the default) or \textbf{Match}. (You need to specify only the first capitalized and highlighted letter; all characters following the first character are ignored, which can be in uppercase or lowercase.)

If you specify \textbf{Match}, the method returns the position of the first character in the receiving buffer that is in \textit{reference}, or returns 0 if none of the characters are found.

The default for \textit{start} is 1. Thus, the search starts at the first character of the receiving buffer. You can override this by specifying a different \textit{start} point, which must be a positive whole number.

The default for \textit{length} is the length of the buffer from \textit{start} to the end of the buffer. Thus, the search proceeds to the end of the receiving buffer. You can override this by specifying a different \textit{length}, which must be a non-negative whole number.

If the receiving string is null, the method returns 0, regardless of the value of the \textit{option}. Similarly, if \textit{start} is greater than \textit{receiving buffer-length}, the method returns 0. If \textit{reference} is null, the method returns 0 if you specify \textbf{Match}. Otherwise, the method returns the \textit{start} value.

5.4.11.47. \textbf{word}

Returns the \textit{n}th whitespace-delimited word in the receiving buffer or the null string if the receiving buffer has fewer than \textit{n} words. The \textit{n} must be a positive whole number. This method is exactly equivalent to \textit{receiving buffer-subWord(n, 1)}.

5.4.11.48. \textbf{wordIndex}
MutexSemaphore Class

Returns the position of the first character in the \( n \)th whitespace-delimited word in the receiving buffer. It returns 0 if the receiving buffer has fewer than \( n \) words. The \( n \) must be a positive whole number.

### 5.4.11.49. wordLength

\[ \text{wordLength}(n) \]

Returns the length of the \( n \)th whitespace-delimited word in the receiving buffer or 0 if the receiving buffer has fewer than \( n \) words. The \( n \) must be a positive whole number.

### 5.4.11.50. wordPos

\[ \text{wordPos}(\text{phrase}, \text{start}) \]

Returns the word number of the first word of \( \text{phrase} \) found in the receiving buffer, or 0 if \( \text{phrase} \) contains no words or if \( \text{phrase} \) is not found. Multiple whitespace characters between words in either \( \text{phrase} \) or the receiving buffer are treated as a single blank for the comparison, but, otherwise, the words must match exactly.

By default the search starts at the first word in the receiving buffer. You can override this by specifying \( \text{start} \) (which must be positive), the word at which the search is to be started.

### 5.4.11.51. words

\[ \text{words} \]

Returns the number of whitespace-delimited words in the receiving buffer.

### 5.4.12. MutexSemaphore Class

A mutex, or mutual exclusion semaphore is a synchronization mechanism which concurrent activities can use to control access to a common resource.

Mutual exclusion is the requirement that one activity, a \textit{concurrent} chain of execution, never enters its critical section at the same time that another concurrent activity enters its own critical section.

An activity acquires the mutex semaphore before entering its critical section, and releases it after the critical section.

See also keyword instructions \textit{GUARD ON} and \textit{GUARD OFF}.

#### Table 5.47. MutexSemaphore Class

<table>
<thead>
<tr>
<th></th>
<th>MutexSemaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Methods inherited from the</strong></td>
<td><strong>Object</strong></td>
</tr>
<tr>
<td><strong>Object class</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Methods inherited from the</strong></td>
<td><strong>Class (Metaclass)</strong></td>
</tr>
<tr>
<td><strong>Class class</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MutexSemaphore</strong></td>
<td></td>
</tr>
</tbody>
</table>
MutexSemaphore Class

### new (Class Method)

**acquire**

**release**

**uninit**

#### 5.4.12.1. new (Class Method)

Returns a new *MutexSemaphore* instance. Initially the mutex semaphore is in the released state.

#### 5.4.12.2. acquire

Returns `.true` if the current activity has already owned or has just acquired the mutex semaphore. Returns `.false` if the mutex is owned by a different activity, or a timeout has occurred.

Nested acquires, from an activity already owning the mutex semaphore, are allowed, with each acquire increasing the mutex nesting level by one. An equivalent number of calls to *release* are needed to make the mutex available again to another activity.

If `timeout` is specified it must be a *TimeSpan* instance or a valid Rexx number. If the value is negative or if `timeout` is omitted, `acquire` suspends the current activity until it can get ownership of the mutex.

If `timeout` is zero, `acquire` immediately returns `.true` if the mutex was acquired, or `.false` otherwise.

If the `timeout` period is positive, `acquire` suspends the current activity for `timeout` seconds or until the current activity can acquire the mutex, whatever comes first.

If an activity still owns mutex semaphores when it ends, these semaphores will be automatically released by the interpreter.

See also method *release*.

---

**Example 5.267. MutexSemaphore class — acquire method**

```rexx
mutex = .MutexSemaphore~new
.Task~new~startWork(mutex, "work 1")
.Task~new~startWork(mutex, "work 2")
say "work tasks started"

::class Task
::method startWork unguarded
  expose mutex name
  use strict arg mutex, name
  reply
  self~doWork(1)
```

---
MutexSemaphore Class

::method doWork unguarded
    expose mutex name
    use strict arg level
    -- five levels of nested acquires
    if level > 5 then
        return
    mutex-acquire
    say name level
    self-doWork(level + 1)

may output

work tasks started
work 2 1
work 2 2
work 2 3
work 2 4
work 2 5
work 1 1
work 1 2
work 1 3
work 1 4
work 1 5

5.4.12.3. release

Returns .true if the mutex semaphore had been owned by the current activity, returns .false otherwise.

A successful release decreases the mutex nesting level. If the nesting level has reached zero, one of the activities, if any, waiting to acquire the mutex gets released and becomes the new owner of the mutex.

See also method acquire.

5.4.12.4. uninit

This method cleans up the mutex semaphore when it is garbage collected.

Note

uninit should not be invoked directly except via an uninit method of a subclass of the MutexSemaphore class. Any such subclassed uninit method must forward to the superclass uninit method.
5.4.13. Orderable Class

The Orderable class can be inherited by classes which wish to provide each of the comparison operator methods without needing to implement each of the individual methods. The inheriting class need only implement the Comparable compareTo method.

The Orderable class is defined as a MIXIN class.

<table>
<thead>
<tr>
<th>Table 5.48. Orderable Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
</tr>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td><strong>Class (Metaclass)</strong></td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td><strong>Orderable (Mixin Class)</strong></td>
</tr>
<tr>
<td><strong>Comparison Methods</strong></td>
</tr>
<tr>
<td>= == &lt;= &lt;&lt;= &lt;&gt; &gt;= &gt;&gt;=</td>
</tr>
<tr>
<td>compareTo</td>
</tr>
</tbody>
</table>

5.4.13.1. compareTo

This method compares the receiving object to the object supplied in the other argument.

This is a default implementation which compares two items based on their identityHash.

```ruby
compareTo(other)
```

This method returns -1 if the other is larger than the receiving object, 0 if the two objects are equal, and 1 if other is smaller than the receiving object.

**Note**

Classes inheriting from Orderable (like builtin classes File, DateTime, or TimeSpan) are strongly encouraged to forward to this default implementation for comparison cases not covered by their class-specific compareTo implementation.

5.4.13.2. Comparison Methods

```ruby
comparison_operator(argument)
```

Returns true or false, the result of performing the specified comparison operation. The receiver object and the argument are the terms compared.

The comparison operators you can use in a message are:

- = .true if the terms are equal
- \=, >=, <= .true if the terms are not equal (inverse of =)
- > Greater than
- < Less than
Greater than or equal to
Not less than
Less than or equal to
Not greater than

All strict comparison operations have one of the characters doubled that define the operator. The Orderable strict comparison operators produce the same results as the non-strict comparisons.

The strict comparison operators you can use in a message are:

== .true if terms are strictly equal
\== .true if the terms are NOT strictly equal (inverse of ==)
>> Strictly greater than
<< Strictly less than
>>= Strictly greater than or equal to
\<< Strictly NOT less than
<<= Strictly less than or equal to
\>> Strictly NOT greater than

5.4.14. Pointer Class

A Pointer instance is a wrapper around a native pointer value. This class is designed primarily for writing methods and functions in native code and can only be created using the native code application programming interfaces. The Pointer class new method will raise an error if invoked.

<table>
<thead>
<tr>
<th>Object</th>
<th>Methods inherited from the Object class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
<td></td>
</tr>
<tr>
<td>Pointer</td>
<td></td>
</tr>
</tbody>
</table>

5.4.14.1. new (Class Method)

Creating Pointer object instances directly from Rexx code is not supported. The Pointer class new method will raise an error if invoked.

5.4.14.2. Comparison Methods
RegularExpression Class

Returns .true or .false, the result of performing a specified comparison operation.

For the Pointer class, the argument object must be a pointer object instance and the wrappered pointer value must be the same.

The comparison operators you can use in a message are:

  ==, === .true if the wrappered pointer values are the same.
  !=, !=, >=, <= .true if the wrappered pointer values are not the same.

5.4.14.3. isNull

isNull

Returns .true if the wrappered pointer value is a NULL pointer (0) value. Returns .false if the pointer value is non-zero.

5.4.15. RegularExpression Class

This class provides support for regular expressions. A regular expression is a pattern you can use to match strings.

Note

The RegularExpression class is not a built-in class and is not preloaded. Use ::requires "rxregexp.cls" to activate its functionality.

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>RegularExpression</td>
</tr>
<tr>
<td>new (Class Method)</td>
</tr>
</tbody>
</table>

match | pos |
parse | position |

Here is a description of the syntax:

| OR operator between the left and right expression |
| Matches any single character |
| Matches the previous expression zero or more times |
| Matches the previous expression one or more times |
Regular Expression Class

\ "Escape" symbol: use the next character literally
(…) Expression in parenthesis (use where needed)
{n} Matches previous expression n times (n > 1)
[..] Set definition: matches any single character out of the defined set.

A "^" right after the opening bracket means that none of the following characters
should be matched.

A "-" (if not used with "\") defines a range between the last specified character and
the one following ".-". If it is the first character in the set definition, it is used literally.

The following symbolic names (they must start and end with ":") can be used to abbreviate common
sets:

:alpha: Characters in the range A-Z and a-z
:lower: Characters in the range a-z
:upper: Characters in the range A-Z
:digit: Characters in the range 0-9
:alnum: Characters in :digit: and :alpha:
:xdigit: Characters in :digit:, A-F and a-f
:blank: Space and tab characters
:space: Characters '09'x to '0d'x and space
:cntrl: Characters '00'x to '1f'x and '7f'x
:print: Characters in the range '20'x to '7e'x
:graph: Characters in :print: without space
:punct: All :print: characters without space and not in :alnum:

Example 5.268. RegularExpression class

::requires "rxregexp.cls"

"(Hi|Hello) World"   Matches "Hi World" and
"Hello World".
"file.???"]"      Matches any file with three
characters after "."]
"file.?{3}"       Same as above.
"a \b"           Matches all strings that begin with
"a" and end with "b" and have an
arbitrary number of spaces in between
both.
"a +b"            Same as above, but at least one space
must be present.
"file.[bd]at"     Matches "file.bat" and "file.dat".
"[A-Za-z]+"      Matches any string containing only
letters.
"[:alpha:]++"     Same as above, using symbolic names.
"[^0-9]+"        Matches any string containing no
numbers, including the empty string.
"[:digit::lower:]" A single character, either a digit or
a lower case character.
"This is (very )+nice." Matches all strings with one or more
occurrences of "very " between
"This is " and "nice."

5.4.15.1. new (Class Method)

\[
\text{new(} \quad \text{pattern} \quad \text{MAXIMAL} \quad \text{MINIMAL} \quad \text{)}
\]

Instantiates a RegularExpression instance. Use the optional parameter pattern to define a regular expression pattern that will be used to match strings. You can select the type of regular expression matching to be “greedy” by specifying option MAXIMAL, or to be “lazy” by specifying option MINIMAL. Option MAXIMAL is the default.

Both pattern and match type can be changed with the parse method.

**Important**

RegularExpression defines its own init method. Any subclass which also defines its own init method, must forward to its superclass to complete object initialization. For details see Section 4.2.9, “Initialization”.

Example 5.269. RegularExpression class — new method

```plaintext
re1 = .RegularExpression~new
re2 = .RegularExpression~new("Hello?\*")
```

5.4.15.2. match

\[
\text{match(} \quad \text{string} \quad \text{)}
\]

This method tries to match string to the regular expression set by calls to the new or parse method.

With option MAXIMAL in effect, it will successfully match only if the whole string matches. With option MINIMAL in effect, any successful match will always start at the first character of string, but doesn’t necessarily have to cover the full string. Thus a match will always be a leading part of string.

Method match returns 0 for an unsuccessful match and 1 for a successful match.

Example 5.270. RegularExpression class — match method

```plaintext
str = "<p>Paragraph 1</p><p>Paragraph 2</p>"
re1 = .RegularExpression-new("<p>\?\*</p>", "MINIMAL")
re1-match(str)
re2 = .RegularExpression-new("<p>\?\*</p>", "MAXIMAL")
re2-match(str)

say "re1 (minimal) matched" str-substr(1, re1-position)
say "re2 (maximal) matched" str-substr(1, re2-position)
```
::requires "rxregexp.cls"

Output:

re1 (minimal) matched <p>Paragraph 1</p>
re2 (maximal) matched <p>Paragraph 1</p><p>Paragraph 2</p>

5.4.15.3. parse

Returns 0 after setting and successfully parsing the regular expression pattern. The new pattern will be used to match strings specified with methods match or pos. Returns an error code otherwise.

The type of regular expression matching can be set to “greedy” by specifying option MAXIMAL, or to “lazy” by specifying option MINIMAL. The default is to use the current matching type.

Return values:

0
   Regular expression was parsed successfully.

1
   An unexpected symbol was met during parsing.

2
   A missing ’)’ was found.

3
   An illegal set was defined.

4
   The regular expression ended unexpectedly.

5
   An illegal number was specified.

6
   An undefined symbolic set name was specified.

Example 5.271. RegularExpression class — parse method

```plaintext
patterns = "A [:alpha:]{4} fl?*.", -
"?*[lle]?></e?*[r|g]?"", -
"[invalid"
texts = "A nice flower.", -
"A yellow flower.", -
```
"A blue flag."

```perl
re = .RegularExpression~new
do pattern over patterns
    code = re~parse(pattern)
    if code == 0 then
        do text over texts
            say text~left(16) -
                re~match(text)?("matches", "doesn't match") "regex" pattern
        end
    else
        say "error" code "parsing pattern" pattern
        say
    end
end::requires rxregexp.cls
```

Output:

A nice flower. matches regex A [:alpha:][4] f1?.
A yellow flower. doesn't match regex A [:alpha:][4] f1?.
A blue flag. matches regex A [:alpha:][4] f1?.

A nice flower. matches regex ?*[l|e]?*e?[r|g]??
A yellow flower. matches regex ?*[l|e]?*e?[r|g]??
A blue flag. matches regex ?*[l|e]?*e?[r|g]??

error 3 parsing pattern [invalid]

Example 5.272. RegularExpression class — parse method

```perl
nrs = 1, 42, 0, 5436412, "1A", "f43g"
re = .RegularExpression~new("[1-9][0-9]*")
do nr over nrs
    say nr "is" re~match(nr)?("a valid", "an invalid") "number"
end
say
-- allow hexadecimal numbers and a single 0
re~parse("0|[1-9a-fA-F][:xdigit:]*")
do nr over nrs
    say nr "is" re~match(nr)?("a valid", "an invalid") "number"
end
::requires rxregexp.cls
```

Output:

1 is a valid number
42 is a valid number
0 is an invalid number
5436412 is a valid number
1A is an invalid number
f43g is an invalid number

1 is a valid number
42 is a valid number
0 is a valid number
5436412 is a valid number
5.4.15.4. pos

This method tries to locate the regular expression set by calls to the *new* or *parse* method in the given *haystack* string.

Method *pos* returns 0 for an unsuccessful match, or the starting position for a successful match. The end position of the match can be retrieved with the *position* method.

Example 5.273. RegularExpression class — haystack method

```rexx
text = "It's the year 2016!"
re = .RegularExpression~new("[1-9][0-9]*")
begin = re~pos(text)
if begin > 0 then
  do
    year = text~substr(begin, re~position - begin + 1)
    say "Found the number" year "in this sentence."
  end
::requires rxregexp.cls
```

Output:

```
Found the number 2016 in this sentence.
```

5.4.15.5. position

Returns the character position at which the last *parse*, *pos*, or *match* method ended.

Example 5.274. RegularExpression class — position method

```rexx
re = .RegularExpression-new
re-parse("[abc")                -- illegal set definition
say re-position                 -- will be 4

re = .RegularExpression-new("[abc]12")
re-match("c12")
say re-position                 -- will be 3

re-match("a13")                -- unsuccessful match
say re-position                 -- will be 2 (failure to match)
::requires "rxregexp.cls"
```

5.4.16. RexxContext Class
The **RexxContext** class gives access to context information about the currently executing Rexx code. Instances of the RexxContext class can only be obtained via the `.CONTEXT environment symbol`. They cannot be directly created by the user. It is a subclass of the **Object class**.

### Table 5.51. RexxContext Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td></td>
</tr>
<tr>
<td>Methods inherited from the <strong>Object class</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Class (Metaclass)</strong></td>
<td></td>
</tr>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
<td></td>
</tr>
<tr>
<td><strong>RexxContext</strong></td>
<td></td>
</tr>
<tr>
<td><code>args</code></td>
<td>form</td>
</tr>
<tr>
<td><code>condition</code></td>
<td>fuzz</td>
</tr>
<tr>
<td><code>digits</code></td>
<td>line</td>
</tr>
<tr>
<td><code>executable</code></td>
<td>name</td>
</tr>
<tr>
<td><code>package</code></td>
<td><code>rs</code></td>
</tr>
<tr>
<td><code>stackFrames</code></td>
<td>variables</td>
</tr>
</tbody>
</table>

#### 5.4.16.1. args

Returns the arguments used to invoke the current context as an array. This is equivalent to using the `ARG(1, 'A')` built-in function.

#### 5.4.16.2. condition

Returns the current context condition object, or `.nil` if the context does not currently have a trapped condition. This is equivalent to using the `CONDITION('O')` built-in function.

#### 5.4.16.3. digits

Returns the current context digits setting. This is equivalent to using the `DIGITS` built-in function.

#### 5.4.16.4. executable

Returns the current executable object for the current context. The executable will be either a **Routine object** or a **Method object**, depending on the type of the active context.

#### 5.4.16.5. form

Returns the current context form setting. This is equivalent to using the `FORM` built-in function.
5.4.16.6. fuzz

Returns the current context fuzz setting. This is equivalent to using the `FUZZ` built-in function.

5.4.16.7. line

Returns the context current execution line. This is equivalent to using the `.LINE` environment symbol.

5.4.16.8. name

Returns the name used to invoke the current context. If the current context is a method, `name` is the message name used to invoke the method. If the current context is a routine invoked as an external call, `name` is the name used to call the routine. If the current context is an internal routine call, `name` is the name of the label used to invoke the call. If the context is the main part of the program, `name` will be the package name.

5.4.16.9. package

Returns the `Package` object associated with the current executable object.

5.4.16.10. rs

Returns the context current return status value. If no host commands have been issued in the current context, `.nil` is returned. This is equivalent to using the `.RS` environment symbol.

5.4.16.11. stackFrames

Returns an Array of `StackFrame` objects representing the current call stack. The first item will represent the current Rexx context and subsequent elements are earlier elements in the call stack.

5.4.16.12. variables

Returns a directory object containing all of the variables in the current execution context. The directory keys will be the variable names and the mapped values are the values of the variables. The directory
will only contain simple variables and stem variables, but no compound variables. Compound variable values may be accessed by using the stem objects that are returned for the stem variable names.

See also Rexx Utility function *SysDumpVariables*.

Example 5.275. RexxContext class — variables method

```rxml
a = 2
b.1 = a
c. = .stem-new
c.["one"] = 11
dir = .Directory-new
dir["item"] = "index"
array = .Array-of("a", "e", "i")
say "SysDumpVariables:
"call SysDumpVariables
drop result
say ".context~variables:
variables = .context~variables
do name over variables
   say "Name="name"," Value="variables[name]"
   if name~right(1) = ".", variables[name]~isA(.Stem) then
      do tail over variables[name]
         say "Name="name||tail"," Value="variables[name][tail]"
      end
   end
end
```

will output

```
SysDumpVariables:
   Name=C.one, Value='11'
   Name=DIR, Value='a Directory'
   Name=B.1, Value='2'
   Name=ARRAY, Value='an Array'
   Name=A, Value='2'
   .context~variables:
   Name=ARRAY, Value='an Array'
   Name=B., Value='B.'
   Name=B.1, Value='2'
   Name=A, Value='2'
   Name=DIR, Value='a Directory'
   Name=C., Value='1'
   Name=C.one, Value='11'
```

5.4.17. RexxInfo Class

The **RexxInfo** class gives access to Rexx language information and other platform-specific information in a single place. Only one instance of the RexxInfo class can be obtained via the `.REXXINFO environment symbol`, other instances cannot be created or copied.

RexxInfo provides read-only attribute methods to get

- the default settings for NUMERIC digits, fuzz, and form,
- internalMaxNumber, internalMinNumber, and internalDigits, the allowed maximum, minimum, and the number of digits of internal whole numbers used e.g. as arguments to built-in functions, as the right-hand operand of the power (**) operator, or as the values of expr and exprf in a DO or LOOP instruction,
• **maxExponent** and **minExponent**, the maximum and minimum allowed exponent values of numbers in scientific notation,
• the language processor **version** and its sub-components **majorVersion**, **release**, and **modification**, and supplemental to that, the source code version control system's **revision**,
• the language processor's descriptive **name**, its **languageLevel**, and release **date**,
• the interpreter's **debug**, **platform**, addressing mode of its **architecture**, full **executable** path, shared/dynamic **libraryPath**, and whether the interpreter is a **debug** version,
• and the platform-specific values for **endofline**, **pathSeparator**, **directorySeparator**, **caseSensitiveFiles**, **maxPathLength**, and **maxArraySize**.

Table 5.52. RexxInfo Class

<table>
<thead>
<tr>
<th>Object</th>
<th>RexxxInfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from</td>
<td>architecture</td>
</tr>
<tr>
<td>the <strong>Object class</strong></td>
<td>internalDigits</td>
</tr>
<tr>
<td></td>
<td>internalMaxNumber</td>
</tr>
<tr>
<td></td>
<td>name</td>
</tr>
<tr>
<td>Class (Metaclass)</td>
<td>modification</td>
</tr>
<tr>
<td></td>
<td>package</td>
</tr>
<tr>
<td>Methods inherited from</td>
<td>caseSensitiveFiles</td>
</tr>
<tr>
<td>the <strong>Class class</strong></td>
<td>languageLevel</td>
</tr>
<tr>
<td></td>
<td>platform</td>
</tr>
<tr>
<td></td>
<td>digits</td>
</tr>
<tr>
<td></td>
<td>majorVersion</td>
</tr>
<tr>
<td></td>
<td>release</td>
</tr>
<tr>
<td></td>
<td>debug</td>
</tr>
<tr>
<td></td>
<td>directorySeparator</td>
</tr>
<tr>
<td></td>
<td>executable</td>
</tr>
<tr>
<td></td>
<td>form</td>
</tr>
<tr>
<td></td>
<td>fuzz</td>
</tr>
<tr>
<td></td>
<td>digits</td>
</tr>
<tr>
<td></td>
<td>languageLevel</td>
</tr>
<tr>
<td></td>
<td>pathSeparator</td>
</tr>
<tr>
<td></td>
<td>platform</td>
</tr>
<tr>
<td></td>
<td>majorVersion</td>
</tr>
<tr>
<td></td>
<td>release</td>
</tr>
<tr>
<td></td>
<td>revision</td>
</tr>
<tr>
<td></td>
<td>version</td>
</tr>
<tr>
<td></td>
<td>maxArraySize</td>
</tr>
<tr>
<td></td>
<td>maxExponent</td>
</tr>
<tr>
<td></td>
<td>maxPathLength</td>
</tr>
<tr>
<td></td>
<td>minExponent</td>
</tr>
</tbody>
</table>

### 5.4.17.1. architecture

Returns the interpreter's addressing mode, specified as a number in bits.

See also method **name**.

Example 5.276. RexxInfo class — architecture method

```plaintext
say .RexxxInfo-architecture -- 32 /* ooRexx 32-bit */
say .RexxxInfo-architecture -- 64 /* ooRexx 64-bit */
```

### 5.4.17.2. caseSensitiveFiles

Returns **true** if the file system is case-sensitive. Otherwise returns **false**.

On Unix-like systems where both case-sensitive and case-insensitive file systems may be mounted, this method returns **true** if the file system root "/" is case-sensitive, and otherwise **false**.
See also File class method isCaseSensitive which returns the same information.

### Example 5.277. RexxInfo class — caseSensitiveFiles method

```rxml
say .RexxInfo~caseSensitiveFiles    -- 0 /* e. g. Windows */
say .RexxInfo~caseSensitiveFiles    -- 1 /* e. g. Unix-like system */
```

#### 5.4.17.3. date

Returns the language processor release date as a string in the default format of the DATE built-in function.

See also
- method name
- keyword instruction PARSE VERSION,

which both also return this date as the third to fifth token of their result.

### Example 5.278. RexxInfo class — date method

```rxml
say .RexxInfo-date    -- 9 Dec 2016 /* e. g. */
```

#### 5.4.17.4. debug

Returns .true if the interpreter is a debug version, .false if it is a release version.

### Example 5.279. RexxInfo class — debug method

```rxml
say .RexxInfo-debug    -- 0 /* release version */, 1 /* debug version */
```

#### 5.4.17.5. digits

Returns the language processor default for the NUMERIC DIGITS setting.

See also
- Package method digits,
- RexxContext method digits, and
- built-in function DIGITS.

### Example 5.280. RexxInfo class — digits method

```rxml
say .RexxInfo-digits    -- 9
```
5.4.17.6. directorySeparator

Returns the platform-specific directory separator used by the file system, "\\" on Windows, and "/'" on Unix-like systems.

See also File method separator, which provides the same information.

Example 5.281. RexxInfo class — directorySeparator method

```plaintext
say .RexxInfo~directorySeparator    -- "\\" /* Windows */
say .RexxInfo~directorySeparator    -- "/" /* Unix-like system */
```

5.4.17.7. endofline

Returns a platform-specific string representing the line-end characters. It returns '0d 0a'x (carriage-return, line-feed) on Windows, and '0a'x (line-feed) on Unix-like systems.

See also the environment symbol .ENDOFLINE, which provides the same information.

Example 5.282. RexxInfo class — endofline method

```plaintext
say .RexxInfo~endofline    -- '0d 0a'x /* Windows */
say .RexxInfo~endofline    -- '0a'x /* Unix-like system */
```

5.4.17.8. executable

Returns a new File instance of the full path of the currently executing interpreter.

See also method libraryPath.

Example 5.283. RexxInfo class — executable method

```plaintext
say .RexxInfo~executable    -- (Windows e.g.) C:\Program Files\ooRexx\rexx.exe
say .RexxInfo~executable    -- (Unix-like system e.g.) /usr/local/bin/rexx
```

5.4.17.9. form

Returns the language processor default for the NUMERIC FORM setting.
See also
- **Package** method *form*,
- **RexxContext** method *form*, and
- built-in function *FORM*.

### Example 5.284. RexxInfo class — form method
```
say .RexxInfo~form    -- SCIENTIFIC
```

5.4.17.10. fuzz

Returns the language processor default for the NUMERIC FUZZ setting.

See also
- **Package** method *fuzz*,
- **RexxContext** method *fuzz*, and
- built-in function *FUZZ*.

### Example 5.285. RexxInfo class — fuzz method
```
say .RexxInfo~fuzz    -- 0
```

5.4.17.11. internalDigits

Returns the NUMERIC DIGITS setting that the built-in functions use internally, which is 9 digits, when running in 32-bit addressing mode, and 18 digits, when running in a 64-bit addressing mode.

See also method *name*.

### Example 5.286. RexxInfo class — internalDigits method
```
say .RexxInfo~internalDigits    -- 9 /* 32-bit addressing mode */
say .RexxInfo~internalDigits    -- 18 /* 64-bit addressing mode */
```

5.4.17.12. internalMaxNumber

Returns the maximum allowed value for internal whole numbers used e.g. as arguments to built-in functions, as the right-hand operand of the power (**) operator, or as the values of *expr* and *exprf* in a DO or LOOP instruction.

See also methods *internalMinNumber* and *internalDigits*. 
5.4.17.13. internalMinNumber

Returns the minimum allowed value for internal whole numbers used e.g. as arguments to built-in functions, as the right-hand operand of the power (** operator), or as the values of exprr, and exprf, in a DO or LOOP instruction.

See also methods internalMaxNumber and internalDigits.

5.4.17.14. languageLevel

Returns the language processor's language level as a decimal number.

See also
• method name and
• keyword instruction PARSE VERSION,

which both also return the language level as the second token of their result.

5.4.17.15. libraryPath

Returns a new File instance of the path of the shared or dynamic libraries for the currently running Rexx interpreter.

See also method executable.

Example 5.290. RexxInfo class — libraryPath method

```plaintext
say .RexxInfo~libraryPath    -- (Windows e.g.) C:\Program Files\ooRexx
```
5.4.17.16. majorVersion

Returns the major version number of the language processor. The major version number is the first part of the language processor version.

See also
- methods release and modification, and
- keyword instruction PARSE VERSION.

Example 5.291. RexxInfo class — majorVersion method

```plaintext
say .RexxInfo~majorVersion    -- 5 /* e.g. ooRexx 5.0.0 */
```

5.4.17.17. maxArraySize

Returns the implementation-defined maximum allocation size allowed for Array and Queue.

Example 5.292. RexxInfo class — maxArraySize method

```plaintext
say .RexxInfo~maxArraySize  -- 100000000          /* 32-bit addressing mode */
say .RexxInfo~maxArraySize  -- 100000000000000000 /* 64-bit addressing mode */
```

5.4.17.18. maxExponent

Returns the maximum allowed exponent value of a number in scientific notation.

See also method minExponent.

Example 5.293. RexxInfo class — maxExponent method

```plaintext
say .RexxInfo~maxExponent    -- 999999999
```

5.4.17.19. maxPathLength

Returns the maximum allowed file system path length of a fully qualified path including any filename portion.
### 5.4.17.20. minExponent

Returns the minimum allowed exponent value of a number in scientific notation.

See also method `maxExponent`.

#### Example 5.295. RexxInfo class — minExponent method

```rexx
say .RexxInfo~minExponent   -- -999999999
```

### 5.4.17.21. modification

Returns the modification number of the language processor. The modification number is the third part of the language processor `version` string.

See also
• methods `majorVersion` and `release`, and
• keyword instruction `PARSE VERSION`.

#### Example 5.296. RexxInfo class — modification method

```rexx
say .RexxInfo~modification    -- 0 /* e. g. ooRexx 5.0.0 */
```

### 5.4.17.22. name

Returns the language processor's descriptive name in the format that is also returned by `PARSE VERSION`, which is
• an interpreter identification string (that includes `version` and `architecture`),
• the interpreter `languageLevel` and
• the interpreter build `date`.

See also methods `version` and `languageLevel`, and keyword instruction `PARSE VERSION`.

#### Example 5.297. RexxInfo class — name method

```rexx
say .RexxInfo~name    -- REXX-ooRexx_5.0.0(MT)_64-bit 6.05 22 Dec 2018 /* e. g. */
```
5.4.17.23. package

Returns a Package instance of all Rexx-defined (namespace "REXX") classes.

See Package class.

Example 5.298. RexxInfo class — package method

```plaintext
say .RexxInfo~package~publicClasses~items -
"Rexx-defined public classes"   -- 56 Rexx-defined public classes /* e. g. */
```

5.4.17.24. pathSeparator

Returns the platform-specific path separator used for file search paths, ";" on Windows, and ":" on Unix-like systems.

See also File method pathSeparator.

Example 5.299. RexxInfo class — pathSeparator method

```plaintext
say .RexxInfo~pathSeparator    -- ";" (Windows)
say .RexxInfo~pathSeparator    -- ":" (Unix-like system)
```

5.4.17.25. platform

Returns the name of the interpreter operating system as a string, e. g. WindowsNT for a Windows, or LINUX for a Linux system.

See also keyword instruction PARSE SOURCE, which returns the same information as its first token.

Example 5.300. RexxInfo class — platform method

```plaintext
say .RexxInfo~platform   -- WindowsNT
say .RexxInfo~platform   -- LINUX
```

5.4.17.26. release

Returns the release number of the language processor. The release number is the second part of the language processor version string.
RexxQueue Class

See also
• methods majorVersion and modification, and
• keyword instruction PARSE VERSION.

Example 5.301. RexxInfo class — release method
```
say .RexxInfo~release    -- 0 /* e. g. ooRexx 5.0.0 */
```

5.4.17.27. revision

Returns the source code version control system's revision from which this language processor was built. Rexx code should not rely on the format returned.

See also keyword instruction PARSE VERSION.

Example 5.302. RexxInfo class — revision method
```
say .RexxInfo~revision    -- 11636 /* e. g. */
```

5.4.17.28. version

Returns the version number of the language processor. The version number consists of the majorVersion, followed by a decimal point, the release, followed by another decimal point, and the modification.

Example 5.303. RexxInfo class — version method
```
say .RexxInfo~version    -- 5.0.0 /* e. g. */
```

5.4.18. RexxQueue Class

The RexxQueue class provides object-style access to Rexx external data queues.

Table 5.53. RexxQueue Class

<table>
<thead>
<tr>
<th>Object</th>
<th>Class (Metaclass)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Object class</td>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td></td>
<td>RexxQueue</td>
</tr>
<tr>
<td>new (Inherited Class Method)</td>
<td></td>
</tr>
<tr>
<td>create (Class Method)</td>
<td></td>
</tr>
<tr>
<td>delete (Class Method)</td>
<td></td>
</tr>
</tbody>
</table>
RexxQueue Class

### 5.4.18.1. new

Returns a new RexxQueue instance associated with the Rexx external data queue named `name`.

If `name` is omitted, the SESSION queue is used. If `name` is specified as `.nil` the new RexxQueue instance is associated with a newly created Rexx external data queue with a system-generated name.

If `name` is specified and an external data queue named `name` exists, it is opened. If such a data queue does not exist, it is created.

Example 5.304. RexxQueue class — new method

```rexx
q = .RexxQueue~new
q~queue("line")
q~push("header")
say q~queued q~pull  -- 2 header
q~delete
```

### 5.4.18.2. create (Class Method)

Returns the name of a newly created Rexx external data queue, which will be either `name` if no external data queue named `name` already exists, or, a system-generated name if `name` is omitted or the external data queue `name` already exists.

See also method open (Class Method).

Example 5.305. RexxQueue class — create method

```rexx
-- no MYQUEUE queue exists
say .RexxQueue~create("myqueue")  -- MYQUEUE
-- MYQUEUE queue already exists
say .RexxQueue~create("myqueue")  -- S0000000000004B48Q00001E1BEE66F20 (e. g.)
```

### 5.4.18.3. delete (Class Method)
Returns 0 if the external Rexx named queue named name was successfully deleted. Non-zero results are the error codes from the RexxDeleteQueue() programming interface.

### 5.4.18.4. exists (Class Method)

```plaintext
effects(name)
```

Returns `.true.` if the external data queue name currently exists, otherwise returns `.false.`.

**Example 5.306. RexxQueue class — exists method**

```
say .RexxQueue~exists("SESSION")    -- 1
```

### 5.4.18.5. open (Class Method)

```plaintext
open(name)
```

Returns 0 if the external Rexx named queue name exists, or, if it doesn't exist, can be successfully created. Non-zero results are the error codes from the RexxOpenQueue() programming interface.

### 5.4.18.6. delete

```plaintext
delete
```

Returns 0 if the external Rexx named queue associated with this RexxQueue instance was successfully deleted. Non-zero results are the error codes from the RexxDeleteQueue() programming interface.

Calling the `lineOut, pull, push, queue, or say` method of a RexxQueue instance with a deleted external data queue will result in an error.

### 5.4.18.7. empty

```plaintext
empty
```

Removes all items from the Rexx external queue associated with this RexxQueue instance. Returns 0 upon success, non-zero results are the error codes from the RexxClearQueue() programming interface.

### 5.4.18.8. get

```plaintext
get
```

Returns the name of the Rexx external queue associated with this instance.

### 5.4.18.9. init
Important

**RexxQueue** defines its own `init` method. Any subclass which also defines its own `init` method, must forward to its superclass to complete object initialization. For details see Section 4.2.9, “Initialization”.

5.4.18.10. lineIn

Reads a single line from the Rexx external queue. If the queue is empty, `lineIn` will wait until a line is added to the queue.

5.4.18.11. lineOut

Adds a line to the Rexx external queue in first-in-first-out (FIFO) order. If `line` is not specified, a null string ("") is added.

5.4.18.12. makeArray

Returns a single-index array with the same number of items as the receiver object. Items in the new array will have the same order as the items in the external queue. The external queue is emptied.

5.4.18.13. pull

Reads a line from the Rexx external queue. If the queue is currently empty, this method will immediately return `.nil` without waiting for lines to be added to the queue.

5.4.18.14. push

Adds a line to the Rexx external queue in last-in-first-out (LIFO) order. If `line` is not specified, a null string ("") is added.
5.4.18.15. queue

Adds a line to the Rexx external queue in first-in-first-out (FIFO) order. If line is not specified, a null string ("") is added.

5.4.18.16. queued

Returns the count of lines currently in the Rexx external queue.

5.4.18.17. say

Adds a line to the Rexx external queue in first-in-first-out (FIFO) order. If line is not specified, a null string ("") is added.

5.4.18.18. set

Switches the Rexx external queue associated with the RexxQueue instance. The new queue must have been previously created. The method return value is the name of current queue being used by the instance.

5.4.19. Singleton Class (Metaclass)

The Singleton class is a metaclass (a subclass of the ooRexx metaclass Class) which can be used for any ooRexx class, if the programmer wishes only a single instance of a class to be created. In this case the class directive needs to denote Singleton in its METACLASS option.

This class is defined as a MXIN class.

<table>
<thead>
<tr>
<th>Table 5.54. Singleton Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
</tr>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>Singleton</td>
</tr>
<tr>
<td>new (Class Method)</td>
</tr>
</tbody>
</table>
5.4.19.1. new (Class Method)

This metaclass makes sure that only a single instance of a class can be created, a singleton. After a singleton got created each time a NEW message gets sent to the class will cause that singleton to be returned. The metaclass Singleton makes also sure that the singleton object will return itself upon receiving the COPY message by defining a proper COPY method for the singleton.

Example 5.307. Employing the Singleton (meta)class

The following program defines a class TEST which makes sure that each instance has a unique number in its instance attribute nr. The TESTSINGLETON subclass uses the METACLASS Singleton option to make sure that it only creates a single instance, a singleton. The program will loop over the classes TEST and TESTSINGLETON, creates three instances of each and displays the value of the instance attribute nr and the identityHash value of each instance, which uniquely identifies each each instance of a class (cf. the MAKESTRING definition).

```rexx
#!/usr/bin/env rexx
do clz over .test, .testSingleton   -- iterate over the two classes
 rounds=3
 say "creating" rounds "objects of type:" clz
 do i=1 to rounds
   say "  round ": i": " clz~new -- create new instance
   do say end

 (;;)=====================================================================
/** This Test class counts the number of instances that get created for it. */
::class Test
 (;;)------------- class method and class attribute definitions -------------
::method init class  -- class constructor
 expose counter
  counter=0          -- make sure attribute is initialized to 0
::attribute counter get private class -- getter method that increases counter
 expose counter
  counter+=1         -- increase counter by 1
::method makestring  -- a string representation of the object
 expose nr          -- expose attribute
  return "a" self~class-id"[nr="nr",identityHash="self~identityHash"]"
 (;;)------------- instance method and instance attribute definitions --------
::attribute nr get -- getter method
 expose nr          -- expose attribute
 namespace self~class-counter -- new instance: fetch new counter from class and save it
::method makestring  -- a string representation of the object
 expose nr          -- expose attribute
  return "a" self~class-id"[nr="nr",identityHash="self~identityHash"]"
 (;;)=====================================================================
/** This class makes sure that only a single instance of it gets created by */
** using Singleton as its metaclass.
::class TestSingleton subclass Test metaclass Singleton
```

Output of running the above program (the hash values may differ on each run):

```
creating 3 objects of type: The TEST class
round # 1: a TEST[nr=1,identityHash=-49885937]
round # 2: a TEST[nr=2,identityHash=-49889489]
```
As can be seen from the output there are three distinct instances of the class TEST, however the three instances of the class TESTSINGLETON are identical (cf. the values of nr and identityHash).

5.4.20. StackFrame Class

The StackFrame class gives access to execution information about an executing Rexx activity. Instances of the StackFrame class can be obtained via the .CONTEXT environment symbol or from a condition object created for a trapped condition. A StackFrame instance represents an instance of an execution unit on the current activity's call stack. It provides information on code location, arguments, etc. that are useful for debugging and problem determination. StackFrame instances cannot be directly created by the user.

Table 5.55. StackFrame Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
<tr>
<td>StackFrame</td>
</tr>
<tr>
<td>arguments</td>
</tr>
<tr>
<td>executable</td>
</tr>
<tr>
<td>line</td>
</tr>
<tr>
<td>makeString</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>string</td>
</tr>
<tr>
<td>target</td>
</tr>
<tr>
<td>traceLine</td>
</tr>
<tr>
<td>type</td>
</tr>
</tbody>
</table>

5.4.20.1. arguments

Returns the arguments used to invoke the execution frame represented by the StackFrame instance.

5.4.20.2. executable

Returns the executable object for the StackFrame context. The executable will be either a Routine object or a Method object, depending on the type of the StackFrame element.

5.4.20.3. line

Returns the current line number of the StackFrame context. If the frame instance is not Rexx code (e.g. compiled native code), .nil is returned.
5.4.20.4. makeString

Returns the frame `traceLine` value as the object's string value.

5.4.20.5. name

Returns the name associated with the stack frame context. The name meaning depends on the type of context.

- If the stack frame context is a method invocation, `name` is the message name used to invoke the method.
- If the stack frame context is a routine invocation, `name` is the routine.
- If the stack frame context is an internal routine invocation, `name` is the label name used to invoke the internal routine.
- If the stack frame context is the initial main part of a Rexx program, `name` is the name of the file containing the program.
- If the stack frame context is an INTERPRET instruction, a zero-length string is returned.

5.4.20.6. string

Returns the frame `traceLine` value as the object's string value.

5.4.20.7. target

Returns the object the method was invoked against, if the StackFrame `type` is **METHOD**. Returns `.nil` for all other StackFrame types.

5.4.20.8. traceLine

Returns the trace back line that would be displayed for error message trace. When possible, this will be the source line of the Rexx code in the call stack. If source is not available, the trace back will identify the method or routine belonging to the stack frame.

5.4.20.9. type
StreamSupplier Class

Returns the type of invocation for this stack frame. Possible values are:

**PROGRAM**

The stack frame is the top level of a program.

**METHOD**

The stack frame is a method invocation.

**ROUTINE**

The stack frame is a routine invocation.

**INTERPRET**

The stack frame is code created by an INTERPRET instruction.

**INTERNALCALL**

The stack frame is a subroutine or function call to an internal label.

**COMPILE**

The stack frame for compiling Rexx code for execution. Many syntax errors will be reported by a COMPILE frame.

**Example 5.308. StackFrame class — type method**

``` Rexx
interpret "call level2" 21
::routine level2
call level3 31, 32
return

level3: procedure
  signal on syntax
  .Method~new("","--")
  return

syntax:
do f over condition("o")"STACKFRAMES"
say (f~type f~name"("f~arguments~makeString(,"",")")~left(27) f~line": " -
f~traceLine~strip
end
```

may output

```
COMPILE ()                  1: 1 *-* ~~
METHOD NEW(,--)             The NIL object: *-* Compiled method "NEW" with scope "Method".
INTERNALCALL LEVEL3(31,32) 9: 9 *-* .Method~new("","--")
ROUTINE LEVEL2(21)          4: 4 *-* call level3 31, 32
INTERPRET ()                1: 1 *-* call level2 21
PROGRAM C:\\stackFrame.rex() 1: 1 *-* interpret "call level2" 21
```

**5.4.21. StreamSupplier Class**

A subclass of the **Supplier Class** that provides stream lines using supplier semantics. This allows the programmer to iterate over the remaining lines in a stream. A StreamSupplier object provides a snapshot of the stream at the point in time it is created, including the current line read position. In general, the iteration is not affected by later changes to the read and write positioning of the stream. However, forces external to the iteration may change the content of the remaining lines as the iteration progresses.
5.4.21.1. available

Returns \texttt{true} if an item is available from the supplier (that is, if the \texttt{item} method would return a value). It returns \texttt{false} if the collection is empty or the supplier has already enumerated the entire collection.

5.4.21.2. index

Returns the index of the current item in the collection. If no item is available, that is, if \texttt{available} would return \texttt{false}, the supplier raises an error.

5.4.21.3. init

Initializes the object instance.

5.4.21.4. item

Returns the current item in the collection. If no item is available, that is, if \texttt{available} would return \texttt{false}, the supplier raises an error.

5.4.21.5. next
Moves to the next item in the collection. By repeatedly sending `next` to the supplier (as long as `available` returns `.true`), you can enumerate all items in the collection. If no item is available, that is, if `available` would return `.false`, the supplier raises an error.

## 5.4.22. Supplier Class

A Supplier object is an iterator that allows the enumeration of an items Collection together with an indexes Collection. All objects inheriting from Collection Class provide a supplier method, that returns a snapshot of the Collection as a Supplier object, which allows iteration using the DO/LOOP WITH instruction. The iteration results are not affected by later changes to the source Collection object.

<table>
<thead>
<tr>
<th>Table 5.57. Supplier Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
</tr>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
<tr>
<td>Supplier</td>
</tr>
<tr>
<td>new (Class Method)</td>
</tr>
<tr>
<td>allIndexes</td>
</tr>
<tr>
<td>allItems</td>
</tr>
<tr>
<td>available</td>
</tr>
</tbody>
</table>

### 5.4.22.1. new (Class Method)

Returns a new supplier object. The `items` argument must be an array of objects over which the supplier iterates. The `indexes` argument is an array of index values with a one-to-one correspondence to the objects contained in the `items` array. The created supplier iterates over the arrays, returning elements of the values array in response to `items` messages, and elements of the indexes array in response to `index` messages. The supplier iterates for the number of items contained in the values array, returning `.nil` for any nonexistent items in either array.

### 5.4.22.2. allIndexes

Returns an array of all index values from the current supplier position to the end of the supplier. Once allIndexes is called, no additional items can be retrieved from the supplier. Calls to `available` will return `.false`.

### 5.4.22.3. allItems
Returns an array of all items from the current supplier position to the end of the supplier. Once allItems is called, no additional items can be retrieved from the supplier. Calls to available will return .false.

5.4.22.4. available

Returns .true if an item is available from the supplier (that is, if the item method would return a value). It returns .false if the collection is empty or the supplier has already enumerated the entire collection.

5.4.22.5. index

Returns the index of the current item in the collection. If no item is available, that is, if available would return .false, the supplier raises an error.

5.4.22.6. item

Returns the current item in the collection. If no item is available, that is, if available would return .false, the supplier raises an error.

5.4.22.7. next

Moves to the next item in the collection. By repeatedly sending next to the supplier (as long as available returns .true), you can enumerate all items in the collection. If no item is available, that is, if available would return .false, the supplier raises an error.

5.4.22.8. Examples

Example 5.309. Supplier class

desserts=.array-of(apples, peaches, pumpkins, 3.14159) /* Creates array */
say "The desserts we have are:"
baker=desserts-supplier /* Creates supplier object named BAKER */
do while baker-available /* Array suppliers are sequenced */
  if baker-index=4
    then say baker-item "is pi, not pie!!"
  else say baker-item
    baker-next
end

/* Produces: */
/* The desserts we have are: */
/* APPLES */
5.4.22.9. supplier

Returns the target supplier as a result. This method allows an existing supplier to be passed to methods that expect an object that implements a supplier method as an argument.

5.4.23. Ticker Class

The **Ticker** class provides a repeating notification capability by sending a notification message to a notification target each trigger interval.

A Ticker object can be cancelled any time. If cancelled, a cancel notification message will be sent to the notification target.

<table>
<thead>
<tr>
<th>Table 5.58. Ticker Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
</tr>
<tr>
<td>Methods inherited from the <strong>Object class</strong></td>
</tr>
<tr>
<td><strong>Class (Metaclass)</strong></td>
</tr>
<tr>
<td>Methods inherited from the <strong>Class class</strong></td>
</tr>
<tr>
<td><strong>Ticker</strong></td>
</tr>
<tr>
<td>attachment</td>
</tr>
<tr>
<td>cancel</td>
</tr>
</tbody>
</table>

5.4.23.1. attachment

Returns the object that has been attached to the Ticker instance upon creation. Returns `.nil` if no object is attached.

See method `init`.

Example 5.310. Ticker class — attachment method

```
eachSecond = .Ticker~new(1, .Target~new, "once each second")
call SysSleep 1.5                      -- will trigger once
eachSecond~cancel
::class Target inherit AlarmNotification
::method triggered                     -- called each interval
use arg ticker
say ticker~attachment                -- displays "once each second"
```

5.4.23.2. cancel

```
Cancels the Ticker represented by the receiver.

See also method canceled/cancelled.

Example 5.311. Ticker class — cancel method

```pascal
twiceASecond = .Ticker~new(0.5, .Target~new)
call SysSleep 0.25 -- too short for Ticker
twiceASecond~cancel -- "ticker cancelled"

::class Target inherit AlarmNotification
::method triggered -- called each interval
  say "ticker triggered"
::method cancel -- called when cancelled
  say "ticker cancelled"
```

5.4.23.3. canceled/cancelled

Returns .true if the ticker has been cancelled. Returns .false otherwise.

See also method cancel.

Example 5.312. Ticker class — canceled/cancelled method

```pascal
tick = .Ticker~new(0.5, .message~new(.stdout, "SAY", "I", "knock, knock"))
call SysSleep 0.75 -- knock, knock
say "Ticker" tick~cancelled~?('cancelled', 'not cancelled') -- Ticker not cancelled
call SysSleep 0.5
 tick~cancel
say "Ticker" tick~cancelled~?('cancelled', 'not cancelled') -- Ticker cancelled
```

5.4.23.4. init

Sets up a Ticker with a specified interval. After each interval, the Ticker sends a message triggered to the specified notification target.

The target must be an object that implements the AlarmNotification interface. It must inherit from or be a subclass of the AlarmNotification class, or a Message object (as the Message class inherits from AlarmNotification). If target is a Message object, the triggered method of the Message class will respond by simply sending the specified message.

The interval can be a TimeSpan or a String object. If it is

- a TimeSpan, it must be of a non-negative length, which specifies the interval time length.
- a String, it must be a non-negative number which specifies the interval time length in seconds.

If specified, attachment can be an arbitrary object that will be attached to the Ticker instance, and can later be retrieved in the event handler. See method attachment.
You can use the `cancel` method at any time to cancel a Ticker. If cancelled, the Ticker sends message `cancel` to the specified notification `target`.

The following code uses a Ticker to display progress information during a long-running task.

**Example 5.313. Ticker class — init method**

```plaintext
.. Progress-new(0.5)-monitor(.Task-new, "tenSeconds")
-- defines tasks together with their "progress" methods
::class Task

-- long-running task
::method tenSeconds unguarded
  expose m n
  n = 1000
  do m = 1 to n
    call SysSleep 0.01
  end
  return

-- returns running value to be displayed as progress
::method "tenSeconds-progress" un guarded
  expose m n
  return m"/"n "steps done"

-- runs task while displaying progress text at each interval
::class Progress inherit AlarmNotification

-- set progress interval
::method init un guarded
  expose interval
  use strict arg interval = 1

-- starts ticker, runs task, and returns task result
-- (requires names of task and progress methods)
::method monitor
  expose interval object progress
  use strict arg object, task, progress = (task"-progress")
  tick = .Ticker-new(interval, self)
  object-send(task)
  tick-cancel

-- displays progress text at each ticker interval
::method triggered un guarded
  expose object progress
  .stdout-charOut(object-send(progress) '0d'x)
```

### 5.4.23.5. interval

**interval**

Returns a `TimeSpan` object representing the interval to which the Ticker has been set.

See also method `init`.

**Example 5.314. Ticker class — interval method**

```plaintext
say .Ticker-new(.TimeSpan-fromSeconds(60), -
  .Message-new(.stdout, "say", "i", -
```
5.4.24. TimeSpan Class

A `TimeSpan` object represents a time interval with microsecond resolution. The interval may be positive or negative, with a maximum duration of 3652059 days less one microsecond (which approximately equals 9999 years less one microsecond). A `TimeSpan` object has methods to allow retrieving components like days or seconds, adding to components, as well as allowing arithmetic operations between `TimeSpan` objects.

Table 5.59. TimeSpan Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the <code>Object</code> class</td>
</tr>
<tr>
<td>Class (MetaClass)</td>
</tr>
<tr>
<td>Methods inherited from the <code>Class</code> class</td>
</tr>
<tr>
<td>+ Orderable (Mixin Class)</td>
</tr>
<tr>
<td>Comparison Methods: `== &lt; &lt;= &lt;&lt; &lt;= &lt;&gt; &gt;= &gt;=</td>
</tr>
<tr>
<td>compareTo</td>
</tr>
<tr>
<td>+ Comparable (Mixin Class)</td>
</tr>
<tr>
<td>Methods inherited from the <code>Comparable</code> class</td>
</tr>
<tr>
<td><code>compareTo</code></td>
</tr>
<tr>
<td><code>TimeSpan</code></td>
</tr>
<tr>
<td><code>new</code> (Inherited Class Method)</td>
</tr>
<tr>
<td><code>fromCivilTime</code> (Class Method)</td>
</tr>
<tr>
<td><code>fromDays</code> (Class Method)</td>
</tr>
<tr>
<td><code>fromHours</code> (Class Method)</td>
</tr>
<tr>
<td><code>fromMinutes</code> (Class Method)</td>
</tr>
<tr>
<td><code>fromNormalTime</code> (Class Method)</td>
</tr>
<tr>
<td><code>fromSeconds</code> (Class Method)</td>
</tr>
<tr>
<td><code>fromStringFormat</code> (Class Method)</td>
</tr>
<tr>
<td>Arithmetic Methods: <code>+ - * / // %</code></td>
</tr>
<tr>
<td><code>addDays</code></td>
</tr>
<tr>
<td><code>addHours</code></td>
</tr>
<tr>
<td><code>addMicroseconds</code></td>
</tr>
<tr>
<td><code>addMinutes</code></td>
</tr>
<tr>
<td><code>addSeconds</code></td>
</tr>
<tr>
<td><code>addWeeks</code></td>
</tr>
<tr>
<td><code>compareTo</code></td>
</tr>
<tr>
<td><code>duration</code></td>
</tr>
<tr>
<td><code>hours</code></td>
</tr>
<tr>
<td><code>init</code></td>
</tr>
<tr>
<td><code>makeString</code></td>
</tr>
<tr>
<td><code>microseconds</code></td>
</tr>
<tr>
<td><code>minutes</code></td>
</tr>
<tr>
<td><code>sign</code></td>
</tr>
<tr>
<td><code>string</code></td>
</tr>
<tr>
<td><code>totalDays</code></td>
</tr>
<tr>
<td><code>totalHours</code></td>
</tr>
<tr>
<td><code>totalMicroseconds</code></td>
</tr>
<tr>
<td><code>totalMinutes</code></td>
</tr>
<tr>
<td><code>totalSeconds</code></td>
</tr>
</tbody>
</table>

5.4.24.1. fromDays (Class Method)

`fromDays(days)`

Creates a `TimeSpan` object from a number of days. The `days` argument must be a valid Rexx number.

5.4.24.2. fromHours (Class Method)
5.4.24.3. fromMinutes (Class Method)

Creates a TimeSpan object from a number of minutes. The minutes argument must be a valid Rexx number.

5.4.24.4. fromSeconds (Class Method)

Creates a TimeSpan object from a number of seconds. The seconds argument must be a valid Rexx number.

5.4.24.5. fromMicroseconds (Class Method)

Creates a TimeSpan object from a number of microseconds. The microseconds argument must be a valid Rexx number.

5.4.24.6. fromNormalTime (Class Method)

Creates a TimeSpan object from a string returned by the Normal option of the TIME built-in function (hh:mm:ss). The TimeSpan will contain an interval equal to the time of day represented by the string.

5.4.24.7. fromCivilTime (Class Method)

Creates a TimeSpan object from a string returned by the Civil option of the TIME built-in function (hh:mmxx). The TimeSpan will contain an interval equal to the time of day represented by the string.

5.4.24.8. fromLongTime (Class Method)

Creates a TimeSpan object from a string returned by the Long option of the TIME built-in function (hh:mm:ss.uuuuuu). The TimeSpan will contain an interval equal to the time of day represented by the string.
5.4.24.9. fromStringFormat (Class Method)

Creates a TimeSpan object from a string in the format returned by the TimeSpan `string` method.

5.4.24.10. init

Initializes a new TimeSpan instance. If the single `fullDate` argument is used, the TimeSpan argument is initialized to the time span `fulldate` microseconds. Otherwise, the TimeSpan instance is initialized to either the `hours`, `minutes`, and `seconds` or the `days`, `hours`, `minutes`, `seconds`, and `microseconds` components. Each of these components must be a valid whole number within the acceptable range for the given component. For example, `hours` must be in the range 0-23, while `minutes` must be in the range 0-59.

Example 5.315. TimeSpan class

```
span = .TimeSpan~new(15, 37, 30)                  -- 15:37:30.000000
span = .TimeSpan~new(6, 4, 33, 15, 100)           -- 6.04:33:15.000100
```

5.4.24.11. Arithmetic Methods

Returns the result of performing the specified arithmetic operation on the receiver TimeSpan object. Depending on the operation, the `argument` be either a TimeSpan object, a DateTime object, or a number. See the description of the individual operations for details. The `arithmetic_operator` can be:

- Addition. If `argument` is a DateTime object, the TimeSpan is added to the DateTime object, returning a new DateTime instance. Neither the receiver TimeSpan or the argument DateTime object is altered by this operation. The TimeSpan may be either positive or negative.
If `argument` is a TimeSpan object, the two TimeSpans are added together, and a new TimeSpan instance is returned. Neither the TimeSpan object is altered by this operation.

- **Subtraction.** The `argument` must be a TimeSpan object. The argument TimeSpan is subtracted from the receiver TimeSpan and a new TimeSpan instance is returned. Neither the TimeSpan object is altered by this operation.

* **Multiplication.** The `argument` must be a valid Rexx number. The TimeSpan is multiplied by the `argument` value, and a new TimeSpan instance is returned. The receiver TimeSpan object is not altered by this operation.

/ **Division.** The `argument` must be a valid Rexx number. The TimeSpan is divided by the `argument` value, and a new TimeSpan instance is returned. The receiver TimeSpan object is not altered by this operation. The `/` operator and `%` produce the same result.

% **Integer Division.** The `argument` must be a valid Rexx number. The TimeSpan is divided by the `argument` value, and a new TimeSpan instance is returned. The receiver TimeSpan object is not altered by this operation. The `/` operator and `%` produce the same result.

// **Remainder Division.** The `argument` must be a valid Rexx number. The TimeSpan is divided by the `argument` value and the division remainder is returned as a new TimeSpan instance. The receiver TimeSpan object is not altered by this operation.

Prefix - The TimeSpan is negated, returning a new TimeSpan instance. The receiver TimeSpan is not altered by this operation.

Prefix + Returns a new instance of the TimeSpan object with the same time value.

---

Example 5.316. TimeSpan class

```rexx
t1 = .timespan~fromHours(1)
t2 = t1 * 2
-- displays "01:00:00.000000 02:00:00.000000 03:00:00.000000"
say t1 t2 (t1 + t2)
```

5.4.24.12. `compareTo`

This method returns -1 if the `other` is larger than the receiving object, 0 if the two objects are equal, and 1 if `other` is smaller than the receiving object.

5.4.24.13. `duration`

Returns a new TimeSpan object containing the absolute value of the receiver TimeSpan object.

5.4.24.14. `days`

---
5.4.24.15. **hours**

Returns the hours component of the TimeSpan, as a positive number.

5.4.24.16. **minutes**

Returns the minutes component of the TimeSpan, as a positive number.

5.4.24.17. **seconds**

Returns the seconds component of the TimeSpan, as a positive number.

5.4.24.18. **makeString**

Returns the time span formatted as a string in the format `-ddddddd.hh:mm:ss.uuuuu`. This is an alias of the `string` method.

5.4.24.19. **microseconds**

Returns the microseconds component of the TimeSpan, as a positive number.

5.4.24.20. **totalDays**

Returns the time span expressed as a number of days. The result includes any fractional part and retains the sign of the receiver TimeSpan.
Returns the time span expressed as a number of hours. The result includes any fractional part and retains the sign of the receiver TimeSpan.

5.4.24.22. totalMinutes

Returns the time span expressed as a number of minutes. The result includes any fractional part and retains the sign of the receiver TimeSpan.

5.4.24.23. totalSeconds

Returns the time span expressed as a number of seconds. The result includes any fractional part and retains the sign of the receiver TimeSpan.

5.4.24.24. totalMicroseconds

Returns the time span expressed as a number of microseconds. The result retains the sign of the receiver TimeSpan.

5.4.24.25. hashCode

Returns a string value that is used as a hash value for MapCollection such as Table, Relation, Set, Bag, and Directory.

5.4.24.26. addWeeks

Adds weeks to the TimeSpan object, returning a new TimeSpan instance. The receiver TimeSpan object is unchanged. The weeks value must be a valid number, including fractional values. Negative values result in week being subtracted from the TimeSpan value.

5.4.24.27. addDays

Adds days to the TimeSpan object, returning a new TimeSpan instance. The receiver TimeSpan object is unchanged. The days value must be a valid number, including fractional values. Negative values result in days being subtracted from the TimeSpan value.
5.4.24.28. addHours

```
addHours(hours)
```

Adds hours to the TimeSpan object, returning a new TimeSpan instance. The receiver TimeSpan object is unchanged. The hours value must be a valid number, including fractional values. Negative values result in hours being subtracted from the TimeSpan value.

5.4.24.29. addMinutes

```
addMinutes(minutes)
```

Adds minutes to the TimeSpan object, returning a new TimeSpan instance. The receiver TimeSpan object is unchanged. The minutes value must be a valid number, including fractional values. Negative values result in minutes being subtracted from the TimeSpan value.

5.4.24.30. addSeconds

```
addSeconds(seconds)
```

Adds seconds to the TimeSpan object, returning a new TimeSpan instance. The receiver TimeSpan object is unchanged. The seconds value must be a valid number, including fractional values. Negative values result in seconds being subtracted from the TimeSpan value.

5.4.24.31. addMicroseconds

```
addMicroseconds(microseconds)
```

Adds microseconds to the TimeSpan object, returning a new TimeSpan instance. The receiver TimeSpan object is unchanged. The microseconds value must be a valid whole number. Negative values result in microseconds being subtracted from the TimeSpan value.

5.4.24.32. sign

```
sign
```

Returns -1 if the TimeSpan duration is negative, 1 if it is positive, and 0 if it is zero.

See also method duration.

5.4.24.33. string

```
string
```

Returns TimeSpan formatted as a string. The string value is in the format -

```
ddddddddd.hh:mm:ss.uuuuuu
```

If the TimeSpan is positive or zero, the sign is omitted. The days field will be formatted without leading zeros or blanks. If the TimeSpan duration is less than a day, the days field and the period separator will be omitted.
5.4.25. Validate Class

The Validate class provides class methods helping with validating arguments being of correct class, logical or numeric type, or within a numeric range.

Table 5.60. Validate Class

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Object class</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class (Metaclass)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Class class</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Validate</th>
</tr>
</thead>
<tbody>
<tr>
<td>classType (Class Method)</td>
</tr>
<tr>
<td>length (Class Method)</td>
</tr>
<tr>
<td>logical (Class Method)</td>
</tr>
<tr>
<td>nonNegativeNumber (Class Method)</td>
</tr>
<tr>
<td>nonNegativeWholeNumber (Class Method)</td>
</tr>
<tr>
<td>number (Class Method)</td>
</tr>
<tr>
<td>numberRange (Class Method)</td>
</tr>
<tr>
<td>position (Class Method)</td>
</tr>
<tr>
<td>positiveNumber (Class Method)</td>
</tr>
<tr>
<td>positiveWholeNumber (Class Method)</td>
</tr>
<tr>
<td>requestClassType (Class Method)</td>
</tr>
<tr>
<td>wholeNumber (Class Method)</td>
</tr>
<tr>
<td>wholeNumberRange (Class Method)</td>
</tr>
</tbody>
</table>

5.4.25.1. classType (Class Method)

Validate that object is an instance of class class.

Raises a syntax error for argument name if the validation fails, else returns with no result.

See also method requestClassType (Class Method).

5.4.25.2. length (Class Method)

Validate that number is zero or a positive whole number under NUMERIC DIGITS digits. If digits is not specified, it defaults to internalDigits.

Raises a syntax error for argument name if the validation fails, else returns with no result.

See also methods position and nonNegativeWholeNumber.

5.4.25.3. logical (Class Method)

Validate that number is either .true or .false.

Raises a syntax error for argument name if the validation fails, else returns with no result.
5.4.25.4. nonNegativeNumber (Class Method)

Validates that *number* is zero or a positive Rexx number under NUMERIC DIGITS *digits*. If *digits* is not specified, it defaults to 9.

Raises a syntax error for argument *name* if the validation fails, else returns with no result.

See also methods *positiveNumber* and *nonNegativeWholeNumber*.

5.4.25.5. nonNegativeWholeNumber (Class Method)

Validates that *number* is zero or a positive whole number under NUMERIC DIGITS *digits*. If *digits* is not specified, it defaults to 9.

Raises a syntax error for argument *name* if the validation fails, else returns with no result.

See also methods *positiveWholeNumber* and *nonNegativeNumber*.

5.4.25.6. number (Class Method)

Validates that *number* is a valid Rexx number.

Raises a syntax error for argument *name* if the validation fails, else returns with no result.

See also method *wholeNumber*.

5.4.25.7. numberRange (Class Method)

Validates that *number* is a valid Rexx number in the range *min* to *max*, with comparisons done under NUMERIC DIGITS *digits*. If *digits* is not specified, it defaults to 9.

Raises a syntax error for argument *name* if the validation fails, else returns with no result.

See also method *wholeNumberRange*.

5.4.25.8. position (Class Method)
Validates that `number` is a positive whole number under NUMERIC DIGITS `digits`. If `digits` is not specified, it defaults to `internalDigits`.

Raises a syntax error for argument `name` if the validation fails, else returns with no result.

See also methods `length` and `positiveWholeNumber`.

### 5.4.25.9. positiveNumber (Class Method)

Validates that `number` is a positive Rexx number under NUMERIC DIGITS `digits`. If `digits` is not specified, it defaults to 9.

Raises a syntax error for argument `name` if the validation fails, else returns with no result.

See also methods `nonNegativeNumber` and `positiveWholeNumber`.

### 5.4.25.10. requestClassType (Class Method)

Validates that `object` can be converted to an instance of class `class` by sending it a request message.

If successful, it returns the converted object, otherwise it raises a syntax error for argument `name`.

See also method `classType (Class Method)`.

### 5.4.25.11. positiveWholeNumber (Class Method)

Validates that `number` is a positive whole number under NUMERIC DIGITS `digits`. If `digits` is not specified, it defaults to 9.

Raises a syntax error for argument `name` if the validation fails, else returns with no result.

See also methods `nonNegativeWholeNumber` and `positiveNumber`.

### 5.4.25.12. wholeNumber (Class Method)
VariableReference Class

5.4.25.13. wholeNumberRange (Class Method)

Validates that number is a valid whole number in the range min to max, with comparisons done under NUMERIC DIGITS digits. If digits is not specified, it defaults to 9.

Raises a syntax error for argument name if the validation fails, else returns with no result.

See also method numberRange.

5.4.26. VariableReference Class

A VariableReference instance maintains a reference to another object. It can only be created using a Variable Reference Term. Calling the new method to create a VariableReference instance is not allowed.

Table 5.61. VariableReference Class

<table>
<thead>
<tr>
<th></th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Object class</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>VariableReference</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>unknown</td>
</tr>
<tr>
<td>request</td>
<td>value (Attribute)</td>
</tr>
</tbody>
</table>

5.4.26.1. name

Returns the name of the variable referenced.

See also method value (Attribute).

Example 5.317. VariableReference class — name method

```
variable = 123
```
5.4.26.2. request

Forwards to the `request` method of the referenced variable.

5.4.26.3. unknown

Forwards to the `unknown` method of the referenced variable.

5.4.26.4. value (Attribute)

- **value get:**
  Returns the current value of the referenced variable.

- **value set:**
  Sets the value of the referenced variable to `object`.

See also method `name`.

Example 5.318. VariableReference class — value method

```
array = 1, 2, 3
say >array-class-id -- VariableReference
say >array-value-class-id -- Array
say >array-value-toString(, ",") -- 1, 2, 3
-- this also works because we have an UNKnOWN method
say >array-toString(, ",") -- 1, 2, 3
```

5.4.27. WeakReference Class

A WeakReference instance maintains a non-pinning reference to another object. A non-pinning reference does not prevent an object from getting garbage collected or having its `uninit` method run when there are no longer normal references maintained to the object. Once the referenced object is eligible for garbage collection, the reference inside the WeakReference instance will be cleared and the `value` method will return `.nil` on all subsequent calls. WeakReferences are useful for maintaining caches of objects without preventing the objects from being reclaimed by the garbage collector when needed.
Table 5.62. WeakReference Class

<table>
<thead>
<tr>
<th></th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from the Object class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Class (Metaclass)</td>
</tr>
<tr>
<td>Methods inherited from the Class class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WeakReference</td>
</tr>
<tr>
<td>new (Class Method)</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
</tbody>
</table>

5.4.27.1. new (Class Method)

![new](object)

Returns a new WeakReference instance containing a reference to `object`.

5.4.27.2. value

![value](object)

Returns the referenced object. If the object has been garbage collected, `.nil` is returned.
Rexx Runtime Objects

In addition to the class objects described in the previous chapter, the Rexx runtime environment also provides objects that are accessible via *environment symbols*.

### 6.1. The Environment Directory (.ENVIRONMENT)

The Environment object is a directory of public objects that are always accessible. The Environment object is automatically searched when environment symbols are used, or the Environment object may be directly accessed using the .ENVIRONMENT symbol. Entries stored in the Environment use the same name as the corresponding environment symbol, but without a leading period. For example:

```plaintext
say .true                   -- Displays "1"
say .environment-true      -- Also displays "1"
say .environment["TRUE"]  -- "1"
say .environment["true"]   -- The NIL object
```

The Environment object directory contains all of the Rexx built-in classes (Array, etc.), the special Rexx constants `.NIL`, `.TRUE`, `.FALSE`, `.ENDOFLINE`, and the `.RexxInfo object`.

#### 6.1.1. The ENDOFLINE Constant (.ENDOFLINE)

The ENDOFLINE object is a string constant representing the *line-end* characters. It is `'0d 0a'x` (carriage-return, line-feed) on Windows, and `'0a'x` (line-feed) on Unix-like systems.

#### 6.1.2. The FALSE Constant (.FALSE)

The FALSE object is the constant "0" representing a FALSE result for logical and comparison operations.

#### 6.1.3. The NIL Object (.NIL)

The Nil object is a special object that does not contain data. It usually represents the absence of an object, as a null string represents a string with no characters. It has only the methods of the Object class. Note that you use the Nil object (rather than the null string ("

```plaintext
if .nil = board[row,col]   /* .NIL rather than "" */
then ...
```

#### 6.1.4. The RexxInfo Object (.RexxInfo)

The RexxInfo object returns the only instance of the *RexxInfo class* and gives access to Rexx language information and other platform-specific information in a single place.

```plaintext
say .RexxInfo~name        -- REXX-ooRexx_5.0.0(MT)_64-bit 6.05 1 Sep 2016 /* e. g. */
```
6.1.5. The TRUE Constant (.TRUE)

The TRUE object is the constant "1", representing a true result for logical and comparison operations.

6.2. The Local Directory (.LOCAL)

The Local environment object is a directory of interpreter instance objects that are always accessible. You can access objects in the Local environment object in the same way as objects in the Environment object. The Local object contains

- the `.INPUT`, `.OUTPUT`, `.ERROR`, `.DEBUGINPUT`, and `.TRACEOUTPUT` Monitor objects used for console I/O,
- the `.STDIN`, `.STDOUT`, and `.STDERR` output streams that are the default I/O targets,
- the `.STDQUE` RexxQueue instance used for Rexx external queue operations,
- and the `.SYSCARGS` array of program command line options.

Because both .ENVIRONMENT and .LOCAL are Directory objects, you can place objects into, or retrieve objects from, these environments by using any of the Directory methods [], [], =, put, at, setEntry, entry, or setMethod). To avoid potential name clashes with built-in objects and public objects that Rexx provides, each object that your programs add to these environments should have a period in its index.

Example 6.1. .LOCAL

```rexx
/* .LOCAL example--places something in the Local environment directory */
.local~my.alarm = theAlarm

/ * To retrieve it */
say .local~my.alarm

/* Another .LOCAL example (Windows) */
.environment["MYAPP.PASSWORD"] = "topsecret"
.environment["MYAPP.UID"] = 200

/ * Create a local directory for my stuff */
.local["MYAPP.LOCAL"] = .directory~new

/ * Add log file for my local directory */
.myapp.local["LOG"] = .stream~new("myapp.log")
say .myapp.password                      /* Displays "topsecret" */
say .myapp.uid                           /* Displays "200" */

/ * Write a line to the log file */
.myapp.log~lineout("Logon at "time()" on "date())

/ * Redirect SAY lines into a file: */
.output~destination(.stream~new("SAY_REDIRECT.TXT"))
say "This goes into a file, and not onto the screen!"

/ * .LOCAL example--get the individual command line arguments */
cmdargs = .local~syscargs
do carg over cmdargs
    say carg
end
```

438
6.3. The Debug Input Monitor (.DEBUGINPUT)

This Monitor object holds the default interactive debug input stream object (see Chapter 14, Input and Output Streams). This input stream is the source for all input for interactive debug mode.

The default for this object's initial source is the .INPUT monitor.

6.4. The Error Monitor (.ERROR)

This Monitor object holds the error stream object. You can redirect the trace output in the same way as with the output object in the Monitor class example.

The default for this object's initial destination is the .STDERR stream.

6.5. The Input Monitor (.INPUT)

This Monitor object holds the default input stream object (see Chapter 14, Input and Output Streams). This input stream is the source for the PARSE LINEIN instruction, the LINEIN method of the Stream class, and, if you specify no stream name, the LINEIN built-in function. It is also the source for the PULL and PARSE PULL instructions if the external data queue is empty.

The default for this object's initial source is the .STDIN stream.

6.6. The Output Monitor (.OUTPUT)

This Monitor object holds the default output stream object (see Chapter 14, Input and Output Streams). This is the destination for output from the SAY instruction, the LINEOUT method (.OUTPUT~LINEOUT), and, if you specify no stream name, the LINEOUT built-in function. You can replace this object in the environment to direct such output elsewhere (for example, to a transcript window).

The default for this object's initial destination is the .STDOUT stream.

6.7. The Trace Output Monitor (.TRACEOUTPUT)

This Monitor object holds the trace output target object. You can redirect the trace output in the same way as with the output object in the Monitor class example.

The default for this object's initial destination is the .ERROR monitor.

6.8. The STDERR Stream (.STDERR)

This Stream object is the default stream used for trace and error message output.

6.9. The STDIN Stream (.STDIN)

This Stream object is representing the standard input file of a process. It is the startup default stream for the .INPUT object.
6.10. The STDOUT Stream (.STDOUT)

This Stream object is representing the standard output file of a process. It is the startup default stream for the .OUTPUT object.

6.11. The STDQUE Queue (.STDQUE)

This RexxQueue object is the destination for the PUSH and QUEUE instruction, and the source for queue lines for the PULL and PARSE PULL instructions.

6.12. The SYSCARGS Array (.SYSCARGS)

The .SysCArgs object is an Array of all command-line arguments supplied to the program. Normally all command-line arguments are combined into a single string and passed to the Rexx program as an argument retrievable via the ARG (Argument) built-in function. .SYSCARGS is a direct collection of the individual C arguments passed to the program.

Example:

Example 6.2. SysCArgs Array

```rexx
say .SysCArgs~items "SysCArgs" .SysCArgs~makeString(, ",")
say arg() "ARGs" arg(1, "A")~makeString(, ",")

/* when called with command line parameters 1 "2 3" 4, displays:
  3 SysCArgs 1,2 3,4
  1 ARGs 1 "2 3" 4
  */
```

Note

.SysCArgs may not be available in all situations, e.g. in program code that gets run via "rexx -e".

6.13. The Rexx Context (.CONTEXT)

The .CONTEXT environment symbol accesses a RexxContext instance for the currently active Rexx execution environment. The returned context object is only active until the current method call, routine call, or program terminates. Once the context object is deactivated, an error will be raised if any of the RexxContext methods are called.

6.14. The Line Number (.LINE)

.LINE is set to the line number of the instruction currently being executed. If the current instruction is defined within an INTERPRET instruction, the line number of INTERPRET instruction is returned.
6.15. The METHODS StringTable (.METHODS)

The .METHODS environment symbol identifies a StringTable of methods that ::ATTRIBUTE, ::CONSTANT, or ::METHOD directives in the currently running program define. The StringTable indexes are the method names. The StringTable values are the method objects.

Only methods or attributes that are not preceded by a ::CLASS directive are in the .METHODS StringTable. These are known as floating methods. If there are no such methods, the .METHODS symbol has the default value of .METHODS.

Example 6.3. .METHODS

```/* .methods contains one entry with the index (method name) "TALK" */
o=.object~enhanced(.methods) /* create object, enhance it with methods */
o~talk("echo this text") /* test "TALK" method */
::method talk /* floating method by the name of "TALK" */
  use arg text /* retrieve the argument */
  say text /* display received argument */
```

6.16. The ROUTINES StringTable (.ROUTINES)

The .ROUTINES environment symbol identifies a StringTable of routines that ::ROUTINE directives in the currently running program define. The StringTable indexes are the routine names. The StringTable values are the routine objects.

If there are no ::ROUTINE directives, the .ROUTINES symbol has the default value of .ROUTINES.

Example 6.4. .ROUTINES

```/* .routines contains one entry with the index (routine name) "TALK" */
.routines~talk~call("echo this text") /* test talk routine */
::routine talk /* floating routine by the name of "TALK" */
  use arg text /* retrieve the argument */
  say text /* display received argument */
```

6.17. The RESOURCES StringTable (.RESOURCES)

The .RESOURCES environment symbol identifies a StringTable of data resources that ::RESOURCE directives in the currently running program define. The StringTable indexes are the resource names, the StringTable values are arrays of individual resource data lines.

If there are no ::RESOURCE directives, the .RESOURCES symbol has the default value of .RESOURCES.

Example 6.5. .RESOURCE

```do name over .resources
  say name":" .resources[name]~items "lines"
  say .resources[name]~makeString
end
::resource greyCat end "-"
La nuit, tous les chats sont gris```
::resource "brown fox"
The quick brown fox jumps over the lazy dog
::END

::resource nollop end ANONYMOUS
  The wicked peon quivered,
  then gazed balefully at the judges
  who examined him.
ANONYMOUS TYPESETTER
/* Displays:
GREYCAT: 1 lines
La nuit, tous les chats sont gris
BROWN FOX: 1 lines
The quick brown fox jumps over the lazy dog
NOLLOP: 3 lines
  The wicked peon quivered,
  then gazed balefully at the judges
  who examined him.
*/

6.18. The Return Status (.RS)

.RS is set to the return status from any executed command (including those submitted with the
ADDRESS instruction). The .RS environment symbol has a value of -1 when a command returns a
FAILURE condition, a value of 1 when a command returns an ERROR condition, and a value of 0
when a command indicates successful completion. The value of .RS is also available after trapping the
ERROR or FAILURE condition.

Note

Commands executed manually during interactive tracing do not change the value of .RS. The
initial value of .RS is .RS.
Chapter 7.

Functions

A function is an internal, built-in, or external routine that returns a single result object. (A subroutine is a function that is an internal, built-in, or external routine that might return a result and is called with the CALL instruction.)

7.1. Syntax

A function call is a term in an expression calling a routine that carries out some procedures and returns an object. This object replaces the function call in the continuing evaluation of the expression. You can include function calls to internal and external routines in an expression anywhere that a data term (such as a string) would be valid, using the following notation:

```
function_name(expression,)
```

The `function_name` is a literal string or a single symbol, which is taken to be a constant.

There can be any number of expressions, separated by commas, between the parentheses. These expressions are called the arguments to the function. Each argument expression can include further function calls.

Note that the left parenthesis must be adjacent to the name of the function, with no whitespace characters in between. (A blank operator would be assumed at this point instead.) Only a comment can appear between the name and the left parenthesis.

The arguments are evaluated in turn from left to right and the resulting objects are then all passed to the function. This function then runs some operation (usually dependent on the argument objects passed, though arguments are not mandatory) and eventually returns a single object. This object is then included in the original expression as though the entire function reference had been replaced by the name of a variable whose value is the returned object.

For example, the function `SUBSTR` is built into the language processor and could be used as:

```
N1="abcdefghijk"
Z1="Part of N1 is: "substr(N1,2,7)
/* Sets Z1 to "Part of N1 is: bcdefgh" */
```

A function can have a variable number of arguments. You need to specify only those required. For example, `SUBSTR("ABCDEF", 4)` would return `DEF`.

7.2. Functions and Subroutines

Functions and subroutines are called in the same way. The only difference between functions and subroutines is that functions must return data, whereas subroutines need not.

The following types of routines can be called as functions:

Internal

If the routine name exists as a label in the program, the current processing status is saved for a later return to the point of invocation to resume execution. Control is then passed to the first label in the program that matches the name. As with a routine called by the CALL instruction, status
information, such as TRACE and NUMERIC settings, is saved too. See the CALL instruction for
details.

If you call an internal routine as a function, you must specify an expression in any RETURN
instruction so that the routine can return. This is not necessary if it is called as a subroutine.

Example 7.1. Recursive internal function execution

```plaintext
arg x
say x"! = " factorial(x)
exit

factorial: procedure /* Calculate factorial by */
arg n /* recursive invocation. */
if n=0 then return 1
return factorial(n-1) * n
```

FACTORIAL is unusual in that it calls itself (this is recursive invocation). The PROCEDURE
instruction ensures that a new variable n is created for each invocation.

Built-in
These functions are always available and are defined in Section 7.4, “Built-in Functions”.

External
You can write or use functions that are external to your program and to the language processor.
An external routine can be written in any language, including Rexx, that supports the system-
dependent interfaces the language processor uses to call it. You can call a Rexx program as a
function and, in this case, pass more than one argument string. The ARG, PARSE ARG, or USE
ARG instruction or the ARG built-in function can retrieve these argument strings. When called as a
function, a program must return data to the caller.

Notes:
1. Calling an external Rexx program as a function is similar to calling an internal routine. For an
   external routine, however, the caller's variables are hidden. To leave the called Rexx program,
   you can use either EXIT or RETURN. In either case, you must specify an expression.
2. You can use the INTERPRET instruction to process a function with a variable function name.
   However, avoid this if possible because it reduces the clarity of the program.

7.2.1. Search Order

Functions are searched in the following sequence: internal routines, built-in functions, external
functions.

Function calls or subroutines may use a name that is specified as a symbol or a literal string. For
example, these calls are equivalent:

```plaintext
call MyProcedure
call ‘MYPROCEDURE’
```

Note that the name value when specified as a symbol is the symbol name translated to upper case.
Both of the calls above will search for a routine named "MYPROCEDURE". When the name is
specified as a literal string, then the literal string value is used as-is. Thus the following two calls are
not equivalent:
Some steps of the function and subroutine search order are case sensitive, so some care may need to be exercised that the correct name form is used:

- **Internal routines.** Normally, labels are specified as a symbol followed by a ":". These labels have a name value that's all uppercase. Since unquoted (symbol) names also have uppercase values, these will match easily. It is also possible to use literal strings for label names. If these labels contain lowercase characters, they will not be located using normal call mechanisms.

- **Built-in functions.** The built-in function names are all uppercase, so using a mixed-case literal string built-in function name will fail to locate the function.

- **External routines.** Some steps of the external function search order may be case sensitive, depending on the system. This may occasionally require a function or subroutine name to be specified as a mixed case literal string to be located.

If the call or function invocation uses a literal string, then the search for internal label is bypassed. This bypass mechanism allows you to extend the capabilities of an existing internal function, for example, and call it as a built-in function or external routine under the same name as the existing internal function. To call the target built-in or external routine from inside your internal routine, you must use a literal string for the function name.

**Example 7.2. DATE function — overriding**

```plaintext
/* This internal DATE function modifies the default for the DATE function to standard date. */
date: procedure
arg in
if in="" then in="Standard"
-- This calls the DATE built-in function rather than recursively
-- calling the DATE: internal routine. Note that the name needs to
-- be all uppercase because built-in functions have uppercase names.
return "DATE"(in)
```

Since built-in functions have uppercase names the literal string must also be in uppercase for the search to succeed.

External functions and subroutines have a system-defined search order.

The search order for external functions is as follows:

1. Functions defined on ::ROUTINE directives within the program.
2. Public functions defined on ::ROUTINE directives of programs referenced with ::REQUIRES.
3. Functions that have been loaded into the macrospace for preorder execution. (See the Open Object Rexx: Application Programming Interfaces for details.)
4. Functions that are part of a function package or library package. (See the Open Object Rexx: Application Programming Interfaces for details.)
5. Rexx functions located in an external file. See below for how these external files are located.
6. Functions that have been loaded into the macrospace for postorder execution.

7.2.1.1. Locating External Rexx Files

Rexx uses an extensive search procedure for locating program files. The first element of the search procedure is the locations that will be checked for files. The locations, in order of checking, are:

1. The same directory the program invoking the external routine is located. If this is an initial program execution or the calling program was loaded from the macrospace, this location is skipped. Checking in this directory allows related program files to be called without requiring the directory be added to the search path.

2. The current filesystem directory.

3. Some applications using Rexx as a scripting language may define an extension path used to locate called programs. If the Rexx program was invoked directly from the system command line, then no extension path is defined.

4. Any directories specified via the REXX_PATH environment variable.

5. Any directories specified via the PATH environment variable.

The second element of the search process is the file extension. If the routine name contains at least one period, then this routine is extension qualified. The search locations above will be checked for the target file unchanged, and no additional steps will be taken. If the routine name is not extension qualified, then additional searches will be performed by adding file extensions to the name. All directory locations will be checked for a given extension before moving to the next potential extension. The following extensions may be used:

1. If the searched file is requested by a ::REQUIRES directive without a LIBRARY option, or the Package methods new and loadPackage when only the name argument is specified, an attempt to locate a file using the extension .cls is made.

2. If the calling program has a file extension, then the interpreter will attempt to locate a file using the same extension as the caller.

3. Some applications using Rexx as a scripting language may define additional extension types. For example, an editor might define a preferred extension that should be used for editor macros. This extension would be searched next.

4. The default system extension, which is .REX on Windows, and both .rex and .REX on Unix-like systems.

5. If the target file has not been located using any of the above extensions, the file name is tried without an added extension.

There are some file system considerations involved when searching for files. Windows file systems typically are case insensitive, so files can be located regardless of how the call is specified. Unix-like systems typically have a case sensitive file system, so files must be exact case matches in order to be located. For these systems, each time a file name probe is attempted, the name will be tried in the case specified and also as a lower case name. The check is not performed on the very last step that uses the file name without an extension to avoid unintentional conflicts with other executable files.

Note that for function or subroutine calls using an unquoted name, the target name is the string value of the name symbol, which will be an uppercase value. Thus calls to myfunc(), MyFunc(), and myFUNC() all trigger a search for a function named “MYFUNC”. Calls specified as a quoted string will maintain the original string case. Thus ‘myfunc()’ and ‘MyFunc’() would search for different names.
Figure 7.1. Function and Routine Resolution and Execution
Errors during Execution

7.2.2. Errors during Execution

If an external or built-in function detects an error, the language processor is informed, and a syntax error results. Syntax errors can be trapped in the caller using SIGNAL ON SYNTAX and recovery might be possible. If the error is not trapped, the program is ended.

7.3. Return Values

A function usually returns a value that is substituted for the function call when the expression is evaluated.

How the value returned by a function (or any Rexx routine) is handled depends on whether it is called by a function call or as a subroutine with the CALL instruction.

- A routine called as a subroutine: If the routine returns a value, that value is stored in the special variable named RESULT. Otherwise, the RESULT variable is dropped, and its value is the string RESULT.

- A routine called as a function: If the function returns a value, that value is substituted in the expression at the position where the function was called. Otherwise, the language processor stops with an error message.

Here are some examples of how to call a Rexx procedure:
Example 7.3. How to call Rexx procedures

```rexx
call Beep 500, 100          /* Example 1: a subroutine call */
```

The built-in function `BEEP` is called as a Rexx subroutine. The return value from `BEEP` is placed in the Rexx special variable `RESULT`.

```rexx
bc = Beep(500, 100)         /* Example 2: a function call */
```

`BEEP` is called as a Rexx function. The return value from the function is substituted for the function call. The clause itself is an assignment instruction; the return value from the `BEEP` function is placed in the variable `bc`.

```rexx
Beep(500, 100)              /* Example 3: result passed as */
/*            a command         */
```

The `BEEP` function is processed and its return value is substituted in the expression for the function call, like in the preceding example. In this case, however, the clause as a whole evaluates to a single expression. Therefore, the evaluated expression is passed to the current default environment as a command.

Note

Many other languages, such as C, throw away the return value of a function if it is not assigned to a variable. In Rexx, however, a value returned like in the third example is passed on to the current environment or subcommand handler. If that environment is the default, the operating system performs a disk search for what seems to be a command.

7.4. Built-in Functions

Rexx provides a set of built-in functions, including character manipulation, conversion, and information functions. The following are general notes on the built-in functions:

- The parentheses in a function are always needed, even if no arguments are required. The first parenthesis must follow the name of the function with no whitespace in between.

- The built-in functions internally work with `NUMERIC DIGITS 9` for 32-bit systems or `NUMERIC DIGITS 18` for 64-bit systems, and `NUMERIC FUZZ 0` and are unaffected by changes to the `NUMERIC` settings, except where stated. Any argument named as a `number` is rounded, if necessary, according to the current setting of `NUMERIC DIGITS` (as though the number had been added to 0) and checked for validity before use. This occurs in the following functions: `ABS`, `FORMAT`, `MAX`, `MIN`, `SIGN`, and `TRUNC`, and for certain options of `DATATYPE`.

- Any argument named as a `string` can be a null string.

- If an argument specifies a `length`, it must be a positive whole number or zero. If it specifies a `start` character or word in a string, it must be a positive whole number, unless otherwise stated.
• If the last argument is optional, you can always include a comma to indicate that you have omitted it. For example, DATATYPE(1,), like DATATYPE(1), would return NUM. You can include any number of trailing commas; they are ignored. If there are actual parameters, the default values apply.

• If you specify a pad character, it must be exactly one character long. A pad character extends a string, usually on the right. For an example, see the LEFT built-in function.

• If a function has an option that you can select by specifying the first character of a string, that character can be in uppercase or lowercase.

• Many of the built-in functions invoke methods of the String class. For the functions ABBREV, ABS, BITAND, BITOR, BITXOR, B2X, CENTER, CENTRE, CHANGESTR, COMPARE, COPIES, COUNTSTR, C2D, C2X, DATATYPE, DELSTR, DELWORD, D2C, D2X, FORMAT, LEFT, LENGTH, LOWER, MAX, MIN, REVERSE, RIGHT, SIGN, SPACE, STRIP, SUBSTR, SUBWORD, TRANSLATE, TRUNC, UPPER, VERIFY, WORD, WORDINDEX, WORDLENGTH, WORDS, X2B, X2C, and X2D, the first argument to the built-in function is used as the receiver object for the message sent, and the remaining arguments are used in the same order as the message arguments. For example, SUBSTR("abcde", 3, 2) is equivalent to "abcde"~substr(3, 2).

For the functions INSERT, LASTPOS, OVERLAY, POS, and WORDPOS, the second argument to the built-in functions is used as the receiver object for the message sent, and the other arguments are used in the same order as the message arguments. For example, POS("a", "Haystack",3) is equivalent to "Haystack"~pos("a", 3).

• The language processor evaluates all built-in function arguments to produce character strings.

### 7.4.1. ABBREV (Abbreviation)

Returns 1 if info is equal to the leading characters of information and the length of info is not less than length. It returns 0 if either of these conditions is not met.

If you specify length, it must be a positive whole number or zero. The default for length is the number of characters in info.

Here are some examples:

#### Example 7.4. Built-in function ABBREV examples

| ABBREV("Print","Pri") | -> 1     |
| ABBREV("PRINT","Pri") | -> 0     |
| ABBREV("PRINT","PRI",4) | -> 0     |
| ABBREV("PRINT","PRY") | -> 0     |
| ABBREV("PRINT","")    | -> 1     |
| ABBREV("PRINT","",1)  | -> 0     |
Note

A null string always matches if a length of 0, or the default, is used. This allows a default keyword to be selected automatically if desired; for example:

Example 7.5. Builtin function ABBREV example

```
say "Enter option:";   pull option .
select /* keyword1 is to be the default */
   when abbrev("keyword1",option) then ... 
   when abbrev("keyword2",option) then ... 
   ... 
   otherwise nop; 
end;
```

7.4.2. ABS (Absolute Value)

```
ABS( number )
```

Returns the absolute value of number. The result has no sign and is formatted according to the current NUMERIC settings.

Here are some examples:

Example 7.6. Builtin function ABS

```
ABS("12.3")       ->    12.3
ABS(" -0.307")    ->    0.307
```

7.4.3. ADDRESS

```
ADDRESS()
```

Returns the name of the environment to which commands are currently submitted. Trailing whitespace characters are removed from the result.

Here is an example:

Example 7.7. Builtin function ADDRESS

```
ADDRESS()    ->    "CMD"   // default for Windows
ADDRESS()    ->    "sh"    // default for Unix-like systems
```

7.4.4. ARG (Argument)
ARG (Argument)

Returns one or more arguments, or information about the arguments to a program, internal routine, or method.

If you do not specify any argument, the number of arguments passed to the program or internal routine is returned.

If you specify only \( n \), the \( n \)th argument object is returned. If the argument object does not exist, the null string is returned. \( n \) must be a positive whole number.

If you specify \( \text{option} \), the value returned depends on the value of \( \text{option} \). The following are valid \( \text{options} \). (Only the capitalized letter is needed; all characters following it are ignored.)

**Array**

returns a single-index array containing the arguments, starting with the \( n \)th argument. The array indexes correspond to the argument positions, so that the \( n \)th argument is at index 1, the following argument at index 2, and so on. If any arguments are omitted, their corresponding indexes are absent.

**Exists**

returns 1 if the \( n \)th argument exists; that is, if it was explicitly specified when the routine was called. Otherwise, it returns 0.

**Normal**

returns the \( n \)th argument, if it exists, or a null string.

**Omitted**

returns 1 if the \( n \)th argument was omitted; that is, if it was not explicitly specified when the routine was called. Otherwise, it returns 0.

Here are some examples:

**Example 7.8. Builtin function ARG**

```plaintext
/* following "Call name;" (no arguments) */
ARG()    ->  0
ARG(1)   ->  ""
ARG(2)   ->  ""
ARG(1,"e") ->  0
ARG(1,"O") ->  1
ARG(1,"a") -> .array-of()

/* following "Call name 'a', 'b';" */
ARG()    ->  3
ARG(1)   ->  "a"
ARG(2)   ->  ""
ARG(3)   ->  "b"
ARG(n)   ->  ""    /* for n<=4 */
ARG(1,"e") ->  1
ARG(2,"E") ->  0
ARG(2,"O") ->  1
ARG(3,"o") ->  0
ARG(4,"o") ->  1
ARG(1,"A") -> .array-of(a, ,b)
ARG(3,"a") -> .array-of(b)
```
B2X (Binary to Hexadecimal)

Notes:

1. The number of argument strings is the largest number \( n \) for which \( \text{ARG}(n, "e") \) returns 1 or 0 if there are no explicit argument strings. That is, it is the position of the last explicitly specified argument string.

2. Programs called as commands can have only 0 or 1 argument strings. The program has 0 argument strings if it is called with the name only and has 1 argument string if anything else (including whitespace characters) is included in the command.

3. Programs called by the RexxStart entry point can have several argument strings. (See the Open Object Rexx: Application Programming Interfaces for information about RexxStart.)

4. You can access the argument objects of a program with the \textit{USE} instruction.

5. You can retrieve and directly parse the argument strings of a program or internal routine with the \textit{ARG} or \textit{PARSE ARG} instructions.

7.4.5. B2X (Binary to Hexadecimal)

\[
\text{B2X(binary\_string)}
\]

Returns a string, in character format, that represents \( \text{binary\_string} \) converted to hexadecimal.

The \( \text{binary\_string} \) is a string of binary (0 or 1) digits. It can be of any length. You can optionally include whitespace characters in \( \text{binary\_string} \) (at 4-digit boundaries only, not leading or trailing) to improve readability; they are ignored.

The returned string uses uppercase alphabetical characters for the values A-F, and does not include blanks or horizontal tabs.

If \( \text{binary\_string} \) is the null string, B2X returns a null string. If the number of binary digits in \( \text{binary\_string} \) is not a multiple of 4, then up to three 0 digits are added on the left before the conversion to make a total that is a multiple of 4.

Here are some examples:

**Example 7.9. Builtin function B2X**

\[
\begin{array}{ll}
\text{B2X("11000011")} & \rightarrow \text{"C3"} \\
\text{B2X("10111")} & \rightarrow \text{"17"} \\
\text{B2X("101")} & \rightarrow \text{"5"} \\
\text{B2X("1 1111 0000")} & \rightarrow \text{"1F0"}
\end{array}
\]

You can combine B2X with the functions X2D and X2C to convert a binary number into other forms. For example:

**Example 7.10. Builtin function B2X combined with X2D function**

\[
\text{X2D(B2X("10111"))} \rightarrow \text{"23" /* decimal 23 */}
\]

7.4.6. BEEP
BITAND (Bit by Bit AND)

Sounds the speaker at frequency Hertz for duration milliseconds. The frequency can be any whole number in the range 37 to 32767 Hertz. The duration can be any whole number in the range 0 to 60000 milliseconds.

This routine is most useful when called as a subroutine. A null string is returned.

Notes

On Unix-like systems, to sound the speaker at a specified frequency, a console with ioctl KDMKTONE support must be available, which typically requires root access. If there is no console with KDMKTONE support (e.g., on BSD or Darwin), a bell character is sent to the console instead.

On Windows beeps may have been deactivated. To check the status of the Windows beep service, use the Service Control Manager command `sc qc beep`. To permanently enable the system beep service, run the command `sc config beep start= auto` in a Command window with Administrator rights, and then reboot your system.

Here is an example for Windows:

Example 7.11. Builtin function BEEP

```plaintext
-- C scale
note.1 = 262   -- Do, middle C
note.2 = 294   -- Re, D
note.3 = 330   -- Mi, E
note.4 = 349   -- Fa, F
note.5 = 392   -- Sol, G
note.6 = 440   -- La, A
note.7 = 494   -- Si, B, H
note.8 = 523   -- Do, C

doi = 1 to 8
    call beep note.i, 250 -- hold each note for 1/4 second
end
```

7.4.7. BITAND (Bit by Bit AND)

Returns a string composed of the two input strings logically ANDed, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no pad character is provided, the AND operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If pad is provided, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for string2 is the zero-length (null) string.

Here are some examples:
**7.4.8. BITOR (Bit by Bit OR)**

Returns a string composed of the two input strings logically inclusive-ORed, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no pad character is provided, the OR operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If pad is provided, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for string2 is the zero-length (null) string.

Here are some examples:

**Example 7.13. Builtin function BITOR**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITOR(&quot;12&quot;x)</td>
<td>&quot;12&quot;x</td>
</tr>
<tr>
<td>BITOR(&quot;15&quot;x,&quot;24&quot;x)</td>
<td>&quot;35&quot;x</td>
</tr>
<tr>
<td>BITOR(&quot;15&quot;x,&quot;2456&quot;x)</td>
<td>&quot;3556&quot;x</td>
</tr>
<tr>
<td>BITOR(&quot;15&quot;x,&quot;2456&quot;x,&quot;F0&quot;x)</td>
<td>&quot;35F6&quot;x</td>
</tr>
<tr>
<td>BITOR(&quot;1111&quot;x,&quot;4D&quot;x)</td>
<td>&quot;5D5D&quot;x</td>
</tr>
<tr>
<td>BITOR(&quot;pQrS&quot;,&quot;20&quot;x)</td>
<td>&quot;pqrs&quot; /* ASCII */</td>
</tr>
</tbody>
</table>

**7.4.9. BITXOR (Bit by Bit Exclusive OR)**

Returns a string composed of the two input strings logically eXclusive-ORed, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no pad character is provided, the XOR operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If pad is provided, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for string2 is the zero-length (null) string.

Here are some examples:

**Example 7.14. Builtin function BITXOR**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITXOR(&quot;12&quot;x)</td>
<td>&quot;12&quot;x</td>
</tr>
<tr>
<td>BITXOR(&quot;12&quot;x,&quot;22&quot;x)</td>
<td>&quot;30&quot;x</td>
</tr>
<tr>
<td>BITXOR(&quot;1211&quot;x,&quot;22&quot;x)</td>
<td>&quot;3011&quot;x</td>
</tr>
<tr>
<td>BITXOR(&quot;1111&quot;x,&quot;444444&quot;x)</td>
<td>&quot;555544&quot;x</td>
</tr>
</tbody>
</table>
7.4.10. C2D (Character to Decimal)

Returns the decimal value of the binary representation of `string`. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS. If you specify `n`, it is the length of the returned result. If you do not specify `n`, `string` is processed as an unsigned binary number.

If `string` is null, 0 is returned.

Here are some examples:

Example 7.15. Builtin function C2D

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2D(&quot;09&quot;X)</td>
<td>9</td>
</tr>
<tr>
<td>C2D(&quot;81&quot;X)</td>
<td>129</td>
</tr>
<tr>
<td>C2D(&quot;FF81&quot;X)</td>
<td>65409</td>
</tr>
<tr>
<td>C2D(&quot;&quot;</td>
<td>0</td>
</tr>
<tr>
<td>C2D(&quot;a&quot;)</td>
<td>97     /* ASCII */</td>
</tr>
</tbody>
</table>

If you specify `n`, the string is taken as a signed number expressed in `n` characters. The number is positive if the leftmost bit is off, and negative if the leftmost bit is on. In both cases, it is converted to a whole number, which can be negative. The `string` is padded on the left with "00"x characters (not "sign-extended"), or truncated on the left to `n` characters. This padding or truncation is as though `RIGHT(string, n, "00"x)` had been processed. If `n` is 0, C2D always returns 0.

Here are some examples:

Example 7.16. Builtin function C2D

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2D(&quot;81&quot;X, 1)</td>
<td>-127</td>
</tr>
<tr>
<td>C2D(&quot;81&quot;X, 2)</td>
<td>129</td>
</tr>
<tr>
<td>C2D(&quot;FF81&quot;X, 2)</td>
<td>-127</td>
</tr>
<tr>
<td>C2D(&quot;FF81&quot;X, 1)</td>
<td>-127</td>
</tr>
<tr>
<td>C2D(&quot;FF7F&quot;X, 1)</td>
<td>127</td>
</tr>
<tr>
<td>C2D(&quot;F081&quot;X, 2)</td>
<td>-3967</td>
</tr>
<tr>
<td>C2D(&quot;F081&quot;X, 1)</td>
<td>-127</td>
</tr>
<tr>
<td>C2D(&quot;0031&quot;X, 0)</td>
<td>0</td>
</tr>
</tbody>
</table>

7.4.11. C2X (Character to Hexadecimal)

Returns a string, in character format, that represents `string` converted to hexadecimal. The returned string contains twice as many bytes as the input string. On an ASCII system, C2X(1) returns 31 because the ASCII representation of the character 1 is "31"X.
The string returned uses uppercase alphabetical characters for the values A-F and does not include whitespace characters. The string can be of any length. If string is null, a null string is returned.

Here are some examples:

### Example 7.17. Builtin function C2X

| C2X("0123"X)   | -> | "0123" /* 30313233X in ASCII */ |
| C2X("2D8")     | -> | "5A4438" /* 354134343338X in ASCII */ |

# 7.4.12. CENTER (or CENTRE)

RETURNS a string of length length with string centered in it and with pad characters added as necessary to make up length. The length must be a positive whole number or zero. The default pad character is blank. If the string is longer than length, it is truncated at both ends to fit. If an odd number of characters is truncated or added, the right-hand end loses or gains one more character than the left-hand end.

Here are some examples:

### Example 7.18. Builtin function CENTER

| CENTER(abc,7)               | -> | "  ABC  " |
| CENTER(abc,8,"-")          | -> | "--ABC---" |
| CENTRE("The blue sky",8)   | -> | "e blue s" |
| CENTRE("The blue sky",7)   | -> | "e blue " |

---

**Note**

To avoid errors because of the difference between British and American spellings, this function can be called either CENTRE or CENTER.

# 7.4.13. CHANGESTR

RETURNS a copy of haystack in which newneedle replaces occurrences of needle. If count is not specified, all occurrences of needle are replaced. If count is specified, it must be a non-negative, whole number that gives the maximum number of occurrences to be replaced.

Here are some examples:
Example 7.19. Builtin function CHANGESTR

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANGESTR(&quot;1&quot;, &quot;101100&quot;, &quot;)</td>
<td>&quot;000&quot;</td>
</tr>
<tr>
<td>CHANGESTR(&quot;1&quot;, &quot;101100&quot;, &quot;X&quot;)</td>
<td>&quot;X0X00&quot;</td>
</tr>
<tr>
<td>CHANGESTR(&quot;1&quot;, &quot;101100&quot;, &quot;X&quot;, 1)</td>
<td>&quot;X01100&quot;</td>
</tr>
</tbody>
</table>

7.4.14. CHARIN (Character Input)

Returns a string of up to \textit{length} characters read from the character input stream \textit{name}. (To understand the input and output functions, see \textit{Chapter 14, Input and Output Streams}. If you omit \textit{name}, characters are read from STDIN, which is the default input stream. The default \textit{length} is 1.

For persistent streams, a read position is maintained for each stream. Any read from the stream starts at the current read position by default. When the language processor completes reading, the read position is increased by the number of characters read. You can give a \textit{start} value to specify an explicit read position. This read position must be a positive whole number and within the bounds of the stream, and must not be specified for a transient stream. A value of 1 for \textit{start} refers to the first character in the stream. If \textit{start} is not a positive whole number the appropriate syntax condition is raised. When the read position is past the bounds of the stream, the empty string is returned and the NOTREADY condition is raised.

If you specify a \textit{length} of 0, then the read position is set to the value of \textit{start}, but no characters are read and the null string is returned.

In a transient stream, if there are fewer than \textit{length} characters available, the execution of the program generally stops until sufficient characters become available. If, however, it is impossible for those characters to become available because of an error or another problem, the NOTREADY condition is raised (see \textit{Section 14.5, “Errors during Input and Output”} and CHARIN returns with fewer than the requested number of characters.

Here are some examples:

Example 7.20. Builtin function CHARIN

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARIN(myfile, 1, 3)</td>
<td>&quot;MFC&quot;  /* the first 3 characters */</td>
</tr>
<tr>
<td>CHARIN(myfile, 1, 0)</td>
<td>&quot;&quot;     /* now at start */</td>
</tr>
<tr>
<td>CHARIN(myfile)</td>
<td>&quot;M&quot;    /* after last call */</td>
</tr>
<tr>
<td>CHARIN(myfile, 2)</td>
<td>&quot;FC&quot;   /* after last call */</td>
</tr>
</tbody>
</table>

/* Reading from the default input (here, the keyboard) */
/* User types "abcd efg" */
CHARIN()          | "a"    /* default is */ |

/* 1 character */
CHARIN(, 5)       | "bcd e"

Notes:

1. CHARIN returns all characters that appear in the stream, including control characters such as line-end and end-of-file.
2. When CHARIN reads from the keyboard, program execution stops until you press the Enter key.

### 7.4.15. CHAROUT (Character Output)

Returns the count of characters remaining after attempting to write string to the character output stream name. (To understand the input and output functions, see Chapter 14, Input and Output Streams.) If you omit name, characters in string are written to STDOUT (generally the display), which is the default output stream. The string can be a null string, in which case no characters are written to the stream, and 0 is always returned.

For persistent streams, a write position is maintained for each stream. Any write to the stream starts at the current write position by default. When the language processor completes writing, the write position is increased by the number of characters written. When the stream is first opened, the write position is at the end of the stream so that calls to CHAROUT append characters to the end of the stream.

You can give a start value to specify an explicit write position for a persistent stream. This write position must be a positive whole number. A value of 1 for start refers to the first character in the stream.

You can omit the string for persistent streams. In this case, the write position is set to the value of start that was given, no characters are written to the stream, and 0 is returned. If you do not specify start or string, the stream is closed and 0 is returned.

Execution of the program usually stops until the output operation is complete.

For example, when data is sent to a printer, the system accepts the data and returns control to Rexx, even though the output data might not have been printed. Rexx considers this to be complete, even though the data has not been printed. If, however, it is impossible for all the characters to be written, the NOTREADY condition is raised (see Section 14.5, “Errors during Input and Output”) and CHAROUT returns with the number of characters that could not be written (the residual count).

Here are some examples:

<table>
<thead>
<tr>
<th>Example 7.21. Builtin function CHAROUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAROUT(myfile,&quot;Hi&quot;) -&gt; 0 /* typically */</td>
</tr>
<tr>
<td>CHAROUT(myfile, &quot;Hi&quot;,5) -&gt; 0 /* typically */</td>
</tr>
<tr>
<td>CHAROUT(myfile, ,6) -&gt; 0 /* now at char 6 */</td>
</tr>
<tr>
<td>CHAROUT(myfile) -&gt; 0 /* at end of stream */</td>
</tr>
<tr>
<td>CHAROUT(&quot;Hi&quot;) -&gt; 0 /* typically */</td>
</tr>
<tr>
<td>CHAROUT(&quot;Hello&quot;) -&gt; 2 /* maybe */</td>
</tr>
</tbody>
</table>

**Note**

This routine is often best called as a subroutine. The residual count is then available in the variable RESULT.
For example:

**Example 7.22. Builtin function CHAROUT**

```
Call CHAROUT myfile,"Hello"
Call CHAROUT myfile,"Hi",6
Call CHAROUT myfile
```

### 7.4.16. CHARS (Characters Remaining)

```
CHARS(name)
```

Returns the total number of characters remaining in the character input stream `name`. The count includes any line separator characters, if these are defined for the stream. In the case of persistent streams, it is the count of characters from the current read position. (See *Chapter 14, Input and Output Streams* for a discussion of Rexx input and output.) If you omit `name`, the number of characters available in the default input stream (STDIN) is returned.

The total number of characters remaining cannot be determined for some streams (for example, STDIN or Windows/Unix-like system devices). For these streams, the CHARS function returns 1 to indicate that data is present, or 0 if no data is present.

**Example 7.23. Builtin function CHARS**

```
CHARS(myfile) -> 42 /* perhaps */
CHARS(nonfile) -> 0
CHARS() -> 1 /* perhaps */
```

### 7.4.17. COMPARE

```
COMPARE(string1, string2, pad)
```

Returns 0 if the strings `string1` and `string2` are identical. Otherwise, it returns the position of the first character that does not match. The shorter string is padded on the right with `pad` if necessary. The default `pad` character is a blank.

**Example 7.24. Builtin function COMPARE**

```
COMPARE("abc", "abc")        -> 0
COMPARE("abc", "ak")         -> 2
COMPARE("ab ", "ab")         -> 0
COMPARE("ab ", "ab", " ")   -> 0
COMPARE("ab ", "ab", "x")   -> 3
COMPARE("ab-- ", "ab", "-" )-> 5
```

### 7.4.18. CONDITION
Returns the condition information associated with the current trapped condition. (See Chapter 11, Conditions and Condition Traps for a description of condition traps.) You can request the following pieces of information:

- The name of the current trapped condition
- Any descriptive string associated with that condition
- Any condition-specific information associated with the current trapped condition
- The instruction processed as a result of the condition trap (CALL or SIGNAL)
- The status of the trapped condition

In addition, you can request a condition object containing all of the preceding information.

To select the information to be returned, use the following options. (Only the capitalized letter is needed; all characters following it are ignored.)

- **Additional** returns any additional object information associated with the current trapped condition. See Section 11.3.2, “Additional Object Information” for a list of possible values. If no additional object information is available or no condition has been trapped, the language processor returns .nil.

- **Condition name** returns the name of the current trapped condition. For user conditions, the returned string is a concatenation of the word USER and the user condition name, separated by a whitespace character.

- **Description** returns any descriptive string associated with the current trapped condition. See Section 11.3.1, “Descriptive Strings” for the list of possible values. If no description is available or no condition has been trapped, it returns a null string.

- **Extra** returns the Rexx error subcode associated with a trapped SYNTAX condition. If no SYNTAX condition has been trapped, it returns a null string.

- **Instruction** returns either CALL or SIGNAL, the keyword for the instruction processed when the current condition was trapped. This is the default if you omit option. If no condition has been trapped, it returns a null string.

- **Object** returns an object that contains all the information about the current trapped condition. See Section 11.3.5, “Condition Object” for more information. If no condition has been trapped, it returns .nil.

- **Reset** resets any currently trapped condition and returns the null string. After a reset, all CONDITION options will return their default values, as if no condition has been trapped.
**Status**

returns the status of the current trapped condition. This can change during processing, and is one of the following:

- **ON** - the condition is enabled
- **OFF** - the condition is disabled
- **DELAY** - any new occurrence of the condition is delayed or ignored

If no condition has been trapped, a null string is returned.

**Example 7.25. Builtin function CONDITION**

| CONDITION() | -> | "CALL" /* perhaps */ |
| CONDITION("C") | -> | "FAILURE" |
| CONDITION("I") | -> | "CALL" |
| CONDITION("D") | -> | "FailureTest" |
| CONDITION("S") | -> | "OFF" /* perhaps */ |

**Note**

The CONDITION function returns condition information that is saved and restored across subroutine calls (including those a CALL ON condition trap causes). Therefore, after a subroutine called with CALL ON trapname has returned, the current trapped condition reverts to the condition that was current before the CALL took place (which can be none). CONDITION returns the values it returned before the condition was trapped.

### 7.4.19. COPIES

COPIES(\textit{string}, \textit{n})

Returns \textit{n} concatenated copies of \textit{string}. The \textit{n} must be a positive whole number or zero.

**Example 7.26. Builtin function COPIES**

| COPIES("abc",3) | -> | "abcabcabc" |
| COPIES("abc",0) | -> | "" |

### 7.4.20. COUNTSTR

COUNTSTR(\textit{needle}, \textit{haystack})

Returns a count of the occurrences of \textit{needle} in \textit{haystack} that do not overlap.

**Example 7.27. Builtin function COUNTSTR**

| COUNTSTR("1","101101") | -> | 4 |
7.4.21. D2C (Decimal to Character)

D2C(whole number, n)

Returns a string, in character format, that is the ASCII representation of the decimal number. If you specify n, it is the length of the final result in characters; leading "00"x (for a positive whole number) or "FF"x (for a negative whole number) characters are added to the result string as necessary. n must be a positive whole number or zero.

Whole number must not have more digits than the current setting of NUMERIC DIGITS.

If you omit n, whole number must be a positive whole number or zero, and the result length is as needed. Therefore, the returned result has no leading "00"x characters.

Example 7.28. Builtin function D2C

<table>
<thead>
<tr>
<th>whole number</th>
<th>n</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td></td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>65,1</td>
<td></td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>65,2</td>
<td></td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>65,5</td>
<td></td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>109</td>
<td></td>
<td>&quot;m&quot;</td>
</tr>
<tr>
<td>-109,1</td>
<td></td>
<td>&quot;ô&quot;</td>
</tr>
<tr>
<td>76,2</td>
<td></td>
<td>&quot;L&quot;</td>
</tr>
<tr>
<td>-180,2</td>
<td></td>
<td>&quot;L&quot;</td>
</tr>
</tbody>
</table>

7.4.22. D2X (Decimal to Hexadecimal)

D2X(whole number, n)

Returns a string, in character format, that represents whole number, a decimal number, converted to hexadecimal. The returned string uses uppercase alphabets for the values A-F and does not include whitespace characters.

Whole number must not have more digits than the current setting of NUMERIC DIGITS.

If you specify n, it is the length of the final result in characters. After conversion the input string is sign-extended to the required length. If the number is too big to fit n characters, it is truncated on the left. n must be a positive whole number or zero.

If you omit n, whole number must be a positive whole number or zero, and the returned result has no leading zeros.

Example 7.29. Builtin function D2X

<table>
<thead>
<tr>
<th>whole number</th>
<th>n</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td></td>
<td>&quot;9&quot;</td>
</tr>
<tr>
<td>129</td>
<td></td>
<td>&quot;81&quot;</td>
</tr>
<tr>
<td>129,1</td>
<td></td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>129,2</td>
<td></td>
<td>&quot;81&quot;</td>
</tr>
<tr>
<td>129,4</td>
<td></td>
<td>&quot;8081&quot;</td>
</tr>
</tbody>
</table>
7.4.23. DATATYPE

Returns **NUM** if you specify only *string* and if *string* is a valid Rexx number that can be added to 0 without error; returns **CHAR** if *string* is not a valid number.

If you specify *type*, it returns 1 if *string* matches the type. Otherwise, it returns 0. If *string* is null, the function returns 0 (except when the *type* is **X** or **B**, for which **DATATYPE** returns 1 for a null string). The following are valid **types**. (Only the capitalized letter, or 9 for the **9Digits** option, is needed; all characters following it are ignored. For example, for the **hexadecimal** option, you must start your string specifying the name of the option with **X** rather than **H**.)

**Alphanumeric**
returns 1 if *string* contains only characters from the ranges **a-z**, **A-Z**, and 0-9.

**Binary**
returns 1 if *string* contains only the characters 0 or 1, or whitespace. Whitespace characters can appear only between groups of 4 binary characters. It also returns 1 if *string* is a null string, which is a valid binary string.

**Internal whole number**
returns 1 if *string* is a Rexx whole number that **built-in functions** will accept. Rexx built-in functions internally work with **NUMERIC DIGITS 9** for 32-bit systems or **NUMERIC DIGITS 18** for 64-bit systems.

**Lowercase**
returns 1 if *string* contains only characters from the range **a-z**.

**Mixed case**
returns 1 if *string* contains only characters from the ranges **a-z** and **A-Z**.

**Number**
returns 1 if **DATATYPE(string)** returns **NUM**.

**Logical**
returns 1 if the string is exactly "0" or "1". Otherwise it returns 0.

**Symbol**
returns 1 if *string* is a valid symbol, that is, if **SYMBOL(string)** does not return **BAD**. (See Section 1.10.4.4, “Symbols”.) Note that both uppercase and lowercase alphabetics are permitted.

**Uppercase**
returns 1 if *string* contains only characters from the range **A-Z**.

**Variable**
returns 1 if *string* could appear on the left-hand side of an assignment without causing a SYNTAX condition.
Whole number
returns 1 if string is a Rexx whole number under the current setting of NUMERIC DIGITS.

Hexadecimal
returns 1 if string contains only characters from the ranges a-f, A-F, 0-9, and whitespace (as long as the whitespace characters appear only between pairs of hexadecimal characters). It also returns 1 if string is a null string, which is a valid hexadecimal string.

9 digits
returns 1 if DATATYPE(string, "W") returns 1 when NUMERIC DIGITS is set to 9.

Here are some examples:

Example 7.30. Built-in function DATATYPE

<table>
<thead>
<tr>
<th>string</th>
<th>DATATYPE(string, &quot;W&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot; 12 &quot;</td>
<td>&quot;NUM&quot;</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;CHAR&quot;</td>
</tr>
<tr>
<td>&quot;123&quot;</td>
<td>&quot;CHAR&quot;</td>
</tr>
<tr>
<td>&quot;12.3&quot;, &quot;N&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;12.3&quot;, &quot;W&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;Fred&quot;, &quot;M&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Fred&quot;, &quot;U&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;Fred&quot;, &quot;L&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;?20K&quot;, &quot;s&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;BCd3&quot;, &quot;X&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;BC d3&quot;, &quot;X&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;1&quot;, &quot;O&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;11&quot;, &quot;0&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

7.4.24. DATE

DATE("N", string, "N", osep, isep)

Returns, by default, the local date in the format: dd mon yyyy (day month year—for example, 14 May 2017), with no leading zero or blank on the day. The first three characters of the English name of the month are used.
You can use the following options to obtain specific formats. (Only the capitalized letter is needed; all characters following it are ignored.)

**Base**
returns the number of complete days (that is, not including the current day) since and including the base date, 1 January 0001, in the format: **dddddd** (no leading zeros or whitespace). The expression `DATE("B")//7` returns a number in the range **0-6** that corresponds to the current day of the week, where **0** is Monday and **6** is Sunday.

**Note**
The base date of 1 January 0001 is determined by extending the current Gregorian calendar backward (365 days each year, with an extra day every year that is divisible by 4 except century years that are not divisible by 400). It does not take into account any errors in the calendar system that created the Gregorian calendar originally.

**Days**
returns the number of days, including the current day, that have passed this year in the format **ddd** (no leading zeros or whitespace).

**European**
returns the date in the format **dd/mm/yy**.

**Full**
returns the number of microseconds since **00:00:00.000000** on 1 January 0001, in the format: **dddddddddddddddddd** (no leading zeros or whitespace).

**Notes**
The base date of 1 January 0001 is determined by extending the current Gregorian calendar backward (365 days each year, with an extra day every year that is divisible by 4 except century years that are not divisible by 400). It does not take into account any errors in the calendar system that created the Gregorian calendar originally.

The value returned by `Date('F')` can be used to calculate the interval between any two dates. Note, however, that values returned generally contain more digits than the default NUMERIC DIGITS setting. The NUMERIC DIGITS setting should be increased to a minimum value of **18** when performing timestamp arithmetic.

**ISO**
Returns the date according to the format specified by ISO 8601. The format is **yyyy-mm-dd** with leading zeroes if required.

**Language**
returns the date in an implementation- and language-dependent, or local, date format. The format is **dd month yyyy**. The name of the month is according to the national language installed on the system. If no local date format is available, the default format is returned.
This format is intended to be used as a whole; Rexx programs must not make any assumptions about the form or content of the returned string.

**Month**
returns the full English name of the current month, for example, **August**.

**Normal**
returns the date in the format *dd mon yyyy*. This is the default.

**Ordered**
returns the date in the format *yy/mm/dd* (suitable for sorting, for example).

**Standard**
returns the date in the format *yyyymmdd* (suitable for sorting, for example).

**Ticks**
returns the number of seconds since 00:00:00.000000 on 1 January 1970, in the format: 
*dddddddddddd* (no leading zeros or whitespace).

The value returned by Date('T') can be used to calculate the interval between any two dates. Note, however, that values returned generally contain more digits than the default NUMERIC DIGITS setting. The NUMERIC DIGITS setting should be increased to a minimum value of 12 when performing timestamp arithmetic.

Date('T') will return a negative number for dates prior to 1 January 1970.

**Usa**
returns the date in the format *mm/dd/yy*.

**Weekday**
returns the English name for the day of the week, in mixed case, for example, **Tuesday**.

Here are some examples, assuming today is 24 Nov 2016:

<table>
<thead>
<tr>
<th>Example 7.31. Builtin function DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE() -- 24 Nov 2016</td>
</tr>
<tr>
<td>DATE(&quot;B&quot;) -- 736291</td>
</tr>
<tr>
<td>DATE(&quot;D&quot;) -- 329</td>
</tr>
<tr>
<td>DATE(&quot;E&quot;) -- 24/11/16</td>
</tr>
<tr>
<td>DATE(&quot;F&quot;) -- 6361568698459000</td>
</tr>
<tr>
<td>DATE(&quot;I&quot;) -- 2016-11-24</td>
</tr>
<tr>
<td>DATE(&quot;L&quot;) -- 24 November 2016</td>
</tr>
<tr>
<td>DATE(&quot;M&quot;) -- November</td>
</tr>
<tr>
<td>DATE(&quot;N&quot;) -- 24 Nov 2016</td>
</tr>
<tr>
<td>DATE(&quot;O&quot;) -- 16/11/24</td>
</tr>
<tr>
<td>DATE(&quot;S&quot;) -- 20161124</td>
</tr>
<tr>
<td>DATE(&quot;T&quot;) -- 1480063888</td>
</tr>
<tr>
<td>DATE(&quot;U&quot;) -- 11/24/16</td>
</tr>
<tr>
<td>DATE(&quot;W&quot;) -- Thursday</td>
</tr>
</tbody>
</table>
The first call to DATE or TIME in one clause causes a time stamp to be made that is then used for all calls to these functions in that clause. Therefore, several calls to any of the DATE or TIME functions, or both, in a single expression or clause are consistent with each other.

If you specify string, DATE returns the date corresponding to string in the format option. The string must be supplied in the format option2. The option2 format must specify day, month, and year (that is, not "D", "L", "M", or "W"). The default for option2 is "N", so you need to specify option2 if string is not in the Normal format. Here are some examples:

<table>
<thead>
<tr>
<th>Example 7.32. Builtin function DATE set</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE(&quot;O&quot;, &quot;13 Feb 1923&quot;)</td>
</tr>
<tr>
<td>DATE(&quot;O&quot;, &quot;06/01/50&quot;, &quot;U&quot;)</td>
</tr>
<tr>
<td>DATE(&quot;N&quot;, &quot;63630372300000000&quot;, &quot;f&quot;)</td>
</tr>
</tbody>
</table>

If you specify an output separator character osep, the days, month, and year returned are separated by this character. Any nonalphanumeric character or an empty string can be used. A separator character is only valid for the formats "E", "I", "N", "O", "S", and "U". Here are some examples:

<table>
<thead>
<tr>
<th>Example 7.33. Builtin function DATE set</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE(&quot;S&quot;, &quot;13 Feb 2017&quot;, &quot;N&quot;, &quot;-&quot; )</td>
</tr>
<tr>
<td>DATE(&quot;N&quot;, &quot;13 Feb 2017&quot;, &quot;N&quot;, &quot;&quot; )</td>
</tr>
<tr>
<td>DATE(&quot;O&quot;, &quot;06/01/50&quot;, &quot;U&quot;, &quot;&quot; )</td>
</tr>
<tr>
<td>DATE(&quot;E&quot;, &quot;02/13/17&quot;, &quot;U&quot;, &quot;-&quot; )</td>
</tr>
<tr>
<td>DATE(&quot;N&quot;, , ,&quot;-&quot; )</td>
</tr>
</tbody>
</table>

In this way, formats can be created that are derived from their respective default format, which is the format associated with option using its default separator character. The default separator character for each of these formats is:

<table>
<thead>
<tr>
<th>Option</th>
<th>Default separator</th>
</tr>
</thead>
<tbody>
<tr>
<td>European</td>
<td>&quot;/&quot;</td>
</tr>
<tr>
<td>ISO</td>
<td>&quot;/&quot;</td>
</tr>
<tr>
<td>Normal</td>
<td>&quot;-&quot;</td>
</tr>
<tr>
<td>Ordered</td>
<td>&quot;/&quot;</td>
</tr>
<tr>
<td>Standard</td>
<td>&quot;&quot; (empty string)</td>
</tr>
<tr>
<td>Usa</td>
<td>&quot;/&quot;</td>
</tr>
</tbody>
</table>

If you specify a string containing a separator that is different from the default separator character of option2, you must also specify isep to indicate which separator character is valid for string. Basically, any date format that can be generated with any valid separator character can be used as input date string as long as its format has the generalized form specified by option2 and its separator character matches the character specified by isep.
Here are some examples:

**Example 7.34. Builtin function DATE set**

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DATE(&quot;S&quot;, &quot;2017-11-13&quot;, &quot;S&quot;, &quot;&quot;, &quot;,&quot;)</code></td>
<td>&quot;20171113&quot;</td>
</tr>
<tr>
<td><code>DATE(&quot;S&quot;, &quot;13-Nov-2017&quot;, &quot;N&quot;, &quot;&quot;, &quot;,&quot;)</code></td>
<td>&quot;20171113&quot;</td>
</tr>
<tr>
<td><code>DATE(&quot;O&quot;, &quot;06*01*50&quot;,&quot;U&quot;, &quot;&quot;, &quot;,&quot;)</code></td>
<td>&quot;500601&quot;</td>
</tr>
</tbody>
</table>

You can determine the number of days between two dates; for example:

**Example 7.35. Builtin function DATE days**

```
say date("B", "12/25/17", "USA") - date("B") "shopping days till Christmas!"
```

If *string* does not include the century but *option* defines that the century be returned as part of the date, the century is determined depending on whether the year to be returned is within the past 50 years or the next 49 years. Assume, for example, that you specify 10/15/61 for *string* and today's date is 10/27/2016. In this case, 1961 would be 55 years ago and 2061 would be 45 years in the future. So, 10/15/2061 would be the returned date.

**Note**

This rule is suitable for dates that are close to today's date. However, when working with birth dates, it is recommended that you explicitly provide the century.

When requesting dates to be converted to Full format or Ticks format, a time value of "00:00:00.000000" is used for the conversion. A time stamp for a time and date combination can be created by combining a value from Time() for the time of day.

**Example 7.36. Builtin function DATE timestamp**

```
numeric digits 18    -- needed to add the timestamps
timestamp = date("f", "20170123", "S") + time("f", "08:14:22", "N")
```

### 7.4.25. DELSTR (Delete String)

![DELSTR Diagram]

Returns *string* after deleting the substring that begins at the *n*th character and is of *length* characters. If *n* is omitted, it defaults to 1. If you omit *length*, or if *length* is greater than the number of characters from *n* to the end of *string*, the function deletes the rest of *string* (including the *n*th character). The *length* must be a positive whole number or zero. *n* must be a positive whole number. If *n* is greater than the length of *string*, the function returns *string* unchanged.
Example 7.37. Builtin function DELSTR

```
DELSTR("abcd",3)       ->    "ab"
DELSTR("abcde",3,2)    ->    "abe"
DELSTR("abcde",6)      ->    "abcde"
```

7.4.26. DELWORD (Delete Word)

Returns string after deleting the substring that starts at the n
th word and is of length whitespace-delimited words. If you omit length, or if length is greater than the
number of words from n to the end of string, the function deletes the remaining words in string (including the
nth word). The length must be a positive whole number or zero. n must be a positive whole number. If n is greater than the
number of words in string, the function returns string unchanged. The string deleted includes any
whitespace characters following the final word involved but none of the whitespace preceding the first
word involved.

Example 7.38. Builtin function DELWORD

```
DELWORD("Now is the time",2,2)  ->  "Now time"
DELWORD("Now is the time",3)    ->  "Now is "
DELWORD("Now is the time",5)    ->  "Now is the time"
DELWORD("Now is the time",3,1)  ->  "Now is time"
DELWORD("Now is the time",2,2)  ->  "Now time"
```

7.4.27. DIGITS

Returns the current setting of NUMERIC DIGITS.

Example 7.39. Builtin function DIGITS

```
DIGITS()    ->    9      /* by default */
```

7.4.28. DIRECTORY

Returns the current directory, first changing it to newdirectory if an argument is supplied and the
named directory exists. If newdirectory is not specified, the name of the current directory is returned.
Otherwise, an attempt is made to change to the specified newdirectory. If successful, the name of the
newdirectory is returned; if an error occurred, null is returned.
On Windows the returned string includes a drive letter prefix as the first two characters of the directory name. Specifying a drive letter prefix as part of newdirectory causes the specified drive to become the current drive. If a drive letter is not specified, the current drive remains unchanged.

For example, the following program fragment saves the current directory and switches to a new directory; it performs an operation there, and then returns to the former directory.

```
Example 7.40. Builtin function DIRECTORY

/* get current directory */
curdir = directory()

/* go play a game */
newdir = directory("/usr/games") /* Linux type subdirectory */
if newdir = "/usr/games" then
do
   fortune /* tell a fortune */
/* return to former directory */
call directory curdir
else
   say "Can't find /usr/games"
```

### 7.4.29. ENDLOCAL (Linux only)

Restores the directory and environment variables in effect before the last SETLOCAL function was run. If ENDLOCAL is not included in a procedure, the initial environment saved by SETLOCAL is restored upon exiting the procedure.

ENDLOCAL returns a value of 1 if the initial environment is successfully restored and a value of 0 if no SETLOCAL was issued or the action is otherwise unsuccessful.

```
Example 7.41. Builtin function ENDLOCAL

n = SETLOCAL()                   /* saves the current environment */
   /* The program can now change environment variables
      (with the VALUE function) and then work in the
      changed environment. */
n = ENDLOCAL()                   /* restores the initial environment */
```

For additional examples, see Section 7.4.56, “SETLOCAL (Linux only)”.

### 7.4.30. ERRORTEXT

```
ERRORTEXT( n )
```

Returns the Rexx error message associated with error number \( n \). \( n \) must be in the range \( 0-99 \). It returns the null string if \( n \) is in the allowed range but is not a defined Rexx error number. See Appendix C, Error Numbers and Messages for a complete description of error numbers and messages.
Example 7.42. Builtin function ERRORTEXT

| ERRORTEXT(16) | -> | "Label not found." |
| ERRORTEXT(60) | -> | "" |

7.4.31. FILESPEC

```
FILESPEC( option, filespec )
```

Returns a selected element of `filespec`, given file specification, identified by one of the following `option` strings. Only the capitalized `option` letter is needed; all characters following it are ignored.

**Drive**

On Windows, returns the drive letter of the given `filespec`. On Unix-like systems, returns a null string ("").

**Path**

Returns the directory path of the given `filespec`.

**Location**

Returns the full location portion of the given `filespec`. On Windows, this will include both the Drive and Path information. On other platforms, this returns the same result as the Path option.

**Name**

Returns the file name of the given `filespec`.

**Extension**

Returns the extension portion of the `filespec` file name.

If `filespec` does not contain the requested information, FILESPEC returns a null string ("").

Example 7.43. Builtin function FILESPEC

```
thisfile = "C:\WINDOWS\UTIL\SYSTEM.INI"
say FILESPEC("drive",thisfile) /* says "C:" */  
say FILESPEC("path",thisfile) /* says "\WINDOWS\UTIL\" */  
say FILESPEC("location",thisfile) /* says "C:\\WINDOWS\UTIL\" */  
say FILESPEC("name",thisfile) /* says "SYSTEM.INI" */  
say FILESPEC("extension",thisfile) /* says "INI" */  
part = "name"
say FILESPEC(part,thisfile) /* says "SYSTEM.INI" */
```

7.4.32. FORM

```
FORM()
```

Returns the current setting of `NUMERIC FORM`.
Example 7.44. Builtin function FORM

```
FORM() -> "SCIENTIFIC" /* by default */
```

### 7.4.33. FORMAT

Returns `number`, rounded and formatted.

The `number` is first rounded according to standard Rexx rules, as though the operation `number+0` had been carried out. The result is precisely that of this operation if you specify only `number`. If you specify any other options, the `number` is formatted as described in the following.

The `before` and `after` options describe how many characters are used for the integer and decimal parts of the result, respectively. If you omit either or both of them, the number of characters used for that part is as needed.

If `before` is not large enough to contain the integer part of the number (plus the sign for a negative number), an error results. If `before` is larger than needed for that part, the number is padded on the left with blanks. If `after` is not the same size as the decimal part of the number, the number is rounded (or extended with zeros) to fit. Specifying `0` causes the number to be rounded to an integer.

Example 7.45. Builtin function FORMAT

| FORMAT( | number       | , , expp | expt) | -> | "3" |
| FORMAT("1.73", 4, 0) | -> | "2" |
| FORMAT("1.73", 4, 3) | -> | "1.738" |
| FORMAT("- .76", 4, 1) | -> | "-0.8" |
| FORMAT("3.03", 4) | -> | "3.03" |
| FORMAT(" - 12.73", , 4) | -> | "-12.7300" |
| FORMAT(" - 12.73") | -> | "-12.73" |
| FORMAT("0.000") | -> | "0" |

The first three arguments are as described previously. In addition, `expp` and `expt` control the exponent part of the result, which, by default, is formatted according to the current NUMERIC settings of DIGITS and FORM. `expp` sets the number of places for the exponent part; the default is to use as many as needed (which can be zero). `expt` specifies when the exponential expression is used. The default is the current setting of NUMERIC DIGITS.

If `expp` is `0`, the number is not in exponential notation. If `expp` is not large enough to contain the exponent, an error results.

If the number of places needed for the integer or decimal part exceeds `expt` or twice `expt`, respectively, the exponential notation is used. If `expt` is `0`, the exponential notation is always used unless the exponent would be `0`. (If `expp` is `0`, this overrides a `0` value of `expt`.) If the exponent would be `0` when a nonzero `expp` is specified, then `expp+2` blanks are supplied for the exponent part of the result. If the exponent would be `0` and `expp` is not specified, the number is not an exponential expression.
Example 7.46. Builtin function FORMAT

<table>
<thead>
<tr>
<th>Format Call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORMAT(&quot;12345.73&quot;, , ,2,2)</td>
<td>&quot;1.234573E+04&quot;</td>
</tr>
<tr>
<td>FORMAT(&quot;12345.73&quot;, ,3, ,0)</td>
<td>&quot;1.235E+4&quot;</td>
</tr>
<tr>
<td>FORMAT(&quot;1.234573&quot;, ,3, ,0)</td>
<td>&quot;1.235&quot;</td>
</tr>
<tr>
<td>FORMAT(&quot;12345.73&quot;, ,3,6)</td>
<td>&quot;12345.73&quot;</td>
</tr>
<tr>
<td>FORMAT(&quot;1234567e5&quot;, ,3,0)</td>
<td>&quot;123456700000.000&quot;</td>
</tr>
</tbody>
</table>

7.4.34. FUZZ

FUZZ()

Returns the current setting of NUMERIC FUZZ.

Example 7.47. Builtin function FUZZ

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUZZ()</td>
<td>0</td>
</tr>
</tbody>
</table>

/* by default */

7.4.35. INSERT

Inserts the string new, padded or truncated to length length, into the string target after the nth character. The default value for n is 0, which means insertion before the beginning of the string. If specified, n and length must be positive whole numbers or zero. If n is greater than the length of the target string, the string new is padded at the beginning. The default value for length is the length of new. If length is less than the length of the string new, then INSERT truncates new to length length. The default pad character is a blank.

Here are some examples:

Example 7.48. Builtin function INSERT

<table>
<thead>
<tr>
<th>Insert Call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT(&quot; &quot;, &quot;abcdef&quot;, 3)</td>
<td>&quot;abc def&quot;</td>
</tr>
<tr>
<td>INSERT(&quot;123&quot;,&quot;abc&quot;, 5, 6)</td>
<td>&quot;abc 123 &quot;</td>
</tr>
<tr>
<td>INSERT(&quot;123&quot;,&quot;abc&quot;, 5, 6,&quot;+&quot;&quot;)</td>
<td>&quot;abc++123+++&quot;</td>
</tr>
<tr>
<td>INSERT(&quot;123&quot;,&quot;abc&quot;)</td>
<td>&quot;123abc&quot;</td>
</tr>
<tr>
<td>INSERT(&quot;123&quot;,&quot;abc&quot;, ,5, &quot;.&quot; )</td>
<td>&quot;123--abc&quot;</td>
</tr>
</tbody>
</table>

7.4.36. LASTPOS (Last Position)

Returns the position of the last occurrence of one string, needle, in another, haystack. It returns 0 if needle is a null string or not found. By default, the search starts at the last character of haystack and scans backward to the beginning of the string. You can override this by specifying start, the point...
at which the backward scan starts and \textit{length}, the range of characters to scan. The \textit{start} must be a positive whole number and defaults to the length of \textit{haystack} if larger than that value or omitted. The \textit{length} must be a non-negative whole number and defaults to \textit{start}.

See also \textit{POS} built-in function.

Here are some examples:

\textbf{Example 7.49. Builtin function \texttt{LASTPOS}}

\begin{verbatim}
LASTPOS( " ", "abc def ghi") -> 8
LASTPOS( " ", "abcdefghi") -> 0
LASTPOS("xy","efgxyz") -> 4
LASTPOS(" ","abc def ghi",7) -> 4
LASTPOS(" ","abc def ghi",7,3) -> 0
\end{verbatim}

\section*{7.4.37. LEFT}

\begin{equation}
\text{LEFT}(\text{string}, \text{length})
\end{equation}

Returns a string of length \textit{length}, containing the leftmost \textit{length} characters of \textit{string}. The string returned is padded with \textit{pad} characters, or truncated, on the right as needed. The default \textit{pad} character is a blank. \textit{length} must be a positive whole number or zero. The \textit{LEFT} function is equivalent to \texttt{SUBSTR(string, 1, length [, pad])}.

Here are some examples:

\textbf{Example 7.50. Builtin function \texttt{LEFT}}

\begin{verbatim}
LEFT("abc d",8) -> "abc d   \\
LEFT("abc d",8,".") -> "abc d..." \\
LEFT("abc  def",7) -> "abc  de"
\end{verbatim}

\section*{7.4.38. LENGTH}

\begin{equation}
\text{LENGTH(\text{string})}
\end{equation}

Returns the length of \textit{string}.

Here are some examples:

\textbf{Example 7.51. Builtin function \texttt{LENGTH}}

\begin{verbatim}
LENGTH("abcdefg") -> 8 \\
LENGTH("abc defg") -> 8 \\
LENGTH("") -> 0
\end{verbatim}

\section*{7.4.39. LINEIN (Line Input)}
Returns count lines read from the character input stream name. The count must be 1 or 0 (To understand the input and output functions, see Chapter 14, Input and Output Streams.) If you omit name, the line is read from the default input stream, STDIN. The default count is 1.

For persistent streams, a read position is maintained for each stream. Any read from the stream starts at the current read position by default. Under certain circumstances, a call to LINEIN returns a partial line. This can happen if the stream has already been read with the CHARIN function, and part but not all of a line (and its termination, if any) has already been read. When the language processor completes reading, the read position is moved to the beginning of the next line.

A line number may be given to set the read position to the start of a specified line. This line number must be positive and within the bounds of the stream, and must not be specified for a transient stream. The read position can be set to the beginning of the stream by giving line a value of 1.

If you give a count of 0, then no characters are read and a null string is returned.

For transient streams, if a complete line is not available in the stream, then execution of the program usually stops until the line is complete. If, however, it is impossible for a line to be completed because of an error or another problem, the NOTREADY condition is raised (see Section 14.5, “Errors during Input and Output”) and LINEIN returns whatever characters are available.

Here are some examples:

**Example 7.52. Builtin function LINEIN**

```plaintext
LINEIN()                  /* Reads one line from the    */ /*  default input stream; */ /*  usually this is an entry */ /*  typed at the keyboard */

myfile = "ANYFILE.TXT"
LINEIN(myfile) -> "Current line" /* Reads one line from */ /* ANYFILE.TXT, beginning */ /* at the current read */ /* position. (If first call, */ /* file is opened and the */ /* first line is read.) */

LINEIN(myfile,1,1) -> "first line" /* Opens and reads the first */ /* line of ANYFILE.TXT (if */ /* the file is already open, */ /* reads first line); sets */ /* read position on the */ /* second line. */

LINEIN(myfile,1,0) -> "" /* No read; opens ANYFILE.TXT */ /* (if file is already open, */ /* sets the read position to */ /* the first line). */

LINEIN(myfile, ,0) -> "" /* No read; opens ANYFILE.TXT */ /* (no action if the file is */ /* already open). */

LINEIN("QUEUE:"), -> "Queue line" /* Read a line from the queue. */ /* If the queue is empty, the */ /* program waits until a line */ /* is put on the queue. */
```
If you want to read complete lines from the default input stream, as in a dialog with a user, use the \texttt{PULL} or \texttt{PARSE PULL} instruction.

The \texttt{PARSE LINEIN} instruction is also useful in certain cases.

### 7.4.40. LINEOUT (Line Output)

Returns 0 if successful in writing \texttt{string} to the character output stream \texttt{name}, or 1 if an error occurs while writing the line. (To understand the input and output functions, see Chapter 14, Input and Output Streams.) If you specify \texttt{line}, this positions the write pointer before writing. If you omit both \texttt{string} and \texttt{line}, the stream is closed. LINEOUT adds platform-specific line-end characters to the end of \texttt{string}.

If you omit \texttt{name}, the line is written to the default output stream STDOUT (usually the display).

For persistent streams, a write position is maintained for each stream. Any write to the stream starts at the current write position by default. (Under certain circumstances the characters written by a call to LINEOUT can be added to a partial line previously written to the stream with the CHAROUT routine. LINEOUT stops a line at the end of each call.) When the language processor completes writing, the write position is set to the beginning of the line following the one just written. When the stream is first opened, the write position is at the end of the stream, so that calls to LINEOUT append lines to the end of the stream.

You can specify a \texttt{line} number to set the write position to the start of a particular line in a persistent stream. This line number must be positive and within the bounds of the stream unless it is a binary stream (though it can specify the line number immediately after the end of the stream). A value of 1 for \texttt{line} refers to the first line in the stream. Note that, unlike CHAROUT, you cannot specify a position beyond the end of the stream for non-binary streams.

You can omit the \texttt{string} for persistent streams. If you specify \texttt{line}, the write position is set to the start of the \texttt{line} that was given, nothing is written to the stream, and the function returns 0. If you specify neither \texttt{line} nor \texttt{string}, the stream is closed. Again, the function returns 0.

Execution of the program usually stops until the output operation is effectively complete. For example, when data is sent to a printer, the system accepts the data and returns control to Rexx, even though the output data might not have been printed. Rexx considers this to be complete, even though the data has not been printed. If, however, it is impossible for a line to be written, the NOTREADY condition is raised (see Section 14.5, "Errors during Input and Output"), and LINEOUT returns a result of 1, that is, the residual count of lines written.

Here are some examples:

#### Example 7.53. Builtin function LINEOUT

```
LINEOUT(,"Display this")          /* Writes string to the default   */
/* output stream (usually, the    */
```
myfile = "ANYFILE.TXT"
LINEOUT(myfile,"A new line") /* Opens the file ANYFILE.TXT and */ /* appends the string to the end. */ /* If the file is already open, */ /* the string is written at the */ /* current write position. */ /* Returns 0 if successful. */
LINEOUT(myfile,"A new start",1) /* Opens the file (if not already */ /* open); overwrites first line */ /* with a new line. */ /* Returns 0 if successful. */
LINEOUT(myfile, ,1) /* Opens the file (if not already */ /* open). No write; sets write */ /* position at first character. */
LINEOUT(myfile) /* Closes ANYFILE.TXT */

LINEOUT is often most useful when called as a subroutine. The return value is then available in the
variable RESULT. For example:

Example 7.54. Builtin function LINEOUT call

Call LINEOUT "A:rexx.bat","Shell",1
Call LINEOUT ,"Hello"

Note

If the lines are to be written to the default output stream without the possibility of error, use the
SAY instruction instead.

7.4.41. LINES (Lines Remaining)

Returns 1 if any data remains between the current read position and the end of the character input
stream name, and no option or option Normal is specified. It returns 0 if no data remains. As such,
LINES reports whether a read action that CHARIN or LINEIN performs will succeed. Option Normal is the
default.

If option Count is specified, LINES returns the actual number of lines remaining in the stream,
irrespective of how long this operation takes.

For an explanation of input and output, see Chapter 14, Input and Output Streams.

Here are some examples:
Example 7.55. Builtin function LINES

```
LINES(myfile)    ->    0    /* at end of the file */
LINES()          ->    1    /* data remains in the default input stream */
```

Note

The CHARS function returns the number of characters in a persistent stream or the presence of data in a transient stream.

7.4.42. LOWER

Returns a new string with the characters of `string` beginning with character `n` for `length` characters converted to lowercase. If `n` is specified, it must be a positive whole number. If `n` is not specified, the case conversion will start with the first character. If `length` is specified, it must be a non-negative whole number. If `length` is not specified, the default is to convert the remainder of the string.

Here are some examples:

Example 7.56. Builtin function LOWER

```
lower("Albert Einstein")      ->    "albert einstein"
lower("ABCDEF", 4)            ->    "ABCdef"
lower("ABCDEF", 3, 2)         ->    "ABcdEF"
```

7.4.43. MAX (Maximum)

Returns the largest number of the list specified, formatted according to the current NUMERIC settings. You can specify any number of `numbers`.

Here are some examples:

Example 7.57. Builtin function MAX

```
MAX(12, 6, 7, 9)              ->    12
MAX(17.3, 19, 17.03)         ->    19
MAX(-7, -3, -4.3)            ->    -3
MAX(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21) ->    21
```
7.4.44. MIN (Minimum)

Returns the smallest number of the list specified, formatted according to the current NUMERIC settings. You can specify any number of numbers.

Here are some examples:

<table>
<thead>
<tr>
<th>Example 7.58. Builtin function MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN(12, 6, 7, 9)</td>
</tr>
<tr>
<td>MIN(17.3, 19, 17.03)</td>
</tr>
<tr>
<td>MIN(-7, -3, -4.3)</td>
</tr>
<tr>
<td>MIN(21, 26, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)</td>
</tr>
<tr>
<td>-&gt; 6</td>
</tr>
<tr>
<td>-&gt; 17.03</td>
</tr>
<tr>
<td>-&gt; -7</td>
</tr>
<tr>
<td>-&gt; 1</td>
</tr>
</tbody>
</table>

7.4.45. OVERLAY

Returns the string target, which, starting at the n-th character, is overlaid with the string new, padded or truncated to length length. The overlay may extend beyond the end of the original target string. If you specify length, it must be a positive whole number or zero. The default value for length is the length of new. If n is greater than the length of the target string, the string new is padded at the beginning. The default pad character is a blank, and the default value for n is 1. If you specify n, it must be a positive whole number.

Here are some examples:

<table>
<thead>
<tr>
<th>Example 7.59. Builtin function OVERLAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVERLAY( &quot; &quot;, &quot;abcdef&quot;, 3 )</td>
</tr>
<tr>
<td>OVERLAY( &quot;,&quot;, &quot;abcdef&quot;, 3, 2 )</td>
</tr>
<tr>
<td>OVERLAY( &quot;qq&quot;, &quot;abcd&quot; )</td>
</tr>
<tr>
<td>OVERLAY( &quot;qq&quot;, &quot;abcd&quot;, 4 )</td>
</tr>
<tr>
<td>OVERLAY(&quot;123&quot;, &quot;abc&quot;, 5, 6,&quot;+&quot; )</td>
</tr>
<tr>
<td>-&gt; &quot;ab def&quot;</td>
</tr>
<tr>
<td>-&gt; &quot;ab. ef&quot;</td>
</tr>
<tr>
<td>-&gt; &quot;qqcd&quot;</td>
</tr>
<tr>
<td>-&gt; &quot;abcqq&quot;</td>
</tr>
<tr>
<td>-&gt; &quot;abc+123+++&quot;</td>
</tr>
</tbody>
</table>

7.4.46. POS (Position)

Returns the position of one string, needle, in another, haystack. It returns 0 if needle is a null string or not found or if start is greater than the length of haystack. By default, the search starts at the first character of the receiving string (that is, the value of start is 1), and continues to the end of the string. You can override this by specifying start, the point at which the search starts, and length, the bounding limit for the search. If specified, start must be a positive whole number and length must be a non-negative whole number.
See also `LASTPOS` built-in function.

Here are some examples:

**Example 7.60. Builtin function POS**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>POS(&quot;day&quot;,&quot;Saturday&quot;)</code></td>
<td>6</td>
</tr>
<tr>
<td><code>POS(&quot;x&quot;,&quot;abc def ghi&quot;)</code></td>
<td>0</td>
</tr>
<tr>
<td><code>POS(&quot; &quot;,&quot;abc def ghi&quot;)</code></td>
<td>4</td>
</tr>
<tr>
<td><code>POS(&quot; &quot;,&quot;abc def ghi&quot;,5)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>POS(&quot; &quot;,&quot;abc def ghi&quot;,5,3)</code></td>
<td>0</td>
</tr>
</tbody>
</table>

### 7.4.47. QUALIFY

```
QUALIFY(name)
```

Returns the fully qualified path for `name` by expanding the file name into a path with absolute directory information. The file does not need to exist to generate the full name.

### 7.4.48. QUEUED

```
QUEUED()
```

Returns the number of lines remaining in the external data queue when the function is called. (See Chapter 14, *Input and Output Streams* for a discussion of Rexx input and output.)

Here is an example:

**Example 7.61. Builtin function QUEUED**

```
QUEUED() -> 5 /* Perhaps */
```

### 7.4.49. RANDOM

```
RANDOM(max, min, seed)
```

Returns a quasi-random whole number in the range `min` to `max` inclusive. If you specify `max` or `min,max`, then `max` minus `min` cannot exceed 999999999. `min` and `max` default to 0 and 999, respectively. To start a repeatable sequence of results, use a specific `seed` as the third argument. This `seed` must be a whole number from 0 to 999999999.

Here are some examples:

**Example 7.62. Builtin function RANDOM**

```
RANDOM() -- 305 /* 0 to 999 */
RANDOM(5, 8) -- 7 /* 5 to 8 */
```
Notes:

1. To obtain a predictable sequence of quasi-random numbers, use RANDOM a number of times, but specify a seed only the first time. For example, to simulate 40 throws of a 6-sided, unbiased die:

Example 7.63. Builtin function RANDOM with seed

```plaintext
sequence = RANDOM(1,6,12345)  /* any number would */
          /* do for a seed */
do 39
    sequence = sequence RANDOM(1,6)
end
say sequence
```

The numbers are generated mathematically, using the initial seed, so that as far as possible they appear to be random. Running the program again produces the same sequence; using a different initial seed almost certainly produces a different sequence. If you do not supply a seed, the first time RANDOM is called, an arbitrary seed is used. Hence, your program usually gives different results each time it is run.

2. The random number generator is global for an entire program; the current seed is not saved across internal routine calls.

### 7.4.50. REVERSE

Returns `string` reversed.

Here are some examples:

Example 7.64. Builtin function REVERSE

```plaintext
REVERSE("Abc.")  ->  "cba"
REVERSE("XYZ ")  ->  "ZXY"
```

### 7.4.51. RIGHT

Returns a string of length `length` containing the rightmost `length` characters of `string`. The string returned is padded with `pad` character, or truncated, on the left as needed. The default `pad` character is a blank. The `length` must be a positive whole number or zero.

Here are some examples:
Example 7.65. Builtin function RIGHT

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIGHT(&quot;abc  d&quot;, 8)</td>
<td>&quot; abc  d&quot;</td>
</tr>
<tr>
<td>RIGHT(&quot;abc def&quot;, 5)</td>
<td>&quot;c def&quot;</td>
</tr>
<tr>
<td>RIGHT(&quot;12&quot;, 5, &quot;0&quot;)</td>
<td>&quot;00012&quot;</td>
</tr>
</tbody>
</table>

7.4.52. RXFUNCADD

RxFuncAdd(

Registers the function, making it available to Rexx procedures. The module is the name of an external library where the native function is located. In some environments, such as Unix-like systems, the library name is case sensitive. The procedure is the name of the exported procedure inside of module. If procedure is not specified, it defaults to name. The procedure is generally case-sensitive. RxFuncAdd will attempt to resolve the procedure address using the name as specified and if that attempt fails, will retry using an uppercased name.

A return value 0 signifies successful registration and that the registered function has been located in the specified module. A return value 1 signifies that the function could not be resolved.

Example 7.66. Builtin function RXFUNCADD

```plaintext
// 0 if rxcalcsqrt can be located, else 1
rxfuncadd("rxcalcsqrt", "rxmath", "rxcalcsqrt")
```

7.4.53. RXFUNCDROP

RxFuncDrop(

Removes (deregisters) the function name from the list of available functions. A zero return value signifies successful removal.

Example 7.67. Builtin function RXFUNCDROP

```plaintext
rxfuncdrop("SysLoadFuncs")     -> 0 /**< if successfully removed */
```

7.4.54. RXFUCNQUERY

RxFuncQuery(

Queries the list of available functions for the function name. It returns a value of 0 if the function is registered, and a value of 1 if it is not.
Example 7.68. Builtin function RXFUNCQUERY

```
rxfuncquery("SysLoadFuncs")        -> 0 /* if registered */
```

7.4.55. RXQUEUE

Creates and deletes external data queues. It also sets and queries their names.

**Create**

creates a queue with the name `queue_name` if you specify `queue_name` and if no queue of that name exists already. You must not use SESSION as a `queue_name`. If you specify no `queue_name`, then the language processor provides a name. The name of the queue is returned in either case.

The maximum length of `queue_name` can be 250 characters.

Many queues can exist at the same time, and most systems have sufficient resources available to support several hundred queues at a time. If a queue with the specified name exists already, a queue is still created with a name assigned by the language processor. The assigned name is then returned to you.

**Delete**

deletes the named queue. It returns 0 if successful or a nonzero number if an error occurs. Possible return values are:

```
0
Queue has been deleted.

5
Not a valid queue name or tried to delete queue named "SESSION".

9
Specified queue does not exist.

10
Queue is busy; wait is active.

12
A memory failure has occurred.

1002
Failure in memory manager.
```
**Get**
returns the name of the queue currently in use.

**Exist**
returns `.true` if the specified `queue name` exists. Returns `.false` otherwise.

**Open**
tests if the specified `queue name` exists and creates if it does not.

**Set**
sets the name of the current queue to `new queue name` and returns the previously active queue name.

The first parameter determines the function. Only the first character of the first parameter is significant. The parameter can be entered in any case. The syntax for a valid queue name is the same as for a valid Rexx symbol.

The second parameter specified for Create, Set, and Delete must follow the same syntax rules as the Rexx variable names. There is no connection, however, between queue names and variable names. A program can have a variable and a queue with the same name. The actual name of the queue is the uppercase value of the name requested.

Named queues prevent different Rexx programs that are running in a single session from interfering with each other. They allow Rexx programs running in different sessions to synchronize execution and pass data. `LINEIN("QUEUE:" )` is especially useful because the calling program stops running until another program places a line on the queue.

### Example 7.69. Builtin function RXQUEUE

```plaintext
/* default queue                                    */
rxqueue("Get")             -> "SESSIoN"

/* assuming FRED does not already exist             */
rxqueue("Create", "Fred")  -> "FREDE"

/* assuming SESSION had been active                 */
rxqueue("Set", "Fred")     -> "SESSIoN"

/* assuming FRED exists                              */
rxqueue("delete", "Fred")   -> "0"
```

### 7.4.56. SETLOCAL (Linux only)

**SETLOCAL**
Saves the current working directory and the current values of the environment variables that are local to the current process.

For example, SETLOCAL can be used to save the current environment before changing selected settings with the `VALUE` function. To restore the directory and environment, use the `ENDLOCAL` function.

SETLOCAL returns a value of 1 if the initial directory and environment are successfully saved and a value of 0 if unsuccessful. If SETLOCAL is not followed by an ENDLOCAL function in a procedure, the initial environment saved by SETLOCAL is restored upon exiting the procedure.

Here is an example:
Example 7.70. Built-in function SETLOCAL

```
/* Current path is "user/bin" */
n = SETLOCAL()           /* saves all environment settings */
/* Now use the VALUE function to change the PATH variable */
p = VALUE("Path","home/user/bin"."ENVIRONMENT")
/* Programs in directory home/user/bin can now be run */
n = ENDLOCAL()           /* restores initial environment including */
/* the changed PATH variable, which is */
/* "user/bin" */
```

7.4.57. SIGN

\[ \text{SIGN}(\text{number}) \]

Returns a number that indicates the sign of \text{number}. The \text{number} is first rounded according to standard Rexx rules, as though the operation \text{number}+0 had been carried out. It returns \(-1\) if \text{number} is less than \(0\), \(0\) if it is \(0\), and \(1\) if it is greater than \(0\).

Here are some examples:

Example 7.71. Built-in function SIGN

| SIGN("12.3") | -> 1 |
| SIGN("-0.307") | -> -1 |
| SIGN(0.0) | -> 0 |

7.4.58. SOURCELINE

\[ \text{SOURCELINE}(n) \]

Returns the line number of the final line in the program if you omit \(n\). If you specify \(n\), returns the \(n\)th line in the program if available at the time of execution. Otherwise, it returns a null string. If specified, \(n\) must be a positive whole number and must not exceed the number that a call to \text{SOURCELINE} with no arguments returns.

If the Rexx program is in translated format, this function returns \(0\) if \(n\) is omitted, and raises an error for all attempts to retrieve a line of the program.

Here are some examples:

Example 7.72. Built-in function SOURCELINE

| SOURCELINE() | -> 10 |
| SOURCELINE(1) | "/* This is a 10-line Rexx program */" |

7.4.59. SPACE
Returns the whitespace-delimited words in \textit{string} with \textit{n} \textit{pad} characters between each word. If you specify \textit{n}, it must be a positive whole number or zero. If it is 0, all whitespace characters are removed. Leading and trailing whitespace characters are always removed. The default for \textit{n} is 1, and the default \textit{pad} character is a blank.

Here are some examples:

\begin{example}

\textbf{Example 7.73. Built-in function \texttt{SPACE}}

\begin{tabular}{ll}
SPACE("abc  def ") & \text{->} \ "abc def" \\
SPACE(" abc def",3) & \text{->} \ "abc def" \\
SPACE("abc  def ",1) & \text{->} \ "abc def" \\
SPACE("abc  def ",0) & \text{->} \ "abcdef" \\
SPACE("abc  def ",2,"+") & \text{->} \ "abc++def"
\end{tabular}

\end{example}

\section*{7.4.60. \texttt{STREAM}}

Returns a string describing the state of, or the result of an operation upon, the character stream \textit{name}. The result may depend on characteristics of the stream that you have specified in other uses of the \texttt{STREAM} function. (To understand the input and output functions, see Chapter 14, \textit{Input and Output Streams}.) This function requests information on the state of an input or output stream or carries out some specific operation on the stream.

The first argument, \textit{name}, specifies the stream to be accessed. The second argument can be one of the following strings that describe the action to be carried out. (Only the capitalized letter is needed; all characters following it are ignored.)

\begin{description}
\item[Command] an operation (specified by the \textit{stream\_command} given as the third argument) is applied to the selected input or output stream. The string that is returned depends on the command performed and can be a null string. The possible input strings for the \textit{stream\_command} argument are described later.
\item[Description] returns any descriptive string associated with the current state of the specified stream. It is identical to the State operation, except that the returned string is followed by a colon and, if available, additional information about the ERROR or NOTREADY states.
\item[State] returns a string that indicates the current state of the specified stream. This is the default operation.
\end{description}

The returned strings are as described in \texttt{Stream} class method \texttt{state}. 

Note

The state (and operation) of an input or output stream is global to a Rexx program; it is not saved and restored across internal function and subroutine calls (including those calls that a CALL ON condition trap causes).

7.4.60.1. Stream Commands

The following stream commands are used to:

• Open a stream for reading, writing, or both.
• Close a stream at the end of an operation.
• Position the read or write position within a persistent stream (for example, a file).
• Get information about a stream (its existence, size, and last edit date).

The stream_command argument must be used when—and only when—you select the operation C (command). The syntax is:

STREAM(name, "Command", stream_command)

In this form, the STREAM function itself returns a string corresponding to the given stream_command if the command is successful. If the command is unsuccessful, STREAM returns an error message string in the same form as the D (Description) operation supplies.

For most error conditions, the additional information is in the form of a numeric return code. This return code is the value of ERRNO that is set whenever one of the file system primitives returns with a -1.

7.4.60.1.1. Command Strings

The argument stream_command can be any expression that the language processor evaluates to a command string that corresponds to the following diagram:
OPEN

opens the named stream. The default for OPEN is to open the stream for both reading and writing data, for example, "OPEN BOTH".

The STREAM function itself returns a description string similar to the one that the D option provides, for example, "READY:" if the named stream is successfully opened, or "ERROR:2" if the named stream is not found.
The following is a description of the options for OPEN:

**READ**
opens the stream for reading only.

**WRITE**
opens the stream for writing only.

**BOTH**
opens the stream for both reading and writing. (This is the default.) Separate read and write pointers are maintained.

**APPEND**
positions the write pointer at the end of the stream. (This is the default.) The write pointer cannot be moved anywhere within the extent of the file as it existed when the file was opened.

**REPLACE**
sets the write pointer to the beginning of the stream and truncates the file. In other words, this option deletes all data that was in the stream when opened.

**SHARED**
Enables another process to work with the stream in a shared mode. This mode must be compatible with the shared mode (SHARED, SHAREREAD, or SHAREWRITE) used by the process that opened the stream.

**SHAREREAD**
Enables another process to read the stream in a shared mode.

**SHAREWRITE**
Enables another process to write the stream in a shared mode.

**NOBUFFER**
turns off buffering of the stream. Thus, all data written to the stream is flushed immediately to the operating system for writing. This option can severely affect output performance. Therefore, use it only when data integrity is a concern, or to force interleaved output to a stream to appear in the exact order in which it was written.

**BINARY**
causes the stream to be opened in binary mode. This means that line-end characters are ignored and treated as data. This is intended to force file operations that are compatible with other Rexx language processors that run on record-based systems, or to process binary data using the line operations.

**Note**

Specifying the BINARY option for a stream that does not exist but is opened for writing also requires the RECLENGTH option to be specified. Omitting the RECLENGTH option in this case raises an error condition.

**RECLENGTH length**
allows the specification of an exact length for each line in a stream. This allows line operations on binary-mode streams to operate on individual fixed-length records. Without this option, line
STREAM

operations on binary-mode files operate on the entire file (for example, as if the RECLength
option were specified with a length equal to that of the file). length must be 1 or greater.

Here are some examples:

Example 7.74. Builtin function STREAM

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream(strout,&quot;c&quot;,&quot;open&quot;)</td>
<td>stream(strout,&quot;c&quot;,&quot;open write&quot;)</td>
</tr>
<tr>
<td>stream(strinp,&quot;c&quot;,&quot;open read&quot;)</td>
<td>stream(strinp,&quot;c&quot;,&quot;open read shared&quot;)</td>
</tr>
</tbody>
</table>

CLOSE

closes the named stream. The STREAM function itself returns READY: if the named stream is
successfully closed, or an appropriate error message. If an attempt is made to close an unopened
file, STREAM returns a null string ("").

Example 7.75. Builtin function STREAM on unopened file

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream(&quot;STRM.TXT&quot;,&quot;c&quot;,&quot;close&quot;)</td>
<td></td>
</tr>
</tbody>
</table>

FLUSH

forces any data currently buffered for writing to be written to this stream.

SEEK offset

sets the read or write position within a persistent stream. If the stream is opened for both reading
and writing and no SEEK option is specified, both the read and write positions are set.

Note

See Chapter 14, Input and Output Streams for a discussion of read and write positions in a
persistent stream.

To use this command, the named stream must first be opened with the OPEN stream command or
implicitly with an input or output operation. One of the following characters can precede the offset
number:

= explicitly specifies the offset from the beginning of the stream. This is the default if no prefix is
supplied. Line Offset=1 means the beginning of stream.

< specifies offset from the end of the stream.

+ specifies offset forward from the current read or write position.

- specifies offset backward from the current read or write position.
The STREAM function itself returns the new position in the stream if the read or write position is successfully located or an appropriate error message otherwise.

The following is a description of the options for SEEK:

**READ**
specifies that the read position is to be set by this command.

**WRITE**
specifies that the write position is to be set by this command.

**CHAR**
specifies that the positioning is to be done in terms of characters. This is the default.

**LINE**
specifies that the positioning is to be done in terms of lines. For non-binary streams, this is an operation that can take a long time to complete, because, in most cases, the file must be scanned from the top to count line-end characters. However, for binary streams with a specified record length, this results in a simple multiplication of the new resulting line number by the record length, and then a simple character positioning. See Section 14.1.5, “Line versus Character Positioning” for a detailed discussion of this issue.

**Note**
If you do line positioning in a file open only for writing, you receive an error message.

**Example 7.76. Builtin function STREAM examples**

```
stream(name,"c","seek =2 read")
stream(name,"c","seek +15 read")
stream(name,"c","seek -7 write line")
fromend = 125
stream(name,"c","seek <"fromend read)
```

**POSITION**
is a synonym for SEEK.

### 7.4.60.1.2. QUERY Stream Commands

Used with these stream commands, the STREAM function returns specific information about a stream. Except for QUERY HANDLE and QUERY SEEK/POSITION, the language processor returns the query information even if the stream is not open. The language processor returns UNKNOWN for QUERY STREAMTYPE and the null string for nonexistent streams.

Note that technically although a directory is persistent, it is not a stream. If the directory exists, the date / time queries return the time stamp of the directory, QUERY SIZE returns 0, and QUERY STREAMTYPE returns UNKNOWN. The other commands return the null string.
**QUERY DATETIME**
returns the date and time stamps of a stream in US format.

Example 7.77. Builtin function STREAM QUERY examples

```
stream("..ile.txt","c","query datetime")
```

A sample output might be:

```
11-12-98 03:29:12
```

**QUERY EXISTS**
returns the full path specification of the named stream, if it exists, or a null string.

Example 7.78. Builtin function STREAM QUERY EXISTS examples

```
stream("..ile.txt","c","query exists")
```

A sample output might be:

```
c:\data\file.txt
```

**QUERY HANDLE**
returns the handle associated with the open stream.

Example 7.79. Builtin function STREAM QUERY HANDLE examples

```
stream("..ile.txt","c","query handle")
```

A sample output might be:

```
3
```

**QUERY POSITION**
returns the current read or write position for the stream, as qualified by the following options:

**READ**
returns the current read position.

**WRITE**
returns the current write position.

**Note**
If the stream is open for both reading and writing, the default is to return the read position. Otherwise, it returns the appropriate position by default.
**CHAR**

returns the position in terms of characters. This is the default.

**LINE**

returns the position in terms of lines. For non-binary streams, this operation can take a long time to complete, because the language processor starts tracking the current line number if not already doing so. Thus, it might require a scan of the stream from the top to count line-end characters. See *Section 14.1.5, “Line versus Character Positioning”* for a detailed discussion of this issue.

```
Example 7.80. Builtin function STREAM QUERY POSITION examples

stream("myfile","c","query position write")
```

A sample output might be:

```
247
```

**SYS**

returns the operating-system stream position in terms of characters.

**QUERY SIZE**

returns the size, in bytes, of a persistent stream.

```
Example 7.81. Builtin function STREAM QUERY SIZE examples

stream("..\file.txt","c","query size")
```

A sample output might be:

```
1305
```

**QUERY STREAMTYPE**

returns a string indicating whether the stream is **PERSISTENT**, **TRANSIENT**, or **UNKNOWN**.

**QUERY TIMESTAMP**

returns the date and time stamps of a stream in an international format. This is the preferred method of getting the date and time because it provides the full 4-digit year.

```
Example 7.82. Builtin function STREAM QUERY TIMESTAMP examples

stream("..\file.txt","c","query timestamp")
```

A sample output might be:

```
2016-11-12 03:29:12
```

### 7.4.61. STRIP
Returns string with leading characters, trailing characters, or both, removed, based on the option you specify. The following are valid options. (Only the capitalized letter is needed; all characters following it are ignored.)

Both
removes both leading and trailing characters from string. This is the default.

Leading
removes leading characters from string.

Trailing
removes trailing characters from string.

The third argument, chars, specifies the set of characters to be removed, and the default is to remove all whitespace characters (spaces and horizontal tabs). If chars is a null string, then no characters are removed. Otherwise, any occurrences of the characters in chars will be removed.

Here are some examples:

**Example 7.83. Builtin function STRIP**

<table>
<thead>
<tr>
<th>STRIP( &quot; ab c &quot; )</th>
<th>-&gt;</th>
<th>&quot;ab c&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRIP( &quot; ab c &quot;, &quot;L&quot; )</td>
<td>-&gt;</td>
<td>&quot;ab c &quot;</td>
</tr>
<tr>
<td>STRIP( &quot; ab c &quot;, &quot;t&quot; )</td>
<td>-&gt;</td>
<td>&quot; ab c&quot;</td>
</tr>
<tr>
<td>STRIP( &quot;12.7000&quot;, , 0 )</td>
<td>-&gt;</td>
<td>&quot;12.7&quot;</td>
</tr>
<tr>
<td>STRIP( &quot;0012.700&quot;, , 0 )</td>
<td>-&gt;</td>
<td>&quot;12.7&quot;</td>
</tr>
<tr>
<td>STRIP( &quot;12.0000&quot;, &quot;T&quot;, &quot;.0&quot; )</td>
<td>-&gt;</td>
<td>&quot;12&quot;</td>
</tr>
</tbody>
</table>

### 7.4.62. SUBSTR (Substring)

Returns the substring of string that begins at the \( n \)th character and is of length length, padded with pad if necessary. \( n \) must be a positive whole number. If \( n \) is greater than \( \text{LENGTH}(\text{string}) \), only pad characters are returned.

See also LEFT and RIGHT builtin-functions.

If you omit length, the rest of the string is returned. The default pad character is a blank.

Here are some examples:

**Example 7.84. Builtin function SUBSTR**

<table>
<thead>
<tr>
<th>SUBSTR(&quot;abc&quot;,2)</th>
<th>-&gt;</th>
<th>&quot;bc&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSTR(&quot;abc&quot;,2,4)</td>
<td>-&gt;</td>
<td>&quot;bc &quot;</td>
</tr>
<tr>
<td>SUBSTR(&quot;abc&quot;,2,6, &quot;.&quot;)</td>
<td>-&gt;</td>
<td>&quot;bc ...&quot;</td>
</tr>
</tbody>
</table>
Note

In some situations the positional (numeric) patterns of parsing templates are more convenient for selecting substrings, especially if more than one substring is to be extracted from a string.

7.4.63. SUBWORD

Returns the substring of \emph{string} that starts at the \emph{n}th word, and is up to \emph{length} whitespace-delimited words. \emph{n} must be a positive whole number. If you omit \emph{length}, it defaults to the number of remaining words in \emph{string}. The returned string never has leading or trailing whitespace, but includes all whitespace characters between the selected words.

Here are some examples:

<table>
<thead>
<tr>
<th>Example 7.85. Builtin function SUBWORD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBWORD(&quot;Now is the time&quot;,2,2) -&gt; &quot;is the&quot;</td>
</tr>
<tr>
<td>SUBWORD(&quot;Now is the time&quot;,3) -&gt; &quot;the time&quot;</td>
</tr>
<tr>
<td>SUBWORD(&quot;Now is the time&quot;,5) -&gt; &quot;&quot;</td>
</tr>
</tbody>
</table>

7.4.64. SYMBOL

Returns the state of the symbol named by \emph{name}. It returns \texttt{BAD} if \emph{name} is not a valid Rexx symbol. It returns \texttt{VAR} if it is the name of a variable, that is, a symbol that has been assigned a value. Otherwise, it returns \texttt{LIT}, indicating that it is either a constant symbol or a symbol that has not yet been assigned a value, that is, a literal.

As with symbols in Rexx expressions, lowercase characters in \emph{name} are translated to uppercase and substitution in a compound name occurs if possible.

Note

You should specify \emph{name} as a literal string, or it should be derived from an expression, to prevent substitution before it is passed to the function.

Here are some examples:

<table>
<thead>
<tr>
<th>Example 7.86. Builtin function SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>/* following: Drop A.3; J=3 */</td>
</tr>
</tbody>
</table>
TIME

7.4.65. TIME

Returns the local time in the 24-hour clock format hh:mm:ss (hours, minutes, and seconds) by default, for example, \texttt{04:41:37}.

You can use the following options to obtain alternative formats, or to gain access to the elapsed-time clock. (Only the capitalized letter is needed; all characters following it are ignored.)

\textbf{Civil}

returns the time in Civil format hh:mmxx. The hours can take the values 1 through 12, and the minutes the values 00 through 59. The minutes are followed immediately by the letters \texttt{am} or \texttt{pm}. This distinguishes times in the morning (12 midnight through 11:59 a.m.--appearing as \texttt{12:00am} through \texttt{11:59am}) from noon and afternoon (12 noon through 11:59 p.m.--appearing as \texttt{12:00pm} through \texttt{11:59pm}). The hour has no leading zero. The minute field shows the current minute (rather than the nearest minute) for consistency with other TIME results.

\textbf{Elapsed}

returns sssssssss.uuuuuu, the number of seconds and microseconds since the elapsed-time clock (described later) was started or reset. The returned number has no leading zeros or whitespace, and the setting of NUMERIC DIGITS does not affect it.

The language processor calculates elapsed time by subtracting the time at which the elapsed-time clock was started or reset from the current time. It is possible to change the system time clock while the system is running. This means that the calculated elapsed time value might not be a true elapsed time. If the time is changed so that the system time is earlier than when the Rexx elapsed-time clock was started (so that the elapsed time would appear negative), the language processor raises an error and disables the elapsed-time clock. To restart the elapsed-time clock, trap the error through SIGNAL ON SYNTAX.

The clock can also be changed by programs on the system. Many LAN-attached programs synchronize the system time clock with the system time clock of the server during startup. This causes the Rexx elapsed time function to be unreliable during LAN initialization.

\textbf{Full}

returns the number of microseconds since 00:00:00.000000 on 1 January 0001, in the format: \texttt{dddddddddddddddd} (no leading zeros or whitespace).
**Notes**

The base date of 1 January 0001 is determined by extending the current Gregorian calendar backward (365 days each year, with an extra day every year that is divisible by 4 except century years that are not divisible by 400). It does not take into account any errors in the calendar system that created the Gregorian calendar originally.

The value returned by Time('F') can be used to calculate the interval between any two times. Note, however, that values returned generally contain more digits than the default NUMERIC DIGITS setting. The NUMERIC DIGITS setting should be increased to a minimum value of 18 when performing timestamp arithmetic.

**Hours**
returns up to two characters giving the number of hours since midnight in the format hh (no leading zeros or whitespace, except for a result of 0).

**Long**
returns time in the format hh:mm:ss.uuuuuu (where uuuuuu are microseconds).

**Minutes**
returns up to four characters giving the number of minutes since midnight in the format mmmm (no leading zeros or whitespace, except for a result of 0).

**Normal**
returns the time in the default format hh:mm:ss. The hours can have the values 00 through 23, and minutes and seconds, 00 through 59. There are always two digits. Any fractions of seconds are ignored (times are never rounded). This is the default.

**Offset**
returns the offset of the local time from UTC in microseconds. The offset value will be negative for timezones west of the Prime Meridian and positive for timezones east of Prime Meridian. The local time('F') value can be converted to UTC by subtracting the time('O') value.

**Reset**
returns sssssssss.uuuuuu, the number of seconds and microseconds since the elapsed-time clock (described later) was started or reset and also resets the elapsed-time clock to zero. The returned number has no leading zeros or whitespace, and the setting of NUMERIC DIGITS does not affect it.

See the Elapsed option for more information on resetting the system time clock.

**Seconds**
returns up to five characters giving the number of seconds since midnight in the format sssss (no leading zeros or whitespace, except for a result of 0).

**Ticks**
returns the number of seconds since 00:00:00.000000 on 1 January 1970, in the format: 

`ddddddddddddd` (no leading zeros or whitespace).
The base date of 1 January 1970 is determined by extending the current Gregorian calendar backward (365 days each year, with an extra day every year that is divisible by 4 except century years that are not divisible by 400). It does not take into account any errors in the calendar system that created the Gregorian calendar originally.

The value returned by Time('T') can be used to calculate the interval between any two times. Note, however, that values returned generally contain more digits than the default NUMERIC DIGITS setting. The NUMERIC DIGITS setting should be increased to a minimum value of 12 when performing timestamp arithmetic.

Time('T') will return a negative number for dates prior to 1 January 1970.

Here are some examples, assuming that the time is 4:54 p.m.:

Example 7.87. Builtin function TIME

<table>
<thead>
<tr>
<th>Function</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME()</td>
<td>&quot;16:54:22&quot;</td>
</tr>
<tr>
<td>TIME(&quot;c&quot;)</td>
<td>&quot;4:54pm&quot;</td>
</tr>
<tr>
<td>TIME(&quot;h&quot;)</td>
<td>&quot;16&quot;</td>
</tr>
<tr>
<td>TIME(&quot;l&quot;)</td>
<td>&quot;16:54:22.120000&quot; /* Perhaps */</td>
</tr>
<tr>
<td>TIME(&quot;m&quot;)</td>
<td>&quot;1014&quot; /* 54 + 60*16 */</td>
</tr>
<tr>
<td>TIME(&quot;n&quot;)</td>
<td>&quot;16:54:22&quot;</td>
</tr>
<tr>
<td>TIME(&quot;s&quot;)</td>
<td>&quot;60862&quot; /* 22 + 60*(54+60*16) */</td>
</tr>
</tbody>
</table>

The elapsed-time clock:

You can use the TIME function to measure real (elapsed) time intervals. On the first call in a program to TIME("E") or TIME("R"), the elapsed-time clock is started, and either call returns 0. From then on, calls to TIME("E") and TIME("R") return the elapsed time since that first call or since the last call to TIME("R").

The clock is saved across internal routine calls, which means that an internal routine inherits the time clock that its caller started. Any timing the caller is doing is not affected, even if an internal routine resets the clock. An example of the elapsed-time clock:

Example 7.88. Builtin function TIME elapsed

<table>
<thead>
<tr>
<th>Function</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>time(&quot;E&quot;)</td>
<td>0 /* The first call */</td>
</tr>
<tr>
<td>/* pause of one second here */</td>
<td></td>
</tr>
<tr>
<td>time(&quot;E&quot;)</td>
<td>1.020000 /* or thereabouts */</td>
</tr>
<tr>
<td>/* pause of one second here */</td>
<td></td>
</tr>
<tr>
<td>time(&quot;R&quot;)</td>
<td>2.030000 /* or thereabouts */</td>
</tr>
<tr>
<td>/* pause of one second here */</td>
<td></td>
</tr>
<tr>
<td>time(&quot;R&quot;)</td>
<td>1.050000 /* or thereabouts */</td>
</tr>
</tbody>
</table>
Note

The elapsed-time clock is synchronized with the other calls to TIME and DATE, so several calls to the elapsed-time clock in a single clause always return the same result. For this reason, the interval between two usual TIME/DATE results can be calculated exactly using the elapsed-time clock.

If you specify string, TIME returns the time corresponding to string in the format option. The string must be supplied in the format option2. The default for option2 is "N". So you need to specify option2 only if string is not in the Normal format. option2 must specify the current time, for example, not "E" or "R". Here are some examples:

Example 7.89. Builtin function TIME formatting

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>time(&quot;C&quot;,&quot;11:27:21&quot;)</td>
<td>11:27am</td>
</tr>
<tr>
<td>time(&quot;N&quot;,&quot;11:27am&quot;,&quot;C&quot;)</td>
<td>11:27:00</td>
</tr>
<tr>
<td>time(&quot;N&quot;, &quot;63326132161828000&quot;, &quot;F&quot;)</td>
<td>08:16:01</td>
</tr>
</tbody>
</table>

You can determine the difference between two times; for example:

Example 7.90. Builtin function TIME difference

```
If TIME("M","5:00pm","C")-TIME("M")<=0
then say "Time to go home"
else say "Keep working"
```

The TIME returned is the earliest time consistent with string. For example, if the result requires components that are not specified in the source format, then those components of the result are zero. If the source has components that the result does not need, then those components of the source are ignored.

When requesting times be converted to Full or Ticks format, a date value of 1 January 0001 is used for the conversion. A time stamp for a time and date combination can be created by combining a value from Date('F') for the time of day.

Example 7.91. Builtin function TIME with F option

```
numeric digits 18  -- needed to add the timestamps
timestamp = date("f", "20170123", "S") + time("f", "08:14:22", "N")
```

Implementation maximum: If the number of seconds in the elapsed time exceeds nine digits (equivalent to over 31.6 years), an error results.

7.4.66. TRACE
TRANSLATE

Returns trace actions currently in effect and, optionally, alters the setting.

If you specify `option`, it selects the trace setting. It must be the valid prefix `?`, one of the alphabetic character options associated with the TRACE instruction (that is, starting with A, C, E, F, I, L, N, O, or R), or both. (See the TRACE instruction in section 2.29.1, "Trace Alphabetic Character (Word) Options" for full details.)

Unlike the TRACE instruction, the TRACE function alters the trace action even if interactive debugging is active. Also unlike the TRACE instruction, `option` cannot be a number.

Here are some examples:

Example 7.92. Builtin function TRACE

<table>
<thead>
<tr>
<th>TRACE()</th>
<th>-&gt;</th>
<th>&quot;?R&quot; /* maybe */</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACE(&quot;O&quot;)</td>
<td>-&gt;</td>
<td>&quot;?R&quot; /* also sets tracing off */</td>
</tr>
<tr>
<td>TRACE(&quot;?I&quot;)</td>
<td>-&gt;</td>
<td>&quot;0&quot; /* now in interactive debugging */</td>
</tr>
</tbody>
</table>

TRANSLATE

Returns `string` with each character translated to another character or unchanged. You can also use this function to reorder the characters in `string`.

The output table is `tableo` and the input translation table is `tablei`. TRANSLATE searches `tablei` for each character in `string`. If the character is found, the corresponding character in `tableo` is used in the result string; if there are duplicates in `tablei`, the first (leftmost) occurrence is used. If the character is not found, the original character in `string` is used. The result string is always the same length as `string`.

The tables can be of any length. If you specify neither table and omit `pad`, `string` is simply translated to uppercase (that is, lowercase a-z to uppercase A-Z), but, if you include `pad`, the language processor translates the entire string to `pad` characters. `tablei` defaults to `XRANGE("00"x,"FF"x)`, and `tableo` defaults to the null string and is padded with `pad` or truncated as necessary. The default `pad` is a blank.

`pos` is the position of the first character of the translated range. The default starting position is 1. `length` is the range of characters to be translated. If `length` is omitted, the remainder of the string from the starting position to the end is used.

Here are some examples:

Example 7.93. Builtin function TRANSLATE

| TRANSLATE("abcdef")     | -> | "ABCDEF"  |
| TRANSATE("abcdef", , , 2, 3) | -> | "aBCDef"  |
The last example shows how to use the TRANSLATE function to reorder the characters in a string. The last character of any four-character string specified as the second argument is moved to the beginning of the string.

### 7.4.68. TRUNC (Truncate)

TRUNC(number, n)

Returns the integer part of `number` and `n` decimal places. The default `n` is 0 and returns an integer with no decimal point. If you specify `n`, it must be a positive whole number or zero. The `number` is rounded according to standard Rexx rules, as though the operation `number+0` had been carried out. Then it is truncated to `n` decimal places or trailing zeros are added to reach the specified length. The result is never in exponential form. If there are no nonzero digits in the result, any minus sign is removed.

Here are some examples:

**Example 7.94. Builtin function TRUNC**

| `TRUNC(12.3)` | -> 12 |
| `TRUNC(127.09782, 3)` | -> 127.097 |
| `TRUNC(127.1, 3)` | -> 127.100 |
| `TRUNC(127, 2)` | -> 127.00 |

### Note

The `number` is rounded according to the current setting of NUMERIC DIGITS, if necessary, before the function processes it.

### 7.4.69. UPPER

UPPER(string)

Returns a new string with the characters of `string` beginning with character `n` for `length` characters converted to uppercase. If `n` is specified, it must be a positive whole number. If `n` is not specified, the
case conversion will start with the first character. If length is specified, it must be a non-negative whole number. If length is not specified, the default is to convert the remainder of the string.

**Example 7.95. Builtin function UPPER**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper(&quot;Albert Einstein&quot;)</td>
<td>&quot;ALBERT EINSTEIN&quot;</td>
</tr>
<tr>
<td>upper(&quot;abcdef&quot;, 4)</td>
<td>&quot;abcDEF&quot;</td>
</tr>
<tr>
<td>upper(&quot;abcdef&quot;, 3, 2)</td>
<td>&quot;abCDef&quot;</td>
</tr>
</tbody>
</table>

### 7.4.70. USERID

The return value is the active user identification.

### 7.4.71. VALUE

Returns the value of the symbol that name (often constructed dynamically) represents and optionally assigns a new value to it. By default, VALUE refers to the current Rexx-variables environment, but other, external collections of variables can be selected. If you use the function to refer to Rexx variables, name must be a valid Rexx symbol. (You can confirm this by using the SYMBOL function.) Lowercase characters in name are translated to uppercase for the local environment. For the global environment lowercase characters are not translated because the global environment supports mixed-case identifiers. Substitution in a compound name (see Section 1.13.5, “Compound Symbols”) occurs if possible.

If you specify newvalue, the named variable is assigned this new value. This does not affect the result returned; that is, the function returns the value of name as it was before the new assignment.

Here are some examples:

**Example 7.96. Builtin function VALUE**

```plaintext
/* After: Drop A3; A3=7; K=3; fred="K"; list.5="Hi" */
VALUE("a\"k\") -> "A3" /* looks up A3 */
VALUE("a\"k\"\"k\") -> "7"
VALUE("fred") -> "K" /* looks up FRED */
VALUE(fred) -> "3" /* looks up K */
VALUE(fred,5) -> "3" /* looks up K and */
/* then sets K=5 */
VALUE(fred) -> "5" /* looks up K */
VALUE("LIST."\"k\") -> "Hi" /* looks up LIST.5 */
```

**Notes:**

1. If the VALUE function refers to an uninitialized Rexx variable, the default value of the variable is always returned. The NOVALUE condition is not raised.

2. The VALUE function is used when a variable contains the name of another variable, or when a name is constructed dynamically. If you specify name as a single literal string and omit
newvalue and selector, the symbol is a constant and the string between the quotation marks can usually replace the whole function call. For example, Fred=VALUE("k"); is identical with the assignment Fred=k; unless the NOVALUE condition is trapped. See Chapter 11, Conditions and Condition Traps.

### 7.4.71.1. Operating System environment variables

To use VALUE to manipulate Windows or Unix-like system environment variables, selector must be \texttt{ENVIRONMENT}. In this case, the variable name need not be a valid Rexx symbol. On Unix-like systems environment variable names are case-sensitive.

VALUE returns the null string for undefined environment variables. You cannot determine whether an environment variable is undefined or has been set to the null string.

Environment variables set by VALUE are not kept after program termination.

**Restriction**

The values assigned to the variables must not contain any character that is a hexadecimal zero ("00"X). For example:

```plaintext
Call VALUE "MYVAR", "FIRST" || "00"X || "SECOND", "ENVIRONMENT"
```

sets MYVAR to "FIRST", truncating "00"x and "SECOND".

Here are some more examples:

**Example 7.97. Builtin function VALUE**

```plaintext
/* Given that an external variable FRED has a value of 4 */
share = "ENVIRONMENT"
say VALUE("fred",7,share) /* says "4" and assigns */
/* FRED a new value of 7 */
say VALUE("fred", ,share) /* says "7" */

/* Accessing and changing Windows environment entries */
env = "ENVIRONMENT"
new = "C:\EDIT\DOCS;"
say value("PATH",new,env) /* says "C:\WINDOWS" (perhaps) */
/* and sets PATH = "C:\EDIT\DOCS;" */
say value("PATH", ,env) /* says "C:\EDIT\DOCS;" */
```

To delete an environment variable use .\texttt{nil} as the newvalue. To delete the environment variable "MYVAR" specify: value("MYVAR", .\texttt{NIL}, "ENVIRONMENT"). If you specify an empty string as the newvalue like in value("MYVAR", "", "ENVIRONMENT") the value of the external environment variable is set to an empty string which on Windows and Unix-like systems is not the same as deleting the environment variable.
Note

Any changes to Windows or Unix-like system environment variables are not kept after program termination.

### 7.4.71.2. Global environment variables

You can use the `VALUE` function to return a value from the Rexx global environment directory. To do so, omit `newvalue` and specify `selector` as the null string. The language processor sends the message `name` (without arguments) to the current environment object. The environment returns the object identified by `name`. If there is no such object, it returns, by default, the string `name` with an added initial period (an environment symbol - see Section 1.13.6, “Environment Symbols”).

Here are some examples:

#### Example 7.98. Builtin function VALUE

```
-- assume the environment name MYNAME identifies the string "Simon"
name = value("MYNAME", , "")       -- sends MYNAME message to the environment
say "Hello," name                  -- "Hello, Simon"

-- Assume the environment name NONAME does not exist
name = value("NONAME", , "")       -- sends NONAME message to the environment
say "Hello," name                  -- "Hello, .NONAME"
```

You can use the `VALUE` function to change a value in the global environment directory. Include a `newvalue` and specify `selector` as the null string. The language processor sends the message `name` (with `=` appended) and the single argument `newvalue` to the environment object. After receiving this message, the environment identifies the object `newvalue` by the name `name`.

Here is an example:

#### Example 7.99. Builtin function VALUE

```
call value "MYNAME", "David", ""   -- sends MYNAME=('David') message to the environment
say "Hello," value("MYNAME", , "") -- "Hello, David"
```

### 7.4.72. VAR

**VAR**(name)

Returns 1 if `name` is the name of a variable (that is, a symbol that has been assigned a value), or 0.

Here are some examples:

#### Example 7.100. Builtin function VAR

```
/* Following: DROP A.3; J=3 */
VAR("J") -> 1
VAR(3)   -> 0  /* has tested "3" */
```
7.4.73. VERIFY

Returns a number that, by default, indicates whether `string` is composed only of characters from `reference`. It returns 0 if all characters in `string` are in `reference`, or returns the position of the first character in `string` that is not in `reference`.

The `option` can be either `Nomatch` (the default) or `Match`. (Only the capitalized and highlighted letter is needed. All characters following it are ignored, and it can be in uppercase or lowercase characters.) If you specify `Match`, the function returns the position of the first character in the `string` that is in `reference`, or returns 0 if none of the characters are found.

The default for `start` is 1; thus, the search starts at the first character of `string`. You can override this by specifying a different `start` point, which must be a positive whole number.

The default for `length` is the length of the string from `start` to the end of the string. Thus, the search proceeds to the end of the receiving string. You can override this by specifying a different `length`, which must be a non-negative whole number.

If `string` is null, the function returns 0, regardless of the value of the third argument. Similarly, if `start` is greater than `LENGTH(string)`, the function returns 0. If `reference` is null, the function returns 0 if you specify `Match`; otherwise, the function returns the `start` value.

Here are some examples:

**Example 7.101. Built-in function VERIFY**

<table>
<thead>
<tr>
<th><code>VERIFY</code></th>
<th><code>-&gt;</code></th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERIFY(&quot;123&quot;, &quot;1234567890&quot;)</td>
<td>-&gt;</td>
<td>0</td>
</tr>
<tr>
<td>VERIFY(&quot;123&quot;, &quot;1234567890&quot;)</td>
<td>-&gt;</td>
<td>2</td>
</tr>
<tr>
<td>VERIFY(&quot;AB4T&quot;, &quot;1234567890&quot;)</td>
<td>-&gt;</td>
<td>1</td>
</tr>
<tr>
<td>VERIFY(&quot;AB4T&quot;, &quot;1234567890&quot;, &quot;M&quot;)</td>
<td>-&gt;</td>
<td>3</td>
</tr>
<tr>
<td>VERIFY(&quot;AB4T&quot;, &quot;1234567890&quot;, &quot;N&quot;)</td>
<td>-&gt;</td>
<td>1</td>
</tr>
<tr>
<td>VERIFY(&quot;1P04&quot;, &quot;1234567890&quot;, ,3)</td>
<td>-&gt;</td>
<td>4</td>
</tr>
<tr>
<td>VERIFY(&quot;123&quot;, &quot;&quot;,N,2)</td>
<td>-&gt;</td>
<td>2</td>
</tr>
<tr>
<td>VERIFY(&quot;ABCDEFGHIJKLMNOPQRSTUVWXYZ&quot;, &quot;&quot;, ,3)</td>
<td>-&gt;</td>
<td>3</td>
</tr>
<tr>
<td>VERIFY(&quot;AB3CD5&quot;, &quot;1234567890&quot;, &quot;M&quot;,4)</td>
<td>-&gt;</td>
<td>6</td>
</tr>
<tr>
<td>VERIFY(&quot;ABCDEF&quot;, &quot;ABC&quot;, &quot;N&quot;,2,3)</td>
<td>-&gt;</td>
<td>4</td>
</tr>
<tr>
<td>VERIFY(&quot;ABCDEF&quot;, &quot;ADEF&quot;, &quot;M&quot;,2,3)</td>
<td>-&gt;</td>
<td>4</td>
</tr>
</tbody>
</table>

7.4.74. WORD

...
Returns the \( n \)th whitespace-delimited word in \( string \) or returns the null string if less than \( n \) words are in \( string \). \( n \) must be a positive whole number. This function is equivalent to \( \text{SUBWORD}(string, \ n, \ 1) \).

Here are some examples:

**Example 7.102. Builtin function WORD**

\[
\begin{align*}
\text{WORD}("Now is the time",3) & \rightarrow \ "the" \\
\text{WORD}("Now is the time",6) & \rightarrow \ "" \\
\end{align*}
\]

### 7.4.75. WORDINDEX

Returns the position of the first character in the \( n \)th whitespace-delimited word in \( string \) or returns 0 if less than \( n \) words are in \( string \). \( n \) must be a positive whole number.

Here are some examples:

**Example 7.103. Builtin function WORDINDEX**

\[
\begin{align*}
\text{WORDINDEX}("Now is the time",3) & \rightarrow \ 8 \\
\text{WORDINDEX}("Now is the time",6) & \rightarrow \ 0 \\
\end{align*}
\]

### 7.4.76. WORDLENGTH

Returns the length of the \( n \)th whitespace-delimited word in the \( string \) or returns 0 if less than \( n \) words are in the \( string \). \( n \) must be a positive whole number.

Here are some examples:

**Example 7.104. Builtin function WORDLENGTH**

\[
\begin{align*}
\text{WORDLENGTH}("Now is the time",2) & \rightarrow \ 2 \\
\text{WORDLENGTH}("Now comes the time",2) & \rightarrow \ 5 \\
\text{WORDLENGTH}("Now is the time",6) & \rightarrow \ 0 \\
\end{align*}
\]

### 7.4.77. WORDPOS (Word Position)

Returns the word number of the first word of \( phrase \) found in \( string \) or returns 0 if \( phrase \) contains no words or if \( phrase \) is not found. Several whitespace characters between words in either \( phrase \) or \( string \) are treated as a single blank for the comparison, but otherwise the words must match exactly.

By default, the search starts at the first word in \( string \). You can override this by specifying \( start \) (which must be positive), the word at which to start the search.
Here are some examples:

### Example 7.105. Builtin function WORDPOS

<table>
<thead>
<tr>
<th><code>WORDPOS</code></th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;the&quot;,&quot;now is the time&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;The&quot;,&quot;now is the time&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;is the&quot;,&quot;now is the time&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;is the&quot;,&quot;now is the time&quot;</td>
<td>2</td>
</tr>
</tbody>
</table>
| "is time 
"","now is the time" | 0 |
| "be","To be or not to be" | 2 |
| "be","To be or not to be",3 | 6 |

### 7.4.78. WORDS

![WORDS](string)

Returns the number of whitespace-delimited words in `string`.

Here are some examples:

### Example 7.106. Builtin function WORDS

<table>
<thead>
<tr>
<th><code>WORDS</code></th>
<th><code>string</code></th>
<th><code>Result</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Now is the time&quot;</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td>&quot; &quot;</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

### 7.4.79. X2B (Hexadecimal to Binary)

![X2B](hexstring)

Returns a string, in character format, that represents `hexstring` converted to binary. The `hexstring` is a string of hexadecimal characters. It can be of any length. Each hexadecimal character is converted to a string of 4 binary digits. You can optionally include whitespace characters in `hexstring` (at byte boundaries only, not leading or trailing) to improve readability; they are ignored.

The returned string has a length that is a multiple of 4, and does not include any whitespace.

If `hexstring` is null, the function returns a null string.

Here are some examples:

### Example 7.107. Builtin function X2B

<table>
<thead>
<tr>
<th><code>X2B</code></th>
<th><code>hexstring</code></th>
<th><code>Result</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;C3&quot;</td>
<td>-</td>
<td>&quot;11000011&quot;</td>
</tr>
<tr>
<td>&quot;7&quot;</td>
<td>-</td>
<td>&quot;0111&quot;</td>
</tr>
<tr>
<td>&quot;1 C1&quot;</td>
<td>-</td>
<td>&quot;0001111000001&quot;</td>
</tr>
</tbody>
</table>

You can combine X2B with the functions D2X and C2X to convert numbers or character strings into binary form.

Here are some examples:
Example 7.108. Builtin function X2B

X2B(C2X("C3"x))  ->  "11000011"
X2B(D2X("129"))  ->  "10000001"
X2B(D2X("12"))   ->  "1100"

7.4.80. X2C (Hexadecimal to Character)

\[ X2C(\text{hexstring}) \]

Returns a string, in character format, that represents \text{hexstring} converted to character. The returned string has half as many bytes as the original \text{hexstring}. \text{hexstring} can be of any length. If necessary, it is padded with a leading zero to make an even number of hexadecimal digits.

You can optionally include whitespace characters in \text{hexstring} (at byte boundaries only, not leading or trailing) to improve readability; they are ignored.

If \text{hexstring} is null, the function returns a null string.

Here are some examples:

Example 7.109. Builtin function X2C

\[
\begin{align*}
X2C("4865 6c6c 6f") & \rightarrow \ "Hello" \quad /\quad \text{ASCII} \quad / \\
X2C("3732 73") & \rightarrow \ "72s" \quad /\quad \text{ASCII} \quad /
\end{align*}
\]

7.4.81. X2D (Hexadecimal to Decimal)

\[ X2D(\text{hexstring}, n) \]

Returns the decimal representation of \text{hexstring}. The \text{hexstring} is a string of hexadecimal characters. If the result cannot be expressed as a whole number, an error occurs. That is, the result must not have more digits than the current setting of NUMERIC DIGITS.

You can optionally include whitespace characters in \text{hexstring} (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

If \text{hexstring} is null, the function returns \textbf{0}.

If you do not specify \textit{n}, the \text{hexstring} is processed as an unsigned binary number.

Here are some examples:

Example 7.110. Builtin function X2D

\[
\begin{align*}
X2D("0E") & \rightarrow \ 14 \\
X2D("81") & \rightarrow \ 129 \\
X2D("F81") & \rightarrow \ 3969 \\
X2D("FF81") & \rightarrow \ 65409 \\
X2D("46 39"x) & \rightarrow \ 240 \quad /\quad \text{ASCII} \quad /
\end{align*}
\]
If you specify \( n \), the string is taken as a signed number expressed in \( n \) hexadecimal digits. If the leftmost bit is off, then the number is positive; otherwise, it is a negative number. In both cases it is converted to a whole number, which can be negative. If \( n \) is 0, the function returns 0.

If necessary, \( \text{hexstring} \) is padded on the left with 0 characters (not "sign-extended"), or truncated on the left to \( n \) characters.

Here are some examples:

Example 7.111. Built-in function \( \text{X2D} \)

\[
\begin{align*}
\text{X2D}(&"81",2) & \to -127 \\
\text{X2D}(&"81",4) & \to 129 \\
\text{X2D}(&"F081",4) & \to -3967 \\
\text{X2D}(&"F081",3) & \to 129 \\
\text{X2D}(&"F081",2) & \to -127 \\
\text{X2D}(&"F081",1) & \to 1 \\
\text{X2D}(&"0031",0) & \to 0 \\
\end{align*}
\]

7.4.82. **XRANGE (Hexadecimal Range)**

\[
\text{XRANGE} \left( \text{start}, \text{end} \right)
\]

Returns a string of all valid 1-byte encodings (in ascending order) between and including the values \( \text{start} \) and \( \text{end} \), or a character sequence representing a POSIX character class name. Any number of
start / end bytes and character class names can be specified, and all resulting sequences are returned as a concatenated string.

The default value for start is '00'x, and the default value for end is 'ff'x. If start is greater than end, the values wrap from 'ff'x to '00'x. If specified, start and end must be single characters.

The characters in a sequence representing a POSIX character class name are returned in ascending order. The following POSIX character class names can be specified:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Returned sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALNUM</td>
<td>Alphanumeric characters</td>
<td>0123456789ABCDEFHJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>ALPHA</td>
<td>Alphabetic characters</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>BLANK</td>
<td>Space and tab</td>
<td>'09 20'x</td>
</tr>
<tr>
<td>CNTRL</td>
<td>Control characters</td>
<td>'00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 7F'x</td>
</tr>
<tr>
<td>DIGIT</td>
<td>Digits</td>
<td>0123456789</td>
</tr>
<tr>
<td>GRAPH</td>
<td>Visible characters</td>
<td>!&quot;#$%&amp;'()*+,-./0123456789;:&lt;=&gt;? @ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^_`abcdefghijklmnopqrstuvwxyz{</td>
</tr>
<tr>
<td>LOWER</td>
<td>Lowercase letters</td>
<td>abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>PRINT</td>
<td>Visible characters and space character</td>
<td>!&quot;#$%&amp;'()*+,-./0123456789;:&lt;=&gt;? @ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^_`abcdefghijklmnopqrstuvwxyz{</td>
</tr>
<tr>
<td>PUNCT</td>
<td>Punctuation characters</td>
<td>!&quot;#$%&amp;'()*+,-;.&lt;=&gt;?@[^^_`{</td>
</tr>
<tr>
<td>SPACE</td>
<td>Whitespace characters</td>
<td>'09 0A 0B 0C 0D 20'x</td>
</tr>
<tr>
<td>UPPER</td>
<td>Uppercase letters</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>XDIGIT</td>
<td>Hexadecimal digits</td>
<td>0123456789ABCDEFabcdef</td>
</tr>
</tbody>
</table>

Here are some examples:

Example 7.112. Builtin function XRANGE

```
XRANGE("a", "f")  --> "abcdef"
XRANGE('03'x, '07'x)  --> '0304050607'x
XRANGE( '04'x)  --> '0001020304'x
XRANGE('FE'x, '02'x)  --> 'FEFF000102'x
XRANGE('t', 't', "h", "i", "r", "t")  --> "thirst"
XRANGE('xdigit')  --> "0123456789ABCDEFabcdef"
XRANGE("a", "f", "A", "F", "digit")  --> "abcdefABCDEF0123456789"
```
Chapter 8.

Rexx Utilities (RexxUtil)

RexxUtil is a function package for Windows and Unix-like systems. It contains functions to:

- Manipulate operating system files and directories
- Manipulate Windows classes and objects
- Perform text screen input and output

All of the RexxUtil functions are registered by the Rexx interpreter on startup so there is no need to manually register the functions.

8.1. A Note on Error Codes

Some of the RexxUtil functions return operating system error codes on failure. The `SysGetErrortext` function can be used to retrieve the description of a system error code.

On Windows, the meaning of these error return codes can be looked up in the Windows System Error Codes documentation provided by Microsoft, currently at [https://docs.microsoft.com/en-us/windows/win32/debug/system-error-codes](https://docs.microsoft.com/en-us/windows/win32/debug/system-error-codes).

8.2. List of Rexx Utility Functions

The following table lists all of the RexxUtil functions and the platforms on which they are available.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Exists on Platform</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Windows</td>
<td>Unix-like</td>
</tr>
<tr>
<td><code>RxMessageBox</code> (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>RxWinExec</code> (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>SysAddRexxMacro</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>SysBootDrive</code> (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>SysClearRexxMacroSpace</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>SysCls</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>SysCreatePipe</code> (Unix-like systems only)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><code>SysCurPos</code> (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>SysCurState</code> (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>SysDriveInfo</code> (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>SysDriveMap</code> (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>SysDropRexxMacro</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>SysDumpVariables</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>SysFileCopy</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>SysFileDelete</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>SysFileExists</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>SysFileMove</code></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Function Name</td>
<td>Exists on Platform</td>
<td>Remarks</td>
</tr>
<tr>
<td>---------------------------------------------------------</td>
<td>--------------------</td>
<td>---------------</td>
</tr>
<tr>
<td></td>
<td>Windows</td>
<td>Unix-like</td>
</tr>
<tr>
<td>SysFileSearch</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysFileSystemType (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SysFileTree</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysFork (Unix-like systems only)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SysFormatMessage</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysFromFileUnicode (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SysGetErrorText</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysGetFileDateTime</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysGetKey</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysGetLongPathName (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SysGetMessage (Unix-like systems only)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SysGetMessageX (Unix-like systems only)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SysGetShortPathName (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SysIni (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SysIsFile</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysIsFileCompressed (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SysIsFileDirectory</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysIsFileEncrypted (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SysIsFileLink</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysIsFileNotContentIndexed (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SysIsFileOffline (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SysIsFileSparse (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SysIsFileTemporary (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SysLinVer (Linux Only)</td>
<td>No</td>
<td>No*</td>
</tr>
<tr>
<td>SysLoadRexxMacroSpace</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysMkDir</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysQueryProcess</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysQueryRexxMacro</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysReorderRexxMacro</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysRmDir</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysSaveRexxMacroSpace</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysSearchPath</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysSetFileDateTime</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysSetPriority</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysShutdownSystem (Windows only)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SysSleep</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SysStemCopy</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### 8.3. RxMessageBox (Windows only)

![Diagram of RxMessageBox](image)

Displays a modal dialog box with a title, a message text, an optional icon, and a set of buttons. The dialog box is displayed using the Windows `MessageBox()` function.

RxMessageBox returns one of the following integer values indicating which button the user clicked.

1. The **OK** button was pressed

2. The **Cancel** button was pressed

3. The **Abort** button was pressed
The **Retry** button was pressed

The **Ignore** button was pressed

The **Yes** button was pressed

The **No** button was pressed

The **Try Again** button was pressed

The **Continue** button was pressed

If a message box has a **Cancel** button, the function returns the value 2 if either the ESC key is pressed or the **Cancel** button is selected. If the message box has no **Cancel** button, pressing ESC will have no effect unless an **OK** button is present. If an **OK** is displayed and the user presses ESC, the return value will be 1.

If the Windows MessageBox() function fails, the return value is zero.

**text**

The message box text.

**title**

The message box title. The default title is **Error**.

**button**

The message box push button style.

**OK**

The message box contains one push button: **OK**. This is the default.

**OKCANCEL**

The message box contains two push buttons: **OK** and **Cancel**.

**RETRYCANCEL**

The message box contains two push buttons: **Retry** and **Cancel**.

**ABORTRETRYIGNORE**

The message box contains three push buttons: **Abort**, **Retry**, and **Ignore**.

**CANCELTRYCONTINUE**

The message box contains three push buttons: **Cancel**, **Try Again**, and **Continue**. Use this message box button style instead of **ABORTRETRYIGNORE**.

**YESNO**

The message box contains two push buttons: **Yes** and **No**.

**YESNOCANCEL**

The message box contains three push buttons: **Yes**, **No**, and **Cancel**.
**icon**
The message box icon. The default is **NONE**. Allowed icons are:

- **NONE**
  No icon is displayed.

- **INFORMATION, ASTERISK**
  A stop-sign icon appears in the message box.

- **WARNING, EXCLAMATION**
  An exclamation-point icon appears in the message box.

- **ERROR, HAND, STOP**
  An icon consisting of a lowercase letter i in a circle appears in the message box.

- **QUESTION, QUERY**
  A question-mark icon appears in the message box. This icon is no longer recommended.

**option**
More than one of the following options may be specified. If neither **DEFBUTTON2, DEFBUTTON3, DEFBUTTON4** is specified, the first button is the default button. Allowed options are:

- **DEFBUTTON2**
  The second button is the default button.

- **DEFBUTTON3**
  The third button is the default button.

- **DEFBUTTON4**
  The fourth button is the default button.

- **RIGHT**
  The message box text is right-justified.

- **TOPMOST**
  The message box is placed above all non-topmost windows and stays above them, even when the window is deactivated.

---

**Example 8.1. RexxUtil — RxMessageBox**

```rexx
-- Give option to quit
if RxMessageBox("Shall we continue?", ",", "YesNo", "Question") = 7
   then exit -- "No" button clicked, exit
```

### 8.4. RxWinExec (Windows only)

Starts (executes) the application as specified in **cmdline**.

**Parameters:**
RxWinExec (Windows only)

**cmdline**
A string containing a file name and optional parameters for the application to be executed. If the name of the executable file in *cmdline* does not contain a directory path, RxWinExec searches for the executable file in this sequence:

1. The directory from which ooRexx was loaded.
2. The current directory.
3. The Windows system directory.
4. The Windows directory.
5. The directories listed in the PATH environment variable.

**cmdshow**
Specifies how a Windows-based application window is to be shown. For a non-Windows-based application, the PIF file, if any, for the application determines the window state.

- **SHOWNORMAL**
  Activates and displays a window.
- **SHOWNOACTIVATE**
  Displays the window while the current active window remains active.
- **SHOWMINNOACTIVE**
  Displays the window as a minimized window, the current active window remains active.
- **SHOWMINIMIZED**
  Activates the window and displays it as a minimized window.
- **SHOWMAXIMIZED**
  Activates the window and displays it as a maximized window.
- **HIDE**
  Hides the window and activates another window.
- **MINIMIZE**
  Minimizes the specified window and activates the next top-level window in the Z order.

**Return codes:**
If the application is started successfully, the process id (PID) is returned. If an error occurs the returned value is less than 32.

Error return codes correspond to a *Windows System Error code*. If the Windows system error code is greater than 32 it is negated. This is to prevent confusion between a legitimate error code and a PID.

Some common error returns for this function are as follows.

2. The specified file was not found.
3
The specified path was not found.

11
The EXE file is invalid.

-53
The network path is invalid.

8.5. SysAddRexxMacro

Adds a routine to the Rexx macrospace. SysAddRexxMacro returns the RexxAddMacro return code.

Parameters:

name
The name of the function added to the macrospace.

file
The file containing the Rexx program.

order
The macrospace search order. The order can be "B" (Before), which is default, or "A" (After).

8.6. SysBootDrive (Windows only)

Returns the drive used to boot Windows, for example, "C:".

8.7. SysClearRexxMacroSpace

Clears the Rexx macrospace. SysClearRexxMacroSpace returns the RexxClearMacroSpace return code.

8.8. SysCls

Clears the screen.

Example 8.2. RexxUtil — SysCls

call SysCls
8.9. SysCreatePipe (Unix-like systems only)

SysCreatePipe()

Creates an unnamed pipe.

Returns:

Returns a string like "handle handle" where the first handle is for read and the second handle for write.

8.10. SysCurPos (Windows only)

SysCurPos(row, column)

Returns the cursor position in the form row col and optionally moves the cursor to a new location.

Parameters:

row
The row to move to.

col
The column to move to.

Note

Position (0,0) is the upper left corner.

You can call SysCurPos without a column and row position to obtain the cursor position without moving the cursor.

Example 8.3. RexxUtil — SysCurPos

say SysCurPos() -- (e. g.) 10 0

8.11. SysCurState (Windows only)

SysCurState(state)

Hides or displays the cursor.

Parameter:

state
The new cursor state. Allowed states are:
"ON"
  Display the cursor

"OFF"
  Hide the cursor

8.12. SysDriveInfo (Windows only)

```plaintext
SysDriveInfo(drive)
```

Returns drive information in the form `drive free total label`, where `drive` is the drive identifier, `free` is the total number of unused bytes, `total` is the total size in bytes, and `label` is the volume label of the specified `drive`. Returns the null string if the drive is not accessible.

If `drive` is omitted, the current drive is used.

Parameter:

`drive`
  A drive specification like `d:\` or `\share\path`.

Example 8.4. RexxUtil — SysDriveInfo

```
say SysDriveInfo() -- e. g. C: 23989809152 254721126400 Windows
```

8.13. SysDriveMap (Windows only)

```plaintext
SysDriveInfo(drive, opt)
```

Returns a string listing accessible drives (separated by blanks) in the form: `C: D: ....`

Parameters:

`drive`
  The first drive letter of the drive map. The default is "C:"

`opt`
  The drivemap option. This can be:
  "USED" returns the drives that are accessible or in use, including all local and remote drives. This is the default.
  "FREE" returns drives that are free or not in use.
  "LOCAL" returns only local drives.
"REMOTE"
  returns only remote drives, such as redirected LAN resources or installable file system (IFS) attached drives.

"REMOVABLE"
  returns removable drives.

"CDROM"
  returns CD-ROM drives.

"RAMDISK"
  returns drives assigned from RAM.

Example 8.5. RexxUtil — SysDriveMap

```plaintext
say SysDriveMap("C:", "USED") -- C: M: N: O: P: R: Z:
```

8.14. SysDropRexxMacro

![SysDropRexxMacro](image)

Removes a routine from the Rexx macrospace. SysDropRexxMacro returns the RexxDropMacro return code.

Parameter:

`name`
  The name of the function removed from the macrospace.

8.15. SysDumpVariables

![SysDumpVariables](image)

Dumps all variables in the current scope either to the specified file `filename` (new data is appended) or to STDOUT if you omit `filename`. The format of the data is, with one variable per line:

```
Name=MYVAR, Value='This is the content of MYVAR'
```

Parameter:

`filename`
  The name of the file to which variables are appended. The dump is written to STDOUT if you omit this parameter.

Return codes:

0
  Dump completed successfully.
-1
Dump failed.

See also **RexxContext** method **variables**.

---

**Example 8.6. RexxUtil — SysDumpVariables**

```plaintext
Call SysDumpVariables "MyVars.Lst" /* append vars to file */
Call SysDumpVariables              /* list vars on STDOUT */
```

### 8.16. SysFileCopy

> SysFileCopy(source, target)

Copies a file from one location to another. Wildcard file specifications are not allowed.

**Parameter:**

**source**
- The path/name of the file to be copied. The source and the target path does not support wildcard characters. On Unix-like systems, if the first character is a ~ (tilde) character optionally followed by a username, it is replaced by the current or specified user's home directory.

**target**
- The path/name of the target location where the file is to be copied.

**Return codes:**

0
- File copied successfully.

Other
- A *Windows System Error code*.

---

**Example 8.7. RexxUtil — SysFileCopy**

```plaintext
call SysFileCopy "c:\temp\myfile.txt", "d:\myfolder\myCopy.txt"
call SysFileCopy "/tmp/myfile.txt", "~\myCopy.txt"
```

### 8.17. SysFileDelete

> SysFileDelete(file)

Deletes a file.

**Parameter:**
The name of the file to be deleted. The name does not support wildcard characters. On Unix-like systems, if the first character is a ~ (tilde) character optionally followed by a username, it is replaced by the current or specified user's home directory.

Return codes:

0
File deleted successfully.

Other
An operating system error code. On Windows, this may be one of the following, but could be others.

2
File not found.

3
Path not found.

5
Access denied or busy.

26
Not DOS disk.

32
Sharing violation.

36
Sharing buffer exceeded.

87
Does not exist.

206
File name exceeds range error.

Example 8.8. RexxUtil — SysFileDelete

| parse arg InputFile OutputFile |
| call SysFileDelete OutputFile -- unconditionally erase output file |

8.18. SysFileExists

Checks for the existence of a file. Returns 1 if any file system entity with the given name exists. In particular, this will return 1 for both regular files and directories.

Parameters:
filename
The name of the file to check for the existence of. The name does not support wildcard characters. On Unix-like systems, if the first character is a ~ (tilde) character optionally followed by a username, it is replaced by the current or specified user's home directory.

Returns:

0
The file does not exist.

1
The file exists.

Example 8.9. RexxUtil — SysFileExists

```rexx
if SysFileExists(InputFile) then say "File Exists!"
else say "File does not exist."
```

8.19. SysFileMove

```
SysFileMove(source, target)
```

Moves a file from one location to another. Wildcard file specifications are not allowed.

Parameter:

source
The path/name of the file to be moved. The filename does not support wildcard characters. On Unix-like systems, if the first character is a ~ (tilde) character optionally followed by a username, it is replaced by the current or specified user's home directory.

target
The path of the target location where the file is to be moved.

Return codes:

0
File copied successfully.

Other
An operating system error code.

Example 8.10. RexxUtil — SysFileMove

```rexx
call SysFileMove "c:\temp\myfile.txt", "d:\myfolder\myCopy.txt"
call SysFileMove "/tmp/myfile.txt", "~/myCopy.txt"
```

8.20. SysFileSearch
SysFileSearch

Returns a list of all file lines containing the target string.

**Parameters:**

*target*
- The target search string.

*file*
- The searched file.

*stemarray*
- A Rexx stem or array variable for the returned lines. If a stem variable, *stem.0* is set to *n*, the number of lines returned, and the lines are returned in *stem.1* to *stem.n*.

*options*
- Any combination of the following one-character options:
  - **C**  
    Conducts a case-sensitive search.
  - **I**  
    Conducts a case-insensitive search.
  - **N**  
    Prepends file line numbers to any returned lines.

The default is a case-insensitive search without line numbers.

**Return codes:**

- **0**  
  Successful.
- **2**  
  Not enough memory.
- **3**  
  Error opening file.

**Example 8.11. RexxUtil — SysFileSearch**

```rexx
/* Find DEVICE statements in CONFIG.SYS */
call SysFileSearch "DEVICE", "C:\CONFIG.SYS", "file."
do i=1 to file.0
  say file.i
end

/* Output */
DEVICE=C:\SB16\DRV\CTSB16.SYS /UNIT=0 /BLASTER=A:240 I:5 D:1 H:5
DEVICE=C:\SB16\DRV\CTMMSYS.SYS
rem ***** DOS SCSI CDROM device drivers ***
DEVICE=C:\SCSI\ASPI8DOS.SYS /D
DEVICE=C:\SCSI\ASPICD.SYS /D:ASPICD0
rem ***** IDE CDROM device drivers
```

525
8.21. SysFileSystemType (Windows only)

Returns the type of file system (e.g. FAT or NTFS) used on a drive. Returns the null string if the drive is not accessible.

**Parameter:**

*drive*

A drive specification like `d:\` or `\share\path`. If omitted, the current drive is used.

**Example 8.12. RexxUtil — SysFileSystemType**

```
say SysFileSystemType("C:\")    -- NTFS
```

8.22. SysFileTree
Returns a list of all files that match a file specification. SysFileTree can return file information like date, time, size, attributes, and file name. The default format for date and time is platform specific.

*SysFileTree* uses operating system APIs to find the files. The found files are placed in the returned list in the order they are found by the operating system. No specific order can be assumed.

**Parameters:**

*filespec [required]*

The search pattern, the search file specification. This can not be the empty string.

The search pattern can be a full or partial path with wildcard characters in the last part of the path information, e.g. *.*.bat, or ../../*.*.sh, or c:\temp. Windows supports wildcard characters * and ?, Unix-like systems additionally support character classes in [ ] brackets, including - for ranges and ! for negation.

*stemarray [required]*

A Rexx stem or array variable for the returned matches. If a stem variable, stem.0 is set to n, the number of matches returned, and the matching files or directories are returned in stem.1 to stem.n.

*options [optional]*

A string with any combination of the following. If this argument is omitted, it defaults to B.

- **B**
  Search for both files and directories. This is the default.

- **D**
  Search for directories only.

- **F**
  Search for files only.

- **H**
  Returns correct file sizes for files larger than 9999999999 bytes (approx. 10 GB). Without this option, the maximum file size returned is 9999999999, even if the file is larger than this.

- **I**
  Perform a case-insensitive search for file names or directories. This option is only used on system that support case-sensitive file names and is ignored on systems like Windows where the file system is case-insensitive by default.

- **L**
  Returns the file date and time in the form YYYY-MM-DD  HH:MM:SS.

- **O**
  Returns only the fully-qualified file name.

- **S**
  Search subdirectories recursively.

- **T**
  Returns the file date and time in the form YY/MM/DD/HH/MM. If the L option is also specified then this option will be ignored.
**tattrib [optional] [Windows only]**

The target attribute mask for file specification matches. Only files that match the target mask are returned. The default mask is 

```
*****
```

This returns all files regardless of the settings (clear or set) of the Archive, Directory, Hidden, Read-Only, and System attributes. The target mask attributes must appear in the order **ADHRS**. This argument is Windows only. It is allowed but ignored on Unix-like systems.

**Target Mask Options**

*  
   The file attribute may be any state.

+  
   The file attribute must be set.

-  
   The file attribute must be cleared.

**Target Mask Examples**

```
****+*
```

Find all files with the Read-Only attribute set.

```
+***+*
```

Find all files with the Read-Only and Archive attributes set.

```
*++**
```

Find all hidden subdirectories.

```
---+-
```

Find all files with only the Read-Only attribute set.

**nattrib [optional] [Windows only]**

The new attribute mask for setting the attributes of each matching file. The default mask is 

```
*****
```

This means not to change the Archive, Directory, Hidden, Read-Only, and System attributes. The target mask attributes must appear in the order **ADHRS**. This argument is Windows only. It is allowed but ignored on Unix-like systems.

**New Attribute Mask Options**

*  
   Do not change the file attribute.

+  
   Set the file attribute.

-  
   Clear the file attribute.

**New Attribute Mask Examples**

```
****+*
```

Set the Read-Only attribute on all files.

```
-***+*
```

Set the Read-Only attribute and clear the Archive attribute of each file.
+*+++

Set all file attributes, except the directory attribute.

-----

Clear all attributes on all files.

Note

You cannot set the directory attribute on non-directory files. SysFileTree returns the file attribute settings after the new attribute mask has been applied.

Return codes:

0
Successful.

2
Not enough memory.

other
On Windows, the return code may be any System Error Code returned by the operating system.

Example 8.13. RexxUtil — SysFileTree

```rexx
/* Find all subdirectories on C: */
call SysFileTree "c:\.\*", "file", "SD"

/* Find all executable (.exe) files under Program Files*/
ret = SysFileTree('C:\Program Files\*.exe', f, 'FS')

/* Find all Read-Only files */
call SysFileTree "c:\.\*", "file", "S", "****+*

/* Clear Archive and Read-Only attributes of files that have them set */
call SysFileTree "c:\.\*", "file", "S", "****++", "++*-+

/****<< Sample Code and Output Example.>>********/
call SysFileTree "c:win\", file., "b"
do i = 1 to file.0
   say file.i   -- 3/12/21   1:56p           0  -D---  c:\Windows
end
```

8.23. SysFork (Unix-like systems only)

Returns

Returns the process id to the parent process.

Returns 0 to the spawned process.
Example:

This is a complete working example. It can be cut and pasted into a file and executed on a Unix-like system.

Example 8.14. RexxUtil — SysFork

```rexx
/* Example SysFork() and SysWait() */
pid = SysFork()
if pid == 0 then do
  say "I am the child."
  code = executeChild()
  say "Child : done with execution, will exit with" code
  exit code
end
else do
  say 'I am the parent, child pid is:' pid
  code = executeParent()
  say 'Parent: going to wait for child.'
  code = SysWait()
  say 'Parent: back from waiting. Child exit code:' code
end

say 'Operating system version:' SysVersion()
::routine executeChild
  say 'Child : will sleep 1 second.'
  j = SysSleep(1)
  say 'Child : done sleeping 1. Will do some calculations.'
  total = 0
  do 786
    total += 3
  end
  say 'Child : 3 * 786 is:' total
  say 'Child : will sleep 2 seconds.'
  j = SysSleep(2)
  say 'Child : done sleeping 2. Will do some calculations.'
  total = 0
  do 1865
    total += 7
  end
  say 'Child : 7 * 1865 is:' total
  say 'Child : will sleep 2 seconds.'
  j = SysSleep(2)
  say 'Child : done sleeping 2.'
  say 'Child : done executing, will return 0.'
  return 0
::routine executeParent
  say 'Parent: 3 * 786 is:' (3 * 786)
  j = SysSleep(2)
  say 'Parent: 7 * 1865 is:' (7 * 1865)
  return 0
```

8.24. SysFormatMessage

Formats and returns a message text, replacing placeholders &n with specified message inserts. SysFormatMessage can replace up to nine placeholders.

message
The message text. This string may include up to nine placeholders &1, &2 ... &9 in any order.

substitutions
Text inserts for message placeholders. This can a be a single string, an Array of strings, or any other object supporting a makeArray method. The first array item is used to replace the &1 placeholder, the second item replaces &2, up to a maximum of nine items.

Example 8.15. RexxUtil — SysFormatMessage

```rexx
s = .Stream~new("xyzzy")
if \\s~open("read")~startsWith("READY") then
  -- Cannot read file 'xyzzy': ERROR:2 No such file or directory
say SysFormatMessage("Cannot read file '&1': &2", (s~string, s~description))
```

8.25. SysFromUnicode (Windows only)

Maps a Unicode character string to an ASCII character string. The new character string and additional information is returned in the outstem.

Parameters:

string
A string containing the Unicode characters to be mapped.

codepage
Specifies the code page used to perform the conversion. This parameter can be the value of any code page that is installed or available in the system. The default is the current original equipment manufacturer (OEM) code-page identifier for the system.

You can also specify one of the following values:

- ACP
  ANSI code page.

- OEMCP
  OEM code page.

- SYMBOL
  symbol code page.
SysFromUnicode (Windows only)

THREAD_ACP
current thread's ANSI code page.

UTF7
translate using UTF-7.

UTF8
translate using UTF-8. When this is set, **mappingflags** must be set.

**mappingflags**
Specifies the handling of unmapped characters. The function performs more quickly when none of these flags is set.

The following flags can be used:

COMPOSITECHECK
Converts composite characters to precomposed characters.

SEPCHARS
Generates separate characters during conversion. This is the default conversion behavior.

DISCARDNS
Discards nonspacing characters during conversion.

DEFAULTCHAR
Replaces non-convertible characters with the default character during conversion.

When **compositecheck** is specified, the function converts composite characters to precomposed characters. A composite character consists of a base character and a nonspacing character, each having different character values. A precomposed character has a single character value for a combination of a base and a nonspacing character. In the character è, the "e" is the base character, and the "grave" accent mark is the nonspacing character.

When **compositecheck** is specified, it can use the last three flags in this list (**discardns**, **sepchars**, and **defaultchar**) to customize the conversion to precomposed characters. These flags determine the function's behavior when there is no precomposed mapping for a combination of a base and a nonspacin character in a Unicode character string. These last three flags can be used only if the **compositecheck** flag is set. The function's default behavior is to generate separate characters (**sepchars**) for unmapped composite characters.

**defaultchar**
Character to be used if a Unicode character cannot be represented in the specified code page. If this parameter is NULL, a system default value is used. The function is faster when **defaultchar** is not used.

**outstem**
The name of the stem variable that will contain the converted result. If the conversion was successful the stem will be composed of the following value(s):

**outstem.**!USEDDEFAULTchar
This variable will be set to "1" if the **defaultchar** was used during the conversion and "0" if it was not.

**outstem.**!TEXT
This variable will contain the converted string.
Return codes:

0
No errors.

Other
An operating system error code. On Windows, this may be one of the following, but could be others.

87
Incorrect code page or codepage value.

1004
Invalid mapping flags.

8.26. SysGetErrorText

SysGetErrorText(errornumber)

Obtains a string describing the system error identified by the error number.

Returns a string with the description of the error, or an empty string if no description is available.

Parameter:

errornumber
The error number for which a description is to be retrieved. Note that both the error number and the descriptive text are operating-system specific.

Example 8.16. RexxUtil — SysGetErrorText

-- Windows: Cannot create a file when that file already exists.
say SysGetErrorText(SysMkDir("\Users"))

-- Unix-like system: File exists
say SysGetErrorText(SysMkDir("/usr"))

8.27. SysGetFileDateTime

SysGetFileDateTime(filename, timesel)

Returns a file timestamp.

Parameters

filename
The name of a file or directory for which a timestamp should be returned. The name does not support wildcard characters. On Unix-like systems, if the first character is a ~ (tilde) character optionally followed by a username, it is replaced by the current or specified user’s home directory.
timesel
An optional selector specifying the type of timestamp to be returned. Depending on the operating
system and the file system, one of the following timestamps may be selected (only the first letter is
required):
ACCESS
The date and time when the file was last accessed. Not available on FAT file systems.
CREATE
The creation date and time. Not available on Unix-like systems or on FAT file systems.
WRITE
The date and time the file was last modified. This is the default if timesel is omitted.

Returns
The file timestamp in the format yyyy-mm-dd hh:mm:ss, or -1 to indicate that the timestamp is
unavailable or could not be retrieved.

See also File methods lastAccessed (Attribute) and lastModified (Attribute), and Rexxutil
SysSetFileDateTime.

Example 8.17. RexxUtil — SysGetFileDateTime

```
say SysGetFileDateTime(".")           -- (e. g.) 2020-04-01 17:45:20
say SysGetFileDateTime("main.c", "w") -- (e. g.) 2020-02-11 05:00:00
```

8.28. SysGetKey

```
SysGetKey(opt)
```

Reads and returns the next key from the keyboard buffer. If the keyboard buffer is empty, SysGetKey
waits until a key is pressed. Unlike the CHARIN built-in function, SysGetKey does not wait until the
Enter key is pressed.

Parameter:

opt
An option controlling screen echoing. Allowed values are:
"ECHO"
    Echo the pressed key to the screen. This is the default.
"NOECHO"
    Do not echo the pressed key.

8.29. SysGetLongPathName (Windows only)

```
SysGetLongPathName(path)
```

Returns the long version of the specified path.

Parameter:
**path**

A relative or a fully qualified path of an existing file or directory.

**Returns:**

The long version of `path`, or the null string, if `path` doesn't exist or the call to the Windows API `GetLongPathName` fails for any other reason.

---

**Example 8.18. RexxUtil — SysGetLongPathName**

```
say SysGetLongPathName("C:\progra~1")            \--- C:\Program Files
say SysGetLongPathName("\windows\explorer.exe")  \--- \Windows\explorer.exe
```

---

### 8.30. SysGetMessage (Unix-like systems only)

Returns a message retrieved from a Unix message catalog, with placeholders replaced with specified message inserts.

SysGetMessage is implemented on Unix-like systems only. It always retrieves messages from the catalog's message set 1. To retrieve messages from a specified message set, use `SysGetMessageX (Unix-like systems only)`.

To create a Unix message catalog use the `gencat` command.

**Parameters:**

`num`

The message number.

`filename`

The name of the catalog file containing the message. If `filename` is omitted or specified as `rexx.cat` the internal Rexx message catalog is searched. Otherwise the Unix message catalog is located using the absolute `filename` path or searched along the `NLSPATH` environment variable.

`str`

Zero to nine replacement strings for placeholders in the message. The first replacement string is for placeholder `&1`, the second for `&2`, up to placeholder `&9`. If a placeholder has no replacement string specified, it is replaced by the null string. Any unused replacement string is ignored. Other placeholders like `%s` are not replaced and are returned unchanged with the message.

---

**Example 8.19. RexxUtil — SysGetMessage**

```
say SysGetMessage(485, "rexx.cat", "foo") -- Class "foo" not found.
```

---

### 8.31. SysGetMessageX (Unix-like systems only)

---
SysGetMessageX (Windows only)

Returns a message retrieved from a Unix message catalog, with placeholders replaced with specified message inserts.

SysGetMessageX is implemented on Unix-like systems only.

To create a Unix message catalog use the `gencat` command.

**Parameters:**

- **set**
  The message set.

- **num**
  The message number.

- **filename**
  The name of the catalog file containing the message. If `filename` is omitted or specified as `rexx.cat` the internal Rexx message catalog is searched. Otherwise the Unix message catalog is located using the absolute `filename` path or searched along the `NLSPATH` environment variable.

- **str**
  Zero to nine replacement strings for placeholders in the message. The first replacement string is for placeholder `&1`, the second for `&2`, up to placeholder `&9`. If a placeholder has no replacement string specified, it is replaced by the null string. Any unused replacement string is ignored. Other placeholders like `%s` are not replaced and are returned unchanged with the message.

**Example 8.20. RexxUtil — SysGetMessageX**

```rexx
say SysGetMessageX(1, 485, "rexx.cat", "foo") -- Class "foo" not found.
```

8.32. SysGetShortPathName (Windows only)

Returns the short version of the specified `path`.

**Parameter:**

- **path**
  A relative or a fully qualified path of an existing file or directory.

**Returns:**

The short version of `path`, or the null string, if `path` doesn't exist or the call to the Windows API `GetShortPathName` fails for any other reason.

**Example 8.21. RexxUtil — SysGetShortPathName**

```rexx
say SysGetShortPathName("C:\Program Files") -- C:\PROGRA~1
```
8.33. SysIni (Windows only)

Allows limited access to INI file variables. Variables are stored in the INI file under Application Names and their associated key names or keywords. You can use SysIni to share variables between applications or as a way of implementing GLOBALV in the Windows operating system. Be careful when changing application profile information.

Note

SysIni works on all types of data stored in an INI file (text, numeric, or binary).

When SysIni successfully sets or deletes key values, it returns "". For a successful query, it returns the value of the specified application keyword.

SysIni may return the string **ERROR**: when an error occurs. Possible error conditions include:

- An attempt was made to query or delete an application/key pair that does not exist.
- An error opening the profile file occurred. You may have specified the current user or system INI file with a relative file specification. Make sure to use the full file specification (specify drive, path, and file name).

Parameters:

**inifile**

The name of the INI file with which you would like to work. The default is WIN.INI.

Note

If this argument does not contain a fully qualified file name, the Windows operating system searches for the file in the Windows directory. Therefore to work with a file outside of the Windows directory, specify the full path name of the file.

**app**

The application name or some other meaningful value with which you want to store keywords (some sort of data).

**key**

The name of a keyword to hold data.

**val**

The value to associate with the keyword of the specified application. This can be "DELETE:" or "ALL:".
stemarray

A Rexx stem or array variable for the returned names. If a stem variable, stem.0 is set to n, the number of names returned, and the names are returned in stem.1 to stem.n.

SysIni has six modes. The modes and the syntax variations are as follows:

Sets a single key value.

Queries a single key value.

Deletes a single key.

Deletes an application and all associated keys.

Queries names of all keys associated with a certain application.

Queries the names of all applications.

Example 8.22. RexxUtil — SysIni

```rexx
/**** Save the user entered name under the key "NAME" of *****
**** the application "MYAPP".           ****/
pull name .
call SysIni , "MYAPP", "NAME", name /* Save the value */
say SysIni(, "MYAPP", "NAME")        /* Query the value */
call SysIni , "MYAPP"                 /* Delete all MYAPP info */
exit
```

Example 8.23. RexxUtil — SysIni

```rexx
/**** Type all WIN.INI file information to the screen *****/
call SysIni "WIN.INI", "All:", "Apps."  
if Result \= "ERROR:" then
```
8.34. SysIsFile

Checks for the existence of a file.

On Unix-like systems block devices are also considered to be regular files by this function.

**Parameters:**

**filename**

The name of the file to check for the existence of. The name does not support wildcard characters. On Unix-like systems, if the first character is a ~ (tilde) character optionally followed by a username, it is replaced by the current or specified user's home directory.

**Returns:**

0

The file does not exist.

1

The file exists.

**Example 8.24. RexxUtil — SysIsFile**

```rexx
if SysIsFile(InputFile) then say "File Exists!"
else say "File does not exist."
```

8.35. SysIsFileCompressed (Windows only)

Checks if a file is compressed. This function does not support wildcard specifications.

**Parameters:**

**filename**

The name of the file to check.

**Returns:**

0

The file is not compressed or does not exist.
1

The file is compressed.

Example 8.25. RexxUtil — SysIsFileCompressed

if SysIsFileCompressed(InputFile) then say "File is compressed!"
else say "File is not compressed or does not exist."

8.36. SysIsFileDirectory

SysIsFileDirectory(
  $dirname$
)

Checks for the existence of a subdirectory.

Parameters:

dirname

The name of the subdirectory to check for the existence of. The directory name does not support wildcard characters. On Unix-like systems, if the first character is a ~ (tilde) character optionally followed by a username, it is replaced by the current or specified user's home directory.

Returns:

0

The subdirectory does not exist.

1

The subdirectory exists.


if SysIsFileDirectory(InputFile) then say "Subdirectory Exists!"
else say "Subdirectory does not exist."

8.37. SysIsFileEncrypted (Windows only)

SysIsFileEncrypted(
  $filename$
)

Checks if a file is encrypted. This function does not support wildcard specifications.

Parameters:

filename

The name of the file to check.

Returns:

0

The file is not encrypted or does not exist.

1

The file is encrypted.
Example 8.27. RexxUtil — SysIsFileEncrypted

```plaintext
if SysIsFileEncrypted(InputFile) then say "File is encrypted!"
else say "File is not encrypted or does not exist."
```

### 8.38. SysIsFileLink

**SysIsFileLink(linkname)**

Checks for the existence of a link.

**Parameters:**

- **linkname**
  
  The name of the link to check for the existence of. The name does not support wildcard characters. On Unix-like systems, if the first character is a `~` (tilde) character optionally followed by a username, it is replaced by the current or specified user's home directory.

**Returns:**

- **0**
  
  The link does not exist or it is not a link.

- **1**
  
  The link exists.

Example 8.28. RexxUtil — SysIsFileLink

```plaintext
if SysIsFileLink(InputFile) then say "Link Exists!"
else say "Link does not exist."
```

### 8.39. SysIsFileNotContentIndexed (Windows only)

**SysIsFileNotContentIndexed(filename)**

Checks if a file is flagged to be indexed by the Index Service. This function does not support wildcard specifications.

**Parameters:**

- **filename**
  
  The name of the file to check.

**Returns:**

- **0**
  
  The file is not flagged to be Indexed or does not exist.

- **1**
  
  The file is flagged to be Indexed.
Example 8.29. RexxUtil — SysIsFileNotContentIndexed

```r
if SysIsFileNotContentIndexed(InputFile) then say "File is flagged to be Indexed!"
else say "File is not flagged to be Indexed."
```

8.40. SysIsFileOffline (Windows only)

```r
SysIsFileOffline(
)```

Checks if a file is flagged as Offline. This function does not support wildcard specifications.

**Parameters:**

- `filename`
  The name of the file to check.

**Returns:**

- `0`
  The file is not flagged as Offline or does not exist.
- `1`
  The file is flagged as Offline.

Example 8.30. RexxUtil — SysIsFileOffline

```r
if SysIsFileOffline(InputFile) then say "File is flagged as Offline!"
else say "File is not flagged as Offline."
```

8.41. SysIsFileSparse (Windows only)

```r
SysIsFileSparse(
)```

Checks if a file is flagged as Sparse. This function does not support wildcard specifications.

**Parameters:**

- `filename`
  The name of the file, subdirectory or link to check.

**Returns:**

- `0`
  The file is not flagged as Sparse or does not exist.
- `1`
  The file is flagged as Sparse.

Example 8.31. RexxUtil — SysIsFileSparse

```r
if SysIsFileSparse(InputFile) then say "File is Sparse!"
else say "File is not Sparse."
```
8.42. SysIsFileTemporary (Windows only)

Checks if a file is flagged as Temporary. This function does not support wildcard specifications.

Parameters:

filename
   The name of the file, subdirectory or link to check.

Returns:

0
   The file is not flagged as Temporary or does not exist.

1
   The file is flagged as Temporary.

Example 8.32. RexxUtil — SysIsFileTemporary

if SysIsFileTemporary(InputFile) then say "File is Temporary!"
else say "File is not Temporary."

8.43. SysLinVer (Linux Only)

Returns a string identifying the Linux system version. The first word of the returned string is Linux and the remaining part of the string identifies the version. Rexx code should not rely on the exact format returned.

Possible output might be:

Example 8.33. RexxUtil — SysLinVer

-- CentOS 7
say SysVersion() -- Linux 3.10.0-957.10.1.el7.x86_64
-- Ubuntu 16.04
say SysVersion() -- Linux 4.4.0-154-generic
-- Raspbian
say SysVersion() -- Linux 4.14.70-v7+
-- SLES/390
say SysVersion() -- Linux 3.12.74-60.64.40-default

See also functions SysVersion and SysWinVer (Windows only).

8.44. SysLoadRexxMacroSpace
Loads functions from a saved macrospace file. SysLoadRexxMacroSpace returns the RexxLoadMacroSpace return code.

**Parameter:**

*file*

The file used to load functions into the Rexx macrospace. SysSaveRexxMacroSpace must have created the file.

### 8.45. SysMkDir

**Windows:**

```
SysMkDir(dirspec)
```

**Unix-like systems:**

```
SysMkDir(dirspec, mode)
```

Creates the specified directory. This function will only create the final directory in the path, all intermediate directories must already exist.

**Parameter:**

*dirspec*

The directory to be created. The directory name does not support wildcard characters. On Unix-like systems, if the first character is a ~ (tilde) character optionally followed by a username, it is replaced by the current or specified user's home directory.

*mode* (Unix-like systems only)

The file permission bits of the new directory as a decimal number between 0 (octal 000) and 511 (octal 777). If not specified, *mode* defaults to 511 (octal 777).

Any mode restrictions specified by *umask* will always be taken into account.

**Return codes:**

0

Directory creation was successful.

Other

An **operating system error code**. On Windows, this may be one of the following, but could be others.

2

File not found.

3

Path not found.

5

Access denied.

26

Not a DOS disk.
87
Invalid parameter.

108
Drive locked.

183
Directory already exists.

206
File name exceeds range.

Example 8.34. RexxUtil — SysMkDir

call SysMkDir "backup"

8.46. SysQueryProcess

Windows

Unix-like systems

Retrieves information about the current process or Windows thread.
Parameter:

info
The kind of information requested:

PID
Returns the process ID of the current process.

PPID
Returns the parent process ID of the current process.

TID
Returns the thread ID of the current thread.

PPRIO
Returns the priority class of the current process.

TPRIO
Returns the relative priority of the current thread.

PTIME
Returns time information on the current process.

TTIME
Returns time information on the current thread.

PMEM
Returns the maximum memory (RSS) used by the current process.

PRCVDSIG
Returns the number of signals that have been received by the process.

Return codes:

• For PID, PPID or TID: an ID
• For Windows PPRIO: "IDLE", "NORMAL", "HIGH", "REALTIME", or "UNKNOWN"
• For Unix-like systems PPRIO: a number from -20 to +20.
• For TPRIO: "IDLE", "LOWEST", "BELOW_NORMAL", "NORMAL", "ABOVE_NORMAL", "HIGHEST", "TIME_CRITICAL", or "UNKNOWN"
• For Windows PTIME or TTIME: the creation date and time, the amount of time that the process executed in kernel mode, and the amount of time that the process executed in user mode
• For Unix-like systems PTIME: the summary and the duration that the process executed in kernel mode, and the duration that the process executed in user mode

8.47. SysQueryRexxMacro

Queries the existence of a macrospace function. SysQueryRexxMacro returns the placement order of the macrospace function or a null string ("") if the function does not exist in the macrospace.

Parameter:
name
   The name of a function in the Rexx macrospace.

8.48. SysReorderRexxMacro

Reorders a routine loaded in the Rexx macrospace. SysReorderRexxMacro returns the RexxxReorderMacro return code.

Parameters:

name
   The name of a function in the macrospace.

order
   The new macro search order. The order can be "B" (Before) or "A" (After).

8.49. SysRmDir

Deletes a specified file directory without your confirmation.

Parameter:

dirspec
   The directory that should be deleted. The name does not support wildcard characters. On Unix-like systems, if the first character is a ~ (tilde) character optionally followed by a username, it is replaced by the current or specified user's home directory.

Return codes:

0
   Directory removal was successful.

Other
   An operating system error code. On Windows, this may be one of the following, but could be others.

2
   File not found.

3
   Path not found.

5
   Access denied or busy.

16
   Current directory.

26
   Not a DOS disk.
32
  Sharing violation.

108
  Drive locked.

123
  Invalid name.

145
  Directory not empty.

146
  Is Subst Path.

147
  Is Join Path.

206
  File name exceeds range.

---

Example 8.35. RexxUtil — SysRmDir

```rexx
call SysRmDir "backup"
```

### 8.50. SysSaveRexxMacroSpace

Saves the Rexx macrospace. SysSaveRexxMacroSpace returns the RexxSaveMacroSpace return code.

**Parameter:**

`file`

The file used to save the functions in the Rexx macrospace.

### 8.51. SysSearchPath

Returns the absolute path of `filename` if it can be found along any of the paths specified in environment variable `path`. Returns the null string otherwise.

**Parameters**

- `path`
  
  An environment variable name. The environment variable must contain a list of paths, like in `PATH`.

filename
The name of the file to be searched. The name can optionally include a relative or an absolute path, but the name part must be a file, not a directory. If the name specified with an absolute path, it is returned without searching along the environment variable's paths. The name does not support wildcard characters. On Unix-like systems, if the first character is a ~ (tilde) character optionally followed by a username, it is replaced by the current or specified user's home directory.

option
Specifies where the search starts.

C
Starts the search at the current directory and then along the specified path. This is the default.

N
Only searches along path.

See also File method searchPath (Class Method).

Example 8.36. RexxUtil — SysSearchPath

```
say SysSearchPath("PATH", "rexx")  -- (Unix e.g.) /usr/local/bin/rexx
say SysSearchPath("PATH", "rexx.exe") -- (Windows e.g.) C:\Program Files\ooRexx\rexx.exe
```

8.52. SysSetFileDateTime

Sets the last-modified timestamp of a file.

If both newdate and newtime are omitted, the last-modified timestamp is set to the current date and time, similar to what the touch command on Unix-like systems does.

If only one of the newdate and newtime arguments is supplied, the respective other last-modified date/time portion stays unchanged.

Parameters

filename
The name of a file or directory for which the last-modified timestamp should be updated. The name does not support wildcard characters. On Unix-like systems, if the first character is a ~ (tilde) character optionally followed by a username, it is replaced by the current or specified user's home directory.

newdate
An optional new date in the format \texttt{yyyy-mm-dd}. On Windows \texttt{yyyy} must be greater than or equal to \texttt{1800}.

newtime
An optional new time in the format \texttt{hh:mm:ss} (24-hour format).

Return codes
The file date and/or time were updated successfully.

-1
The timestamp update failed.

See also File methods `lastModified (Attribute)` and `lastModified (Attribute)`, and Rexxutil `SysGetFileDateTime`.

**Example 8.37. RexxUtil — SysSetFileDateTime**

```plaintext
call SysSetFileDateTime ",."
   -- touch current directory
call SysSetFileDateTime "main.c", , "05:00:00"
   -- change file time only
```

### 8.53. SysSetPriority

Changes the priority of the current process. A return code of 0 indicates no error.

**Parameters:**

**class**

The new process priority class. The allowed classes are:
- 0 or "IDLE"
  - Idle time priority
- 1 or "NORMAL"
  - Regular priority
- 2 or "HIGH"
  - High or time-critical priority
- 3 or "REALTIME"
  - Real-time priority

**delta**

The change applied to the process priority level. `delta` must be in the range -15 to +15. It can also be a symbolic name:
- "IDLE" for -15
- "LOWEST" for -2
- "BELOW_NORMAL" for -1
- "NORMAL" for 0
- "ABOVE_NORMAL" for 1
- "HIGHEST" for 2
- "TIME_CRITICAL" for 15

0
No errors.
Other

An *operating system error code*. On Windows, this may be one of the following, but could be others.

307
Invalid priority class.

8.54. **SysShutdownSystem (Windows only)**

Provides an interface to the `InitiateSystemShutdown()` API on Windows. If the user has sufficient privileges, this function can be used to shut down the local machine or a remote system. In general all users have sufficient privileges to shut down the local machine and only Administrators have sufficient privileges to shut down remote machines.

The user of this function is *strongly* encouraged to read the Microsoft documentation for `InitiateSystemShutdown()` to understand the finer points to using this function. The documentation is freely available online. A Google search using `InitiateSystemShutdown MSDN` will provide a link to the documentation. In particular, this function can be used to force a shut down of a system while users are logged on and applications have unsaved data.

If the `timeout` argument is not 0, a shut down dialog is displayed on the machine being shut down, naming the user who initiated the shut down, a timer counting down the seconds until the machine is shut down, and prompting the user to log off. This dialog can be moved but it can not be closed and remains on top of all other windows on the system.

If any open application has unsaved data, the operating system gives the application a chance to prompt the user to save and close the application. If the `force` argument is false, the shut down will be delayed until the user responds, and ultimately the user could cancel the shut down. If the `force` argument is true the system will force the application closed whether the data gets saved or not. The application is still given a chance to prompt the user to save the data, but the user only has a few seconds to respond before the system forcibly closing the application.

The implications of the preceding two paragraphs are this, if the `timeout` argument is 0 and the `force` argument is true, the system immediately shuts down and any unsaved data is forever lost. This is why the user of this function is encouraged to fully understand this function before using it.

**Parameters:**

---

**Warning**

Once this function has been called, the initiated shutdown cannot be aborted—data loss may occur!
computer
Indicates which system to shut down. If omitted or the empty string, the local machine is shut down. Otherwise this should be the network name of the remote machine.

message
An additional message that is added to the shut down dialog. If omitted, no additional message is added.

timeout
The time, in seconds, before the system is shut down. If this value is 0, the system is immediately shut down and no shut down dialog is displayed. The shut down can not be aborted. If omitted the default time out is 30 seconds.

force
If this argument is true, applications with unsaved data will be forced close by the system whether the data is saved or not. If false, the shut down is delayed until applications with unsaved data responds. If an application does not respond, the user will be prompted by the system to end the application. At this point, if the user chooses not to forcibly end the application, the shut down will be aborted.

Please note some consequences of this argument as described by Microsoft. If this argument is false, i.e. applications are not forced to close, and an application with unsaved changes is running on the console session, the shutdown will remain in progress until the user logged into the console session aborts the shutdown, saves changes, closes the application, or forces the application to close. During this period, the shutdown may not be aborted except by the console user, and another shutdown may not be initiated. Using true for this argument prevents that situation. But, using true can also result in unsaved data being lost.

reboot
If this argument is true, the system is rebooted after the shut down. If it is false, the system is shut down. The default if omitted is false.

Returns:

Returns 0 for success or a Windows System Error code for failure. For instance a return of 1300 would indicate the user does not have sufficient privileges to shut down the named system. Use SysGetErrorText to get a generic text description for any Windows System Error code. For example, the description for error code 1300 is Not all privileges referenced are assigned to the caller.

8.55. SysSleep

SysSleep( delay )
Pauses a Rexx program for a specified time interval.

Parameter:

delay
The number of seconds for which the program is to be paused, which must be in the range 0 to 2147483 for Windows, and 0 to 999999999 for Unix-like systems.
Example 8.38. RexxUtil — SysSleep

```
Say "Now paused for 2 seconds ..."
Call SysSleep 2
Say "Now paused for 0.1234567 seconds ..."
Call SysSleep 0.1234567
```

8.56. Sysstemcopy

Covers items from the source stem to the target stem. Items in the source stem are copied starting at
the from index (default is 1) into the target stem beginning at the to index (default is 1). The number of
items to be copied to the target stem can be specified with the count. The default is to copy all items in
the source stem.

You can also specify that the items are to be inserted into the target stem at the position and the
existing items are shifted to the end.

This function operates only on stem arrays that specify the number of items in stem.0 and all items
must be numbered from 1 to n without omitting an index.

Parameters:

fromstem
  The name of the source stem.

tostem
  The name of the target stem.

from
  The first index in the source stem to be copied.

to
  The position at which the items are to be inserted in the target stem.

count
  The number of items to be copied or inserted.

insert
  Either of the following values:
    I
      Insert items.
Overwrite items.

Return codes:

0
The stem was copied successfully.

-1
Copying the stem failed.

Example 8.39. RexxUtil — SysStemCopy

Source.0 = 3
Source.1 = "Hello"
Source.2 = "from"
Source.3 = "Rexx"
Call SysStemCopy "Source.", "Target."
Call SysStemCopy "Source.", "Target.", 1, 5, 2, "I"

8.57. SysStemDelete

Deletes the specified item at the index startitem in the stem. If more than one item is to be deleted the itemcount must be specified. After deleting the requested items the stem is compacted, which means that items following the deleted items are moved to the vacant positions.

This function operates only on stem arrays that specify the number of items in stem.0 and all items must be numbered from 1 to n without omitting an index.

Parameters:

stem
The name of the stem from which the item is to be deleted.

startitem
The index of the item to be deleted.

itemcount
The number of items to be deleted if more than one.

Return codes:

0
Deleting was successful.

-1
Deleting failed.
8.58. Sys StemInsert

Sys StemInsert( stem, position, value )

Inserts a new item at position in the stem. All items in the stem following this position are shifted down by one position.

This function operates only on stem arrays that specify the number of items in stem.0 and all items must be numbered from 1 to n without omitting an index.

Parameters:

stem
The name of the stem in which an item is to be inserted.

position
The index at which the new item is to be inserted.

value
The value of the new item.

Return codes:

0
Inserting was successful.

-1
Inserting failed.

Example 8.41. RexxUtil — Sys StemInsert

Call Sys StemInsert "MyStem.", 5, "New value for item 5"

8.59. Sys StemSort

Sys StemSort( stem, start, end, firstcol, lastcol )

Sorts all or the specified items in the stem. The items can be sorted in ascending or descending order and the case of the strings being compared can be respected or ignored. Sorting can be
further narrowed by specifying the first and last item to be sorted or the columns used as sort keys. Because the sort uses a quick-sort algorithm, the order of sorted items according to the sort key is undetermined.

This function operates only on stems that specify the number of items in stem.0 and all items must be numbered from 1 to n without omitting an index. A value of 0 in stem.0 is also valid but no sort will be performed.

**Parameters:**

- **stem**
  The name of the stem to be sorted.

- **order**
  Either "A" for ascending or "D" for descending. The default is "A".

- **type**
  The type of comparison: either "C" for case or "I" for ignore. The default is "C".

- **start**
  The index at which the sort is to start. The default is 1.

- **end**
  The index at which the sort is to end. The default is the last item.

- **firstcol**
  The first column to be used as sort key. The default is 1.

- **lastcol**
  The last column to be used as sort key. The default is the last column.

**Return codes:**

- **0**
  The sort was successful.

- **-1**
  The sort failed.

---

**Example 8.42. RexxUtil — SysStemSort**

```rxx
/* sort all elements descending, use cols 5 to 10 as key */
Call SysStemSort "MyStem.", "D", , , ,5, 10

/* sort all elements ascending, ignore the case */
Call SysStemSort "MyStem.", "A", "I"

/* sort elements 10 to 20 ascending, use cols 1 to 10 as key */
Call SysStemSort "MyStem.", , ,10, 20, 1, 10
```

**8.60. SysSwitchSession (Windows only)**
Brings the named window to the foreground. Modern versions of Windows do not always allow a window to be brought to the foreground programmatically. Instead, the icon for the window on the task bar is set to flashing.

**Parameter:**

name

The name of the window you want to be the foreground window. The name of the window must exactly match the title of the window, but is not case sensitive. The title of a window is the text displayed in its title bar.

0 is returned on success and a Windows *System Error code* on failure.

### 8.61. SysSystemDirectory (Windows only)

**SysSystemDirectory()**

Returns the Windows system directory.

### 8.62. SysTempFileName

**SysTempFileName(templatem,filler)**

Returns a unique name for a file or directory that does not currently exist. If an error occurs or SysTempFileName cannot create a unique name from the template, it returns a null string. SysTempFileName is useful when a program requires a temporary file.

**Parameters:**

*template*

The *template* is a valid file or directory specification including a path, with between one and nine filler characters.

SysTempFileName generates filler character replacements with a random number algorithm. If the resulting file or directory already exists, SysTempFileName increments the replacement value until all possibilities have been exhausted.

The file path and name of the template can be of any length. If the template contains no or more than nine filler characters, SysTempFileName returns a null string. Filler characters are allowed both in the path and the name part of *template*. The unique name is created solely by substituting a random numeric digit for each filler character. If there is a directory name part of the template in addition to the file name part, and the directory name part contains a non-existent directory, the function will succeed. However, the temporary file will not be writable unless the user first creates the non-existent directory, or directories.

*filler*

The filler character used in *template*. SysTempFileName replaces each filler character in *template* with a random numeric digit. The resulting string represents a file or directory that does not exist. The default filler character is ?.

See also *File* method *temporaryPath (Class Method)*.
Example 8.43. RexxUtil — SysTempFileName

```
say SysTempFileName("file???")                -- (e. g.) C:\Users\User1\file392
-- (e. g.) /home/user1/file384
say SysTempFileName("/temp/myfile.***", ")**") -- (e. g.) C:\temp\myfile.938
-- (e. g.) /temp/myfile.341
```

8.63. SysTextScreenRead (Windows only)

![Diagram](image)

Returns characters from a specified console screen buffer location.

**Parameters:**

- **row**
  - The row from which to start reading. Console screen buffer rows are numbered from top to bottom starting with 0.

- **col**
  - The column from which to start reading. Console screen buffer columns are numbered from left to right starting with 0.

- **len**
  - The number of characters to read. The default is to read to the end of the console screen buffer.

**Limitation**

This function reads in only screen characters and does not consider the color attributes of each character read. When restoring a character string to the screen with SAY or the CHAROUT built-in function, the previous color settings are lost.

**Note**

The Windows API used by function is not recommended any more.

Example 8.44. RexxUtil — SysTextScreenRead

```
// read the entire console screen buffer
screen = SysTextScreenRead(0, 0)
// read one screen buffer line
```
8.64. **SysTextScreenSize (Windows only)**

Returns or sets the size of the console screen buffer, returns or sets the position of the current console window rectangle, or returns the maximum possible console window size.

**Parameters:**

**option**

Specifies which size or position is to be returned or set.

**BUFFERSIZE**

Returns or sets the size of the current console buffer. This is the default.

If a size is returned, it is a string of the form `rows columns`, where `rows` is the height and `columns` is the width in terms of character cells. If no character-mode console is attached, the string `0 0` is returned.

`rows, columns`

If both optional `rows` and `columns` are specified, the console buffer size is changed to the height and width given by `rows` and `columns`. The buffer size cannot be smaller than the size of the current console window. If the function succeeds, it returns 0, otherwise a Windows System Error code is returned.

**WINDOWRECT**

Returns or sets the position of the current console window rectangle.

If a position is returned, it is a string of the form `top left bottom right`, where `(top, left)` and `(bottom, right)` are absolute character-cell coordinates in the current console buffer, with the origin (0, 0) at the upper-left corner. If no character-mode console is attached, the string `0 0 0 0` is returned.

`top, left, bottom, right`

If all optional arguments `top`, `left`, `bottom`, and `right` are specified, the position of the current console window rectangle is changed to the given `(top, left)` and `(bottom, right)` positions. Both positions must be within the boundaries of the current console buffer. If the function succeeds, it returns 0, otherwise a Windows System Error code is returned.

**MAXWINDOWSIZE**

Returns the maximum possible console window size, considering the size of the current console screen buffer, the chosen window font, and the screen size.
SysToUnicode (Windows only)

The size is returned as a string of the form rows columns, where rows is the height and columns is the width in terms of character cells. If no character-mode console is attached, the string 0 0 is returned.

Example 8.45. RexxUtil — SysTextScreenSize

```rexx
parse value SysTextScreenSize() with rows columns
parse value SysTextScreenSize("windowrect") with top left bottom right
say "current console buffer is" rows "rows by" columns "columns"
say "current window rectangle is ("top"," left") ("bottom"," right")"
```

8.65. SysToUnicode (Windows only)

Maps a character string to a Unicode string.

**Parameters:**

- **string**
  A string that should be mapped to Unicode characters.

- **codepage**
  Specifies the code page used to perform the conversion. This parameter can be the value of any code page that is installed or available in the system. The default is the current original equipment manufacturer (OEM) code-page identifier for the system.

  You can also specify one of the following values:

  - ACP
    ANSI code page.

  - OEMCP
    OEM code page.

  - SYMBOL
    symbol code page.

  - THREAD_ACP
    current thread's ANSI code page.

  - UTF7
    translate using UTF-7.

  - UTF8
    translate using UTF-8. When this is set, translateflags must be set.

- **translateflags**
  Indicates whether to translate to precomposed or composite-wide characters (if a composite form exists), whether to use glyph characters in place of control characters, and how to deal with invalid characters.
You can specify a combination of the following flags:

**PRECOMPOSED**
- Always use precomposed characters, that is, characters in which a base character and a nonspacing character have a single character value. This is the default translation option.
- Cannot be used with COMPOSITE.

**COMPOSITE**
- Always use composite characters, that is, characters in which a base character and a nonspacing character have different character values. Cannot be used with PRECOMPOSED.

**ERR_INVALID_CHARS**
- If the function encounters an invalid input character, it fails and returns "1113".

**USEGLYPHCHARS**
- Use glyph characters instead of control characters.

A composite character consists of a base character and a nonspacing character, each having different character values. A precomposed character has a single character value for a base-nonspacing character combination. In the character è, the "e" is the base character and the "grave" accent mark is the nonspacing character. The function's default behavior is to translate to the precomposed form. If a precomposed form does not exist, the function attempts to translate to a composite form.

The flags PRECOMPOSED and COMPOSITE are mutually exclusive. The USEGLYPHCHARS flag and the ERR_INVALID_CHARS can be set regardless of the state of the other flags.

**outstem**
- The name of the stem variable that will contain the converted result. If the conversion was successful the stem will be composed of the following value(s):

```
outstem.!TEXT
```
- This variable will contain the converted string.

**Return codes:**

0
- No errors.

Other
- An error occurred. A *Windows System Error code* is returned. This may be one of the following, but could be others.

87
- Incorrect code page or codepage value.

1004
- Invalid translate flags.

1113
- No mapping for the Unicode character exists in the target code page.

**8.66. SysUtilVersion**
SysUtilVersion()

Returns a version number that identifies the current level of the Rexx Utilities package.

Return code: The RexxUtil version number in the format n.m.l.

8.67. SysVersion

SysVersion()

Returns a string to identify the operating system and version. The first word of the returned string contains the identifier for the operating system and the rest of the string contains an operating system specific version string. Rexx code should not rely on the exact format returned.

Possible output may be:

Example 8.46. RexxUtil — SysVersion

```
-- CentOS 7
say SysVersion() -- Linux 3.10.0-957.10.1.e17.x86_64
-- Ubuntu 16.04
say SysVersion() -- Linux 4.4.0-154-generic
-- macOS 10.13
say SysVersion() -- Darwin 18.7.0
-- Raspbian
say SysVersion() -- Linux 4.14.70-v7+
-- SLES/390
say SysVersion() -- Linux 3.12.74-60.64.40-default
-- Windows 10
say SysVersion() -- Windows 10.0.18363
```

See also functions SysWinVer (Windows only) and SysLinVer (Linux Only).

8.68. SysVolumeLabel (Windows only)

SysVolumeLabel(drive)

Returns the volume label for the specified drive.

Parameter:

```
drive
```

A drive specification like d: \ or \share\path. If omitted, the current drive is used.

8.69. SysWait (Unix-like systems only)

SysWait()

Waits for all child processes to end.

Returns:
SysWaitNamedPipe (Windows only)

The exit code from the child process.

Example:

SysFork (Unix-like systems only) has an example that uses wait().

8.70. SysWaitNamedPipe (Windows only)

```c
SysWaitNamedPipe(name, timeout)
```

Performs a timed wait on a named pipe and returns the WaitNamedPipe return code.

Parameters:

- `name`: The name of the pipe in the form "\servername\pipe\pipename."
- `timeout`: The number of microseconds to be waited. If you omit `timeout` or specify 0, SysWaitNamedPipe uses the default timeout value. To wait until the pipe is no longer busy, you can use a value of -1.

Returns 0 on success. A Windows System Error code is returned on error.

8.71. SysWinDecryptFile (Windows only)

```c
SysWinDecryptFile(filename)
```

Decrypts a given file.

Parameter:

- `filename`: The file to be decrypted.

Return codes:

0
Decryption was successful.

Other
A Windows System Error code. This may be one of the following, but could be others.

2
File not found.

4
Cannot open file.

5
Access denied.

82
Cannot decrypt.
8.72. SysWinEncryptFile (Windows only)

```
| SysWinEncryptFile( filename ) |
```

Encrypts a given file.

**Parameter:**

* filename
  The file to be encrypted.

**Return codes:**

0
  Encryption was successful.

Other
  A *Windows System Error code*. This may be one of the following, but could be others.

2
  File not found.

4
  Cannot open file.

5
  Access denied.

82
  Cannot encrypt.

8.73. SysWinGetDefaultPrinter (Windows only)

```
| SysWinGetDefaultPrinter() |
```

Returns the current default printer in the form “Printername,Drivername,Portname”.

8.74. SysWinGetPrinters (Windows only)

```
| SysWinGetPrinters( stemarray ) |
```

Returns a list of available printer descriptions.

**Parameters:**

* stemarray
  A Rexx stem or array variable for the returned printers. If a stem variable, `stem . 0` is set to `n`, the number of printers returned, and the printers are returned in `stem . 1` to `stem.n`.

Each entry is of the form “Printername,Drivername,Portname”.

**Return codes:**
SysWinSetDefaultPrinter (Windows only)

Sets the default printer.

Parameter:

description
A string identifying the printer. This can be either just the printer name, or be of the form “Printername,Drivername,Portname”.

Note
Either form of the description string is accepted. However, using just the printer name is the preferred method. Using the “Printername,Drivername,Portname” form will invoke an outdated Windows API that Microsoft has deprecated.

Return codes:

0
Success

non-zero
A Windows System Error code. You can use SysGetErrorText to get a description of the error.

Example 8.47. RexxUtil — SysWinGetPrinters and SysWinGetDefaultPrinter

```rexx
/* set default printer */
default = SysWinGetDefaultPrinter() parse var default default 
say 'The Default printer is:' default 
say 
if SysWinGetPrinters(list.) == 0 then do 
say "List of available printers (* = default):"
do i=1 to list.0 
parse var list.i pname 
if pname == default then 
say i list.i "*"
else
say i list.i 
end say 
say "Please enter number of new default printer (0 = keep default)"
```
pull i

numberOk = .false

if i~datatype('W') then do
  if 0 <= i & i <= list.0 then do
    numberOK = .true
    if i > 0 then do
      parse var list.i pname",".
      ret = SysWinSetDefaultPrinter(pname)
      if ret <> 0 then do
        say "Error setting default printer ("ret"):" SysGetErrorText(ret)
      end
      else do
        say "The new default printer is:" pname
      end
    end
  end
end

if \ numberOk then do
  say "You did not enter a valid printer number."
end
else do
  say "Failed to get a list of the available printers."
end

8.76. SysWinVer (Windows only)

`SysWinVer()`

Returns a string specifying the Windows operating system version information. Rexx code should not rely on the exact format returned.

Possible output might be:

Example 8.48. RexxUtil — SysWinVer

```
-- Windows 7
say SysWinVer() -- Windows 6.01.7601
-- Windows 10
say SysWinVer() -- Windows 10.0.18363
```

See also functions `SysVersion` and `SysLinVer (Linux Only).`
Chapter 9.

Parsing

The parsing instructions are ARG, PARSE, and PULL.

The data to be parsed is a source string. Parsing splits the data in a source string and assigns pieces of it to the variables named in a template. A template is a model specifying how to split the source string. The simplest kind of template consists of a list of variable names. Here is an example:

| variable1 variable2 variable3 |

This kind of template parses the source string into whitespace-delimited words. More complicated templates contain patterns in addition to variable names:

String patterns
Match the characters in the source string to specify where it is to be split. (See Section 9.2, “Templates Containing String Patterns” for details.)

Positional patterns
Indicate the character positions at which the source string is to be split. (See Section 9.3, “Templates Containing Positional (Numeric) Patterns” for details.)

Parsing is essentially a two-step process:
1. Parse the source string into appropriate substrings using patterns.
2. Parse each substring into words.

9.1. Simple Templates for Parsing into Words

Here is a parsing instruction:

Example 9.1. Parsing templates

parse value "time and tide" with var1 var2 var3

The template in this instruction is: var1 var2 var3. The data to be parsed is between the keywords PARSE VALUE and the keyword WITH, the source string time and tide. Parsing divides the source string into whitespace-delimited words and assigns them to the variables named in the template as follows:

| var1="time" |
| var2="and" |
| var3="tide" |

In this example, the source string to be parsed is a literal string, time and tide. In the next example, the source string is a variable.

Example 9.2. Parse value

/* PARSE VALUE using a variable as the source string to parse */
strings="time and tide"
parse value string with var1 var2 var3 /* same results */
PARSE VALUE does not convert lowercase a-z in the source string to uppercase A-Z. If you want to convert characters to uppercase, use PARSE UPPER VALUE. See Section 9.5, “Using UPPER, LOWER, and CASELESS” for a summary of the effect of parsing instructions on the case.

Note that if you specify the CASELESS option on a PARSE instruction, the string comparisons during the scanning operation are made independently of the alphabetic case. That is, a letter in uppercase is equal to the same letter in lowercase.

All of the parsing instructions assign the parts of a source string to the variables named in a template. There are various parsing instructions because of the differences in the nature or origin of source strings. For a summary of all the parsing instructions, see Section 9.6, “Parsing Instructions Summary”.

The PARSE VAR instruction is similar to PARSE VALUE except that the source string to be parsed is always a variable. In PARSE VAR, the name of the variable containing the source string follows the keywords PARSE VAR. In the next example, the variable stars contains the source string. The template is star1 star2 star3.

Example 9.3. Parse var

```plaintext
/* PARSE VAR example                                             */
stars="Sirius Polaris Rigil"
parse var stars star1 star2 star3 /* star1="Sirius" */ /* star2="Polaris" */ /* star3="Rigil" */
```

All variables in a template receive new values. If there are more variables in the template than words in the source string, the leftover variables receive null (empty) values. This is true for the entire parsing: for parsing into words with simple templates and for parsing with templates containing patterns. Here is an example of parsing into words:

```plaintext
/* More variables in template than (words in) the source string */
satellite="moon"
parse var satellite Earth Mercury /* Earth="moon" */ /* Mercury="" */
```

If there are more words in the source string than variables in the template, the last variable in the template receives all leftover data. Here is an example:

```plaintext
/* More (words in the) source string than variables in template */
satellites="moon Io Europa Callisto..."
parse var satellites Earth Jupiter /* Earth="moon" */ /* Jupiter="Io Europa Callisto..." */
```

Parsing into words removes leading and trailing whitespace characters from each word before it is assigned to a variable. The exception to this is the word or group of words assigned to the last variable. The last variable in a template receives leftover data, preserving extra leading and trailing whitespace characters. Here is an example:

Example 9.4. Parse var

```plaintext
/* Preserving extra blanks                                       */
solar5="Mercury Venus Earth Mars Jupiter " /*
parse var solar5 var1 var2 var3 var4 /* var1 ="Mercury" */ /* var2 ="Venus" */
```
In the source string, Earth has two leading blanks. Parsing removes both of them (the word-separator blank and the extra blank) before assigning var3="Earth". Mars has three leading blanks. Parsing removes one word-separator blank and keeps the other two leading blanks. It also keeps all five blanks between Mars and Jupiter and both trailing blanks after Jupiter.

Parsing removes no whitespace characters if the template contains only one variable. For example:

```
parse value "   Pluto   " with var1        /* var1="   Pluto   "*/
```

9.1.1. Message Term Assignments

In addition to assigning values to variables, the PARSE instruction also allows any message term value that can be used on the left side of an assignment instruction (See Section 1.13, "Assignments and Symbols"). For example:

Example 9.5. Parse with Message Terms Assignments

```
string = "1 one two 3 4"
a = .Array~new
d = .Directory~new
parse var string n a[n] a[n + 1] d~three d~four
-- a[1] = one
-- a[2] = two
-- d~three = 3
-- d~four = 4
```

9.1.2. The Period as a Placeholder

A period in a template is a placeholder. It is used instead of a variable name, but it receives no data. It is useful as a "dummy variable" in a list of variables or to collect unwanted information at the end of a string. And it saves the overhead of unneeded variables.

The period in the first example is a placeholder. Be sure to separate adjacent periods with whitespace; otherwise, an error results.

Example 9.6. Period placeholder

```
/* Period as a placeholder                                       */
stars="Arcturus Betelgeuse Sirius Rigil"
parse var stars . . brightest .            /* brightest="Sirius" */
/* Alternative to period as placeholder                          */
stars="Arcturus Betelgeuse Sirius Rigil"
parse var stars drop junk brightest rest   /* brightest="Sirius" */
```

9.2. Templates Containing String Patterns
A string pattern matches characters in the source string to indicate where to split it. A string pattern can be either of the following:

Literal string pattern
One or more characters within quotation marks.

Variable string pattern
A variable within parentheses with no plus (+), minus (-), or equal sign (=) before the left parenthesis. (See Section 9.4, “Parsing with Variable Patterns” for details.)

Here are two templates, a simple template and a template containing a literal string pattern:

```
var1 var2          /* simple template                            */
var1 ", " var2     /* template with literal string pattern       */
```

The literal string pattern is ", ". This template puts characters:

• From the start of the source string up to (but not including) the first character of the match (the comma) into var1
• Starting with the character after the last character of the match (the character after the blank that follows the comma) and ending with the end of the string into var2

A template with a string pattern can omit some of the data in a source string when assigning data to variables. The next two examples contrast simple templates with templates containing literal string patterns.

**Example 9.7. Template string patterns**

```
/* Simple template                                               */
name="Smith, John"
parse var name ln fn                                            /* Assigns: ln="Smith," */
/* fn="John" */
```

Notice that the comma remains (the variable ln contains "Smith, "). In the next example the template is ln ", " fn. This removes the comma.

**Example 9.8. Template string patterns**

```
/* Template with literal string pattern                          */
name="Smith, John"
parse var name ln ", " fn                                       /* Assigns: ln="Smith" */
/* fn="John" */
```

First, the language processor scans the source string for ", ". It splits the source string at that point. The variable ln receives data starting with the first character of the source string and ending with the last character before the match. The variable fn receives data starting with the first character after the match and ending with the end of string.

A template with a string pattern omits data in the source string that matches the pattern. (There is a special case (see Section 9.8.2, “Combining String and Positional Patterns”) in which a template with
Templates Containing Positional (Numeric) Patterns

9.3. Templates Containing Positional (Numeric) Patterns

A positional pattern is a number that identifies the character position at which the data in the source string is to be split. The number must be a whole number.

An absolute positional pattern is:

- A number with no plus (+) or minus (-) sign preceding it or with an equal sign (=) preceding it.
- An expression in parentheses with an equal sign before the left parenthesis. (See Section 9.4, “Parsing with Variable Patterns” for details on variable positional patterns.)

The number specifies the absolute character position at which the source string is to be split.

Here is a template with absolute positional patterns:

```
variable1 11 variable2 21 variable3
```

The numbers 11 and 21 are absolute positional patterns. The number 11 refers to the 11th position in the input string, 21 to the 21st position. This template puts characters:

- 1 through 10 of the source string into variable1
- 11 through 20 into variable2
- 21 to the end into variable3

Positional patterns are probably most useful for working with a file of records, such as:

The following example uses this record structure:

```
/* Parsing with absolute positional patterns in template */
record.1="Clemens   Samuel    Mark Twain          
record.2="Evans     Mary Ann  George Eliot        
record.3="Munro     H.H.      Saki                
```
Templates Containing Positional (Numeric) Patterns

```plaintext
do n=1 to 3
  parse var record.n lastname 11 firstname 21 pseudonym
  If lastname="Evans" & firstname="Mary Ann" then say "By George!"
end                         /* Says "By George!" after record 2  */
```

The source string is split at character position 11 and at position 21. The language processor assigns characters 1 to 10 to `lastname`, characters 11 to 20 to `firstname`, and characters 21 to 40 to `pseudonym`.

The template could have been:

```plaintext
1 lastname 11 firstname 21 pseudonym
```

instead of

```plaintext
lastname 11 firstname 21 pseudonym
```

Specifying 1 is optional.

Optionally, you can put an equal sign before a number in a template. An equal sign is the same as no sign before a number in a template. The number refers to a particular character position in the source string. These two templates are equal:

```plaintext
lastname  11 first  21 pseudonym
```

A relative positional pattern is a number with a plus (+) or minus (-) sign preceding it. It can also be a variable within parentheses, with a plus (+) or minus (-) sign preceding the left parenthesis; for details see Section 9.4, “Parsing with Variable Patterns”.

The number specifies the relative character position at which the source string is to be split. The plus or minus indicates movement right or left, respectively, from the start of the string (for the first pattern) or from the position of the last match. The position of the last match is the first character of the last match. Here is the same example as for absolute positional patterns done with relative positional patterns:

**Example 9.10. Parsing with relative positional patterns**

```plaintext
/* Parsing with relative positional patterns in template         */
record.1="Clemens   Samuel    Mark Twain          
record.2="Evans     Mary Ann  George Eliot        
record.3="Munro     H.H.      Saki                
do n=1 to 3
  parse var record.n lastname +10 firstname + 10 pseudonym
  If lastname="Evans" & firstname="Mary Ann" then say "By George!"
end                                             /* same results  */
```

Whitespace characters between the sign and the number are insignificant. Therefore, +10 and +10 have the same meaning. Note that +0 is a valid relative positional pattern.

Absolute and relative positional patterns are interchangeable except in the special case (see Section 9.8.2, “Combining String and Positional Patterns”) when a string pattern precedes a variable pattern.
name and a positional pattern follows the variable name. The templates from the examples of absolute and relative positional patterns give the same results.

<table>
<thead>
<tr>
<th>(Implied starting point is position 1)</th>
<th>Put characters 1 through 10 in lastname.</th>
<th>Put characters 1 through 10 in firstname.</th>
<th>Put characters 21 through end of string in pseudonym.</th>
</tr>
</thead>
<tbody>
<tr>
<td>lastname 11</td>
<td>(non-inclusive stopping point is 11 (1 + 10))</td>
<td>(non-inclusive stopping point is 21 (11 + 10))</td>
<td>pseudonym pseudonym</td>
</tr>
</tbody>
</table>

With positional patterns, a matching operation can back up to an earlier position in the source string. Here is an example using absolute positional patterns:

```/* Backing up to an earlier position (with absolute positional) */
string="astronomers"
parse var string 2 var1 4 1 var2 2 4 var3 5 11 var4
say string "study" var1||var2||var3||var4
/* Displays: "astronomers study stars" */```

The absolute positional pattern 1 backs up to the first character in the source string.

With relative positional patterns, a number preceded by a minus sign backs up to an earlier position. Here is the same example using relative positional patterns:

```/* Backing up to an earlier position (with relative positional) */
string="astronomers"
parse var string 2 var1 +2 -3 var2 +1 +2 var3 +1 +6 var4
say string "study" var1||var2||var3||var4 /* same results */```

In the previous example, the relative positional pattern -3 backs up to the first character in the source string.

The templates in the previous two examples are equivalent.

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
<th>4</th>
<th>3</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Start at 2.</td>
<td>Go to 1 (4-3=1).</td>
<td>Non-inclusive stopping point is 4 (2 + 2 = 4).</td>
<td>Non-inclusive stopping point is 2 (1+1=2).</td>
<td>Go to 4 (2+2=4).</td>
<td>Go to 11 (5+6=11).</td>
</tr>
<tr>
<td>var1  var2 +2</td>
<td>var2  var2 +1</td>
<td>var3  var3 +1</td>
<td>var4  var4 +6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You can use templates with positional patterns to make several assignments:

```/* Making several assignments */
books="Silas Marner, Felix Holt, Daniel Deronda, Middlemarch"
parse var books 1 Eliot 1 Evans
/* Assigns the (entire) value of books to Eliot and to Evans. */```

A length positional pattern is a number with a (> or (< preceding it. It can also be an expression within parentheses, with a (> or (< preceding the left parenthesis; for details see Section 9.4, “Parsing with Variable Patterns”.)
The number specifies the length at which the source string is to be split, relative to the current position. The > or < indicates movement right or left, respectively, from the start of the string (for the first pattern) or from the position of the last match. The position of the last match is the first character of the last match. Here is the same example as for relative positional patterns done with length positional patterns:

**Example 9.11. Parsing with relative positional patterns**

```plaintext
/* Parsing with relative positional patterns in template */
record.1="Clemens   Samuel    Mark Twain          
record.2="Evans     Mary Ann  George Eliot        
record.3="Munro     H.H.      Saki                
do n=1 to 3
  parse var record.n lastname >10 firstname >10 pseudonym
  If lastname="Evans" & firstname="Mary Ann" then say "By George!"
end                                              /* same results */
```

Whitespace characters between the trigger and the number are insignificant. Therefore, >10 and >10 have the same meaning. Note that >0 <0 and are valid length positional pattern.

The > length pattern and the + relative positional pattern are interchangeable except in the special case of the value 0. A >0 pattern will split the string into a null string and leave the match position unchanged. This is particularly useful for parsing off length-qualified fields from a string.

**Example 9.12. Parsing with length patterns**

```plaintext
/* Parsing with length patterns in template */
line = "04Mark0005Twain"
paste var line len +2 first >(len) len +2 middle >(len) len +2 last >(len)
say '"first" '"middle" '"last"' -- displays "Mark" "" "Twain"
/* parsing with relative patterns only */
paste var line len +2 first +(len) len +2 middle +(len) len +2 last +(len)
say '"first" '"middle" '"last"' -- displays "Mark" "05Twain" "Twain"
```

The < length pattern will move the position the indicated position to the left, and split the string between the original position and the movement position. At of the operation, the current position is returned to the original position. This movement is equivalent to using a negative relative pattern followed by a positive relative pattern for the same length. This operation allows for easy extraction of characters that precede a string match.

**Example 9.13. Parsing with length patterns**

```plaintext
/* Parsing with length patterns in template */
paste value '12345.6789' with '.' digit <1 -- digit -> "5"
/* parsing with relative patterns only */
paste value '12345.6789' with '.' -1 digit +1 -- digit -> "5"
```

**9.3.1. Combining Patterns and Parsing into Words**

If a template contains patterns that divide the source string into sections containing several words, string and positional patterns divide the source string into substrings. The language processor then applies a section of the template to each substring, following the rules for parsing into words.
Example 9.14. Combining string patterns

```/* Combining string pattern and parsing into words */
name="    John      Q.   Public"
parse var name fn init "." ln /* Assigns: fn="John" */
/* init="     Q" */
/* ln="   Public" */
```

The pattern divides the template into two sections:
- fn init
- ln

The matching pattern splits the source string into two substrings:
- "    John      Q"
- "   Public"

The language processor parses these substrings into words based on the appropriate template section.

John has three leading blanks. All are removed because parsing into words removes leading and trailing blanks except from the last variable.

Q has six leading blanks. Parsing removes one word-separator blank and keeps the rest because init is the last variable in that section of the template.

For the substring "   Public", parsing assigns the entire string into ln without removing any blanks. This is because ln is the only variable in this section of the template. (For details about treatment of whitespace characters, see Section 9.1, “Simple Templates for Parsing into Words”.)

Example 9.15. Combining positional patterns

```/* Combining positional patterns with parsing into words */
string="R E X X"
parse var string var1 var2 4 var3 6 var4 /* Assigns: var1="R" */
/* var2="E" */
/* var3=" X" */
/* var4=" X" */
```

The pattern divides the template into three sections:
- var1 var2
- var3
- var4

The matching patterns split the source string into three substrings that are individually parsed into words:
- "R E"
- " X"
- " X"

The variable var1 receives "R"; var2 receives "E". Both var3 and var4 receive " X" (with a blank before the X) because each is the only variable in its section of the template. (For details on treatment of whitespace characters, see Section 9.1, “Simple Templates for Parsing into Words”.)
9.4. Parsing with Variable Patterns

You might want to specify a pattern by using the value of a variable or expression instead of a fixed string or number. You do this by placing an expression in parentheses. This is a variable reference. Whitespace characters are not necessary inside or outside the parentheses, but you can add them if you wish.

The template in the next parsing instruction contains the following literal string pattern ".  ".

```
parse var name fn init ".  " ln
```

Here is how to specify that pattern as a variable string pattern:

```
strngptrn="  
parse var name fn init (strngptrn) ln
```

If no equal, plus sign, minus sign, >, or < precedes the parenthesis that is before the variable name, the character string value of the variable is then treated as a string pattern. The expression can reference variables that have been set earlier in the same template.

**Example 9.16. Combining variables in patterns**

```c
/* Using a variable as a string pattern                          */
/*  The variable (delim) is set in the same template             */
SAY "Enter a date (mm/dd/yy format). =====> " /* assume 11/15/98 */
pull date
parse var date month 3 delim +1 day +2 (delim) year
/* Sets: month="11"; delim="/"; day="15"; year="98"  */
```

If an equal, a plus, a minus sign, > or < precedes the left parenthesis, the value of the expression is treated as an absolute, relative positional, or length positional pattern. The value of the expression must be a positive whole number or zero.

The expression can reference variables that have been set earlier in the same template. In the following example, the first two fields specify the starting-character positions of the last two fields.

**Example 9.17. Combining variables and positional patterns**

```c
/* Using a variable as a positional pattern                      */
datailine = "12 26 .....Samuel ClemensMark Twain"
parse var datailine pos1 pos2 6 =(pos1) realname =(pos2) pseudonym
/* Assigns: realname="Samuel Clemens"; pseudonym="Mark Twain" */
```

The positional pattern 6 is needed in the template for the following reason: Word parsing occurs after the language processor divides the source string into substrings using patterns. Therefore, the positional pattern =pos1 cannot be correctly interpreted as =12 until after the language processor has split the string at column 6 and assigned the whitespace-delimited words 12 and 26 to pos1 and pos2, respectively.

9.5. Using UPPER, LOWER, and CASELESS
Specifying UPPER on any of the PARSE instructions converts lowercase a-z to uppercase A-Z before parsing.

The ARG instruction is a short form of PARSE UPPER ARG. The PULL instruction is a short form of PARSE UPPER PULL. If you do not desire uppercase translation, use PARSE ARG instead of ARG or PARSE UPPER ARG, and PARSE PULL instead of PULL or PARSE UPPER PULL.

Specifying LOWER on any of the PARSE instructions converts uppercase A-Z to lowercase a-z before parsing.

Specifying CASELESS means the comparisons during parsing are independent of the case—that is, a letter in uppercase is equal to the same letter in lowercase.

### 9.6. Parsing Instructions Summary

All parsing instructions assign parts of the source string to the variables named in the template. The following table summarizes where the source string comes from.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Where the source string comes from</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARG</td>
<td>Arguments you list when you call the program or arguments in the call to a subroutine or function.</td>
</tr>
<tr>
<td>PARSE ARG</td>
<td>Arguments you list when you call the program or arguments in the call to a subroutine or function.</td>
</tr>
<tr>
<td>PARSE LINEIN</td>
<td>Next line in the default input stream.</td>
</tr>
<tr>
<td>PULL</td>
<td>The string at the head of the external data queue. (If the queue is empty, it uses default input, typically the terminal.)</td>
</tr>
<tr>
<td>PARSE PULL</td>
<td>The string at the head of the external data queue. (If the queue is empty, it uses default input, typically the terminal.)</td>
</tr>
<tr>
<td>PARSE SOURCE</td>
<td>System-supplied string giving information about the executing program.</td>
</tr>
<tr>
<td>PARSE VALUE</td>
<td>Expression between the keywords VALUE and WITH in the instruction.</td>
</tr>
<tr>
<td>PARSE VAR name</td>
<td>Parses the value of name.</td>
</tr>
<tr>
<td>PARSE VERSION</td>
<td>System-supplied string specifying the language, language level, and (three-word) date.</td>
</tr>
</tbody>
</table>

### 9.7. Parsing Instructions Examples

All examples in this section parse source strings into words.

ARG

**Example 9.18. ARG with source string named in Rexx program invocation**

```rexx
/* ARG with source string named in Rexx program invocation */
/* Program name is PALETTE. Specify 2 primary colors (yellow, */
/* red, blue) on call. Assume call is: palette red blue */
arg var1 var2
  /* Assigns: var1="RED"; var2="BLUE" */
  If var1<>"RED" & var1<>"YELLOW" & var1<>"BLUE" then signal err
  If var2<>"RED" & var2<>"YELLOW" & var2<>"BLUE" then signal err
  total=length(var1)+length(var2)
```
ARG converts alphabetic characters to uppercase before parsing. An example of ARG with the arguments in the CALL to a subroutine is in Section 9.8.1, “Parsing Several Strings”.

PARSE ARG is similar to ARG except that PARSE ARG does not convert alphabetic characters to uppercase before parsing.

PARSE LINEIN

Example 9.19. PARSE LINEIN

```plaintext
parse linein "a" num1 "c" num2 /* Assume: 8 and 9 */
sum=num1+num2 /* Enter: a=8 b=9 as input */
say sum /* Displays: "17" */
```

PARSE PULL

Example 9.20. PARSE PULL

```plaintext
PUSH "80 7" /* Puts data on queue */
parse pull fourscore seven /* Assigns: fourscore="80"; seven="7" */
SAY fourscore+seven /* Displays: "87" */
```

PARSE SOURCE

Example 9.21. PARSE SOURCE

```plaintext
parse source sysname .
Say sysname /* Possibly Displays: */
/* "Windows" */
```

PARSE VAR examples are throughout the chapter, starting with Chapter 9, Parsing.

PARSE VERSION

Example 9.22. PARSE VERSION

```plaintext
parse version . level .
say level /* Displays: "6.02" */
```

PULL is similar to PARSE PULL except that PULL converts alphabetic characters to uppercase before parsing.
9.8. Advanced Topics in Parsing

This section includes parsing several strings and flow charts illustrating a conceptual view of parsing.

9.8.1. Parsing Several Strings

Only ARG and PARSE ARG can have more than one source string. To parse several strings, you can specify several comma-separated templates. Here is an example:

```
parse arg template1, template2, template3
```

This instruction consists of the keywords PARSE ARG and three comma-separated templates. For an ARG instruction, the source strings to be parsed come from arguments you specify when you call a program or CALL a subroutine or function. Each comma is an instruction to the parser to move on to the next string.

Example:

```
Example 9.23. Parsing variable strings

/* Parsing several strings in a subroutine                       */
um="3"
musketeers="Porthos Athos Aramis D'Artagnan"
CALL Sub num, musketeers /* Passes num and musketeers to sub       */
SAY total; say fourth /* Displays: "4" and "D'Artagnan" */
EXIT

Sub:
parse arg subtotal, ... fourth
total=subtotal+1
RETURN
```

Note that when a Rexx program is started as a command, only one argument string is recognized. You can pass several argument strings for parsing if:

- One Rexx program calls another Rexx program with the CALL instruction or a function call
- Programs written in other languages start a Rexx program

If there are more templates than source strings, each variable in a leftover template receives a null string. If there are more source strings than templates, the language processor ignores leftover source strings. If a template is empty (two subsequent commas) or contains no variable names, parsing proceeds to the next template and source string.

9.8.2. Combining String and Positional Patterns

There is a special case in which absolute and relative positional patterns do not work identically. Parsing with a template containing a string pattern skips the data in the source string that matches the pattern (see Section 9.2, "Templates Containing String Patterns"). But a template containing the sequence string pattern, variable name, and relative position pattern does not skip the matching data.
A relative positional pattern moves relative to the first character matching a string pattern. As a result, assignment includes the data in the source string that matches the string pattern.

Example 9.24. Combining string and positional patterns

```/* Template containing string pattern, then variable name, then */
/* relative positional pattern does not skip any data. */
string="REstructured eXtended eXecutor"
parse var string var1 3 junk "X" var2 +1 junk "X" var3 +1 junk
say var1||var2||var3 /* Concatenates variables; displays: "REXX" */
```

Here is how this template works:

<table>
<thead>
<tr>
<th>var1 3</th>
<th>junk 'X'</th>
<th>var2 +1</th>
<th>junk 'X'</th>
<th>var3 +1</th>
<th>junk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put characters 1 through 2 in var1 (stopping point is 3).</td>
<td>Starting point at 3, put characters upto (not including) first 'X' in junk.</td>
<td>Starting with first 'X' put 1 (+1) character in var2.</td>
<td>Starting with second 'X' put 1 (+1) character in var3.</td>
<td>Starting with second 'X' put rest in junk.</td>
<td></td>
</tr>
<tr>
<td>var1='RE' junk='structured e'</td>
<td>var2=X' junk='tended e'</td>
<td>var3='X' junk='ecutor'</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9.8.3. Conceptual Overview of Parsing

The following figures are to help you understand the concept of parsing.

The figures include the following terms:

string start
- is the beginning of the source string (or substring).

string end
- is the end of the source string (or substring).

length
- is the length of the source string.

match start
- is in the source string and is the first character of the match.

match end
- is in the source string. For a string pattern, it is the first character after the end of the match. For a positional pattern, it is the same as match start.

match position
- is in the source string. For a string pattern, it is the first matching character. For a positional pattern, it is the position of the matching character.
token
is a distinct syntactic element in a template, such as a variable, a period, a pattern, or a comma.

value
is the numeric value of a positional pattern. This can be either a constant or the resolved value of a variable.

Figure 9.1. Conceptual Overview of Parsing
Figure 9.2. Conceptual View of Finding Next Pattern
Figure 9.3. Conceptual View of Word Parsing

Note

The figures do not include error cases.
Chapter 10.

Numbers and Arithmetic

This chapter gives an overview of the arithmetic facilities of the Rexx language.

Numbers can be expressed flexibly. Leading and trailing whitespace characters are permitted, and exponential notation can be used. Valid numbers are, for example:

Example 10.1. Numbers

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>a whole number</td>
</tr>
<tr>
<td>&quot;-76&quot;</td>
<td>a signed whole number</td>
</tr>
<tr>
<td>12.76</td>
<td>decimal places</td>
</tr>
<tr>
<td>&quot; + 0.003 &quot;</td>
<td>blanks around the sign and so forth</td>
</tr>
<tr>
<td>17.</td>
<td>same as 17</td>
</tr>
<tr>
<td>.5</td>
<td>same as 0.5</td>
</tr>
<tr>
<td>4E9</td>
<td>exponential notation</td>
</tr>
<tr>
<td>0.73e-7</td>
<td>exponential notation</td>
</tr>
</tbody>
</table>

Ignoring Exponential Notation, a number in Rexx is defined as follows:

- **whitespace** are one or more blanks or horizontal tab characters.
- **sign** is either + or -.
- **digits** are one or more of the decimal digits 0-9.

Note that a single period alone is not a valid number.

The arithmetic operators include addition (+), subtraction (-), multiplication (*), power (**), division (/), prefix plus (+), and prefix minus (-). In addition, it includes integer divide (%), which divides and returns the integer part, and remainder (//), which divides and returns the remainder. For examples of the arithmetic operators, see Section 10.2.4, "Operator Examples".

The result of an arithmetic operation is formatted as a character string according to specific rules. The most important rules are:

- Results are calculated up to a maximum number of significant digits. The default is 9, but can be overridden on a source-file basis with the ::OPTIONS directive. The default setting can be altered with the NUMERIC DIGITS instruction. Thus, with NUMERIC DIGITS 9, if a result requires more than 9 digits, it is rounded to 9 digits. For example, the division of 2 by 3 results in 0.666666667.
- Except for division and power, trailing zeros are preserved. For example:
Example 10.2. Arithmetic

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.40 + 2</td>
<td>4.40</td>
</tr>
<tr>
<td>2.40 - 2</td>
<td>0.40</td>
</tr>
<tr>
<td>2.40 * 2</td>
<td>4.80</td>
</tr>
<tr>
<td>2.40 / 2</td>
<td>1.2</td>
</tr>
</tbody>
</table>

If necessary, you can remove trailing zeros with the **String strip** method, the **STRIP** built-in function, or by division by 1.

- A zero result is always expressed as the single digit 0.
- Exponential form is used for a result depending on its value and the setting of NUMERIC DIGITS. If the number of places needed before the decimal point exceeds the NUMERIC DIGITS setting, or the number of places after the point exceeds twice the NUMERIC DIGITS setting, the number is expressed in exponential notation:

Example 10.3. Arithmetic

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e6 * 1e6</td>
<td>1E+12</td>
</tr>
<tr>
<td>1 / 3E10</td>
<td>3.3333333E-11</td>
</tr>
</tbody>
</table>

10.1. Precision

Precision is the maximum number of significant digits that can result from an operation. This is controlled by the instruction:

```plaintext
NUMERIC DIGITS expression
```

The expression is evaluated and must result in a positive whole number. This defines the precision (number of significant digits) of a calculation. Results are rounded to that precision, if necessary.

If you do not specify expression in this instruction, or if no NUMERIC DIGITS instruction has been processed since the start of a program, the default precision is used. The Rexx standard for the default precision is 9. The default may be overridden on a source-file basis using the **::OPTIONS** directive.

NUMERIC DIGITS can set values smaller than nine. However, use small values with care because the loss of precision and rounding affects all Rexx computations, including, for example, the computation of new values for the control variable in DO loops.

10.2. Arithmetic Operators

Rexx arithmetic is performed by the operators +, - , * , / , % , // , and ** (add, subtract, multiply, divide, integer divide, remainder, and power).

Before every arithmetic operation, the terms operated upon have leading zeros removed (noting the position of any decimal point, and leaving only one zero if all the digits in the number are zeros). They are then truncated, if necessary, to DIGITS + 1 significant digits before being used in the computation. The extra digit improves accuracy because it is inspected at the end of an operation, when a number
is rounded to the required precision. When a number is truncated, the LOSTDIGITS condition is raised if a SIGNAL ON LOSTDIGITS condition trap is active. The operation is then carried out under up to double that precision. When the operation is completed, the result is rounded, if necessary, to the precision specified by the NUMERIC DIGITS instruction.

The values are rounded as follows: 5 through 9 are rounded up, and 0 through 4 are rounded down.

10.2.1. Power

The ** (power) operator raises a number to a power, which can be positive, negative, or 0. The power must be a whole number. The second term in the operation must be a whole number and is rounded to DIGITS digits, if necessary, as described in Section 10.5, "Limits and Errors when Rexx Uses Numbers Directly". If negative, the absolute value of the power is used, and the result is inverted (that is, the number 1 is divided by the result). For calculating the power, the number is multiplied by itself for the number of times expressed by the power. Trailing zeros are then removed as though the result were divided by 1.

10.2.2. Integer Division

The % (integer divide) operator divides two numbers and returns the integer part of the result. The result is calculated by repeatedly subtracting the divisor from the dividend as long as the dividend is larger than the divisor. During this subtraction, the absolute values of both the dividend and the divisor are used: the sign of the final result is the same as that which would result from regular division.

If the result cannot be expressed as a whole number, the operation is in error and fails—that is, the result must not have more digits than the current setting of NUMERIC DIGITS. For example, **10000000000%3** requires 10 digits for the result (3333333333) and would, therefore, fail if NUMERIC DIGITS 9 were in effect.

10.2.3. Remainder

The // (remainder) operator returns the remainder from an integer division and is defined to be the residue of the dividend after integer division. The sign of the remainder, if nonzero, is the same as that of the original dividend.

This operation fails under the same conditions as integer division, that is, if integer division on the same two terms fails, the remainder cannot be calculated.

10.2.4. Operator Examples

Example 10.4. Operators

```
/* With:  NUMERIC DIGITS 5 */
12+7.00    ->  19.00
1.3-1.07   ->  0.23
1.3-2.07   -> -0.77
1.20*3     ->   3.60
7*3        ->   21
0.9*0.8    ->   0.72
1/3        ->  0.33333
2/3        ->  0.66667
```
Exponential Notation

5/2 -> 2.5
1/10 -> 0.1
12/12 -> 1
8.0/2 -> 4
2**3  -> 8
2**-3 -> 0.125
1.7**8 -> 69.758
2%3   -> 0
2.1/3 -> 2.1
10%3  -> 3
10//3 -> 1
-10//3 -> -1
10.2//1 -> 0.2
10//0.3 -> 0.1
3.6//1.3 -> 1.0

10.3. Exponential Notation

For both large and small numbers, an exponential notation can be useful. For example:

```
numeric digits 5
say 54321*54321
```

would display 2950800000 in the long form. Because this is misleading, the result is expressed as 2.9508E+9 instead.

The definition of numbers is, therefore, extended as follows:

The integer following the E represents a power of ten that is to be applied to the number. The E can be in uppercase or lowercase.

Certain character strings are numbers even though they do not appear to be numeric, such as 0E123 (0 times 10 raised to the power of 123) and 1E342 (1 times 10 raised to the power of 342). Also, a comparison such as 0E123=0E567 gives a true result of 1 (0 is equal to 0). To prevent problems when comparing nonnumeric strings, use the strict comparison operators.

Here are some examples:

```
Example 10.5. Exponential notation

12E7  =  120000000 /* Displays "1" */
12E-5 =  0.00012  /* Displays "1" */
```
The results of calculations are returned in either conventional or exponential form, depending on the setting of NUMERIC DIGITS. If the number of places needed before the decimal point exceeds DIGITS, or the number of places after the point exceeds twice DIGITS, the exponential form is used. The exponential form the language processor generates always has a sign following the E to improve readability. If the exponent is 0, the exponential part is omitted—that is, an exponential part of E+0 is not generated.

You can explicitly convert numbers to exponential form, or force them to be displayed in the long form, by using the FORMAT built-in function.

Scientific notation is a form of exponential notation that adjusts the power of ten so that the number contains only one nonzero digit before the decimal point. Engineering notation is a form of exponential notation in which up to three digits appear before the decimal point, and the power of ten is always a multiple of three. The integer part can, therefore, range from 1 through 999. You can control whether scientific or engineering notation is used with the following instruction:

Example 10.6. Scientific notation

```
/* after the instruction */
Numeric form scientific
123.45 * 1e11   ->   1.2345E+13
/* after the instruction */
Numeric form engineering
123.45 * 1e11   ->   12.345E+12
```

10.4. Numeric Comparisons

The comparison operators are listed in Section 1.11.2.3, "Comparison". You can use any of them for comparing numeric strings. However, you should not use ==, !=, <=, >=, >, <, >>, <<, and <<< for comparing numbers because leading and trailing whitespace characters and leading zeros are significant with these operators.

Numeric values are compared by subtracting the two numbers (calculating the difference) and then comparing the result with 0. That is, the operation:
where \( ? \) is any numeric comparison operator, is identical with:

\[
(A - Z) \ ? \ "0"
\]

It is, therefore, the difference between two numbers, when subtracted under Rexx subtraction rules, that determines their equality.

Fuzz affects the comparison of two numbers. It controls how much two numbers can differ and still be considered equal in a comparison. The FUZZ value is set by the following instruction:

\[
\text{NUMERIC FUZZ expression}
\]

expression must result in a positive whole number or zero. The default is 0.

Fuzz is to temporarily reduce the value of DIGITS. That is, the numbers are subtracted with a precision of DIGITS minus FUZZ digits during the comparison. The FUZZ setting must always be less than DIGITS.

If, for example, DIGITS = 9 and FUZZ = 1, the comparison is carried out to 8 significant digits, just as though \text{NUMERIC DIGITS 8} had been put in effect for the duration of the operation.

Example 10.7. Comparison

<table>
<thead>
<tr>
<th>Numeric digits 5</th>
<th>Numeric fuzz 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>say 4.9999 = 5</td>
<td>/* Displays &quot;0&quot; */</td>
</tr>
<tr>
<td>say 4.9999 &lt; 5</td>
<td>/* Displays &quot;1&quot; */</td>
</tr>
<tr>
<td>Numeric fuzz 1</td>
<td></td>
</tr>
<tr>
<td>say 4.9999 = 5</td>
<td>/* Displays &quot;1&quot; */</td>
</tr>
<tr>
<td>say 4.9999 &lt; 5</td>
<td>/* Displays &quot;0&quot; */</td>
</tr>
</tbody>
</table>

10.5. Limits and Errors when Rexx Uses Numbers Directly

When Rexx uses numbers directly, that is, numbers that have not been involved in an arithmetic operation, they are rounded, if necessary, according to the setting of NUMERIC DIGITS. The normal whole number limit depends on the default NUMERIC DIGITS setting. The default setting is 9, making the normal whole number limit 999999999.

The following table shows which numbers must be whole numbers and what their limits are:

<table>
<thead>
<tr>
<th>Table 10.1. Whole Number Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power values (right-hand operand of the power operator)</td>
</tr>
<tr>
<td>Values of expr and exprf in the DO instruction</td>
</tr>
<tr>
<td>Values given for DIGITS or FUZZ in the NUMERIC instruction</td>
</tr>
<tr>
<td>Positional patterns in parsing templates</td>
</tr>
<tr>
<td>Number given for option in the TRACE instruction</td>
</tr>
</tbody>
</table>
When Rexx uses numbers directly, the following types of errors can occur:

- **Overflow or underflow.**

  This error occurs if the exponential part of a result exceeds the range that the language processor can handle, when the result is formatted according to the current settings of NUMERIC DIGITS and NUMERIC FORM. The language defines a minimum capability for the exponential part, namely the largest number that can be expressed as an exact integer in default precision. Because the default precision is 9, you can use exponents in the range -999999999 through 999999999.

  Because this allows for (very) large exponents, overflow or underflow is treated as a syntax error.

- **Insufficient storage.**

  Storage is needed for calculations and intermediate results, and if an arithmetic operation fails because of lack of storage. This is considered as a terminating error.
Conditions and Condition Traps

A condition is an event or state that CALL ON or SIGNAL ON can trap. A condition trap can modify the flow of execution in a Rexx program. Condition traps are turned on or off using the ON or OFF subkeywords of the SIGNAL and CALL instructions.
condition, usercondition, and trapname are single symbols that are taken as constants. Following one of these instructions, a condition trap is set to either ON (enabled) or OFF (disabled). The initial setting for all condition traps is OFF.

If a condition trap is enabled and the specified condition or usercondition occurs, control passes to the routine or label trapname if you have specified trapname. Otherwise, control passes to the routine or label usercondition or condition. CALL or SIGNAL is used, depending on whether the most recent trap for the condition was set using CALL ON or SIGNAL ON, respectively.

If you use CALL, the trapname can be an internal label, a built-in function, or an external routine. When calling trapname, the current Condition Object is supplied as an argument.

If you use SIGNAL, the trapname can only be an internal label.

The conditions and their corresponding events that can be trapped are:

**ANY**
traps any condition that a more specific condition trap does not trap. For example, if NOVALUE is raised and there is no NOVALUE trap enabled, but there is a SIGNAL ON ANY trap, the ANY trap is called for the NOVALUE condition. For example, a CALL ON ANY trap is ignored if NOVALUE is raised because CALL ON NOVALUE is not allowed.

**ERROR**
raised if a command indicates an error condition upon return. It is also raised if any command indicates failure and none of the following is active:
- CALL ON FAILURE
- SIGNAL ON FAILURE
- CALL ON ANY
- SIGNAL ON ANY

The condition is raised at the end of the clause that called the command but is ignored if the ERROR condition trap is already in the delayed state. The delayed state is the state of a condition
trap when the condition has been raised but the trap has not yet been reset to the enabled (ON) or disabled (OFF) state.

If the package contains an \texttt{::OPTIONS ERROR SYNTAX} directive and ERROR is not trapped with \texttt{CALL/SIGNAL ON/OFF ERROR/ANY}, a SYNTAX condition is raised instead.

\textbf{FAILURE}  
raised if a command indicates a failure condition upon return. The condition is raised at the end of the clause that called the command but is ignored if the FAILURE condition trap is already in the delayed state.

An attempt to enter a command to an unknown subcommand environment also raises a FAILURE condition.

If the package contains an \texttt{::OPTIONS FAILURE SYNTAX} directive and FAILURE is not trapped with \texttt{CALL/SIGNAL ON/OFF FAILURE/ANY}, a SYNTAX condition is raised instead.

\textbf{HALT}  
raised if an external attempt is made to interrupt and end execution of the program. The condition is usually raised at the end of the clause that was processed when the external interruption occurred. When a Rexx program is running in a full-screen or command prompt session, the Ctrl+Break key combination raises the halt condition. However, if Ctrl+Break is pressed while a command or non-Rexx external function is processing, the command or function ends.

\textbf{Notes:}

1. Application programs that use the Rexx language processor might use the RXHLT exit or the RexxStart programming interface to halt the execution of a Rexx macro. (See the Open Object Rexx: Application Programming Interfaces for details about exits.)

2. Only SIGNAL ON HALT or CALL ON HALT can trap error 4, described in Appendix C, Error Numbers and Messages.

\textbf{LOSTDIGITS}  
raised if a number used in an arithmetic operation has more digits than the current setting of NUMERIC DIGITS. Leading zeros are not counted in this comparison. You can specify the LOSTDIGITS condition only for SIGNAL ON.

If the package contains an \texttt{::OPTIONS LOSTDIGITS SYNTAX} directive and LOSTDIGITS is not trapped with \texttt{SIGNAL ON/OFF LOSTDIGITS/ANY}, a SYNTAX condition is raised instead.

\textbf{NOMETHOD}  
raised if an object receives a message for which it has no method defined, and the object does not have an UNKNOWN method. You can specify the NOMETHOD condition only for SIGNAL ON.

\textbf{NOSTRING}  
raised when the language processor requires a string value from an object and the object does not directly provide a string value. See Section 4.2.11, "Required String Values" for more information. You can specify the NOSTRING condition only for SIGNAL ON.

If the package contains an \texttt{::OPTIONS NOSTRING SYNTAX} directive and NOSTRING is not trapped with \texttt{SIGNAL ON/OFF NOSTRING/ANY}, a SYNTAX condition is raised instead.

\textbf{NOTREADY}  
raised if an error occurs during an input or output operation. See Section 14.5, "Errors during Input and Output". This condition is ignored if the NOTREADY condition trap is already in the delayed state.
If the package contains an ::OPTIONS NOTREADY SYNTAX directive and NOTREADY is not trapped with CALL/SIGNAL ON/OFF NOTREADY/ANY, a SYNTAX condition is raised instead.

NOVALUE
raised if an uninitialized variable is used as:
• A term in an expression
• The name following the VAR subkeyword of a PARSE instruction
• A variable reference in a parsing template, an EXPOSE instruction, a PROCEDURE instruction, or a DROP instruction
• A method selection override specifier in a message term

Notes:
NOVALUE is not raised for any uninitialized variables in tails in compound variables.

If the package contains an ::OPTIONS NOVALUE SYNTAX directive and NOVALUE is not trapped with SIGNAL ON/OFF NOVALUE/ANY, a SYNTAX condition is raised instead.

Example 11.1. NOVALUE not raised for stems

```plaintext
/* The following does not raise NOVALUE. */
signal on novalue
a.=0
say a.z
say "NOVALUE is not raised."
exit

novalue:
say "NOVALUE is raised."
```

You can specify this condition only for SIGNAL ON.

SYNTAX
raised if any language-processing error is detected while the program is running. This includes all kinds of processing errors:
• True syntax errors
• “Run-time” errors (such as attempting an arithmetic operation on nonnumeric terms)
• Syntax errors propagated from higher call or method invocation levels
• Untrapped HALT conditions
• Untrapped NOMETHOD conditions

You can specify this condition only for SIGNAL ON.

Notes:
1. SIGNAL ON SYNTAX cannot trap the errors 3 and 5.
2. SIGNAL ON SYNTAX can trap the errors 6 and 30 only if they occur during the execution of an INTERPRET instruction.

For information on these errors, refer to Appendix C, Error Numbers and Messages.
USER raised if a condition specified on the USER option of CALL ON or SIGNAL ON occurs. USER conditions are raised by a RAISE instruction that specifies a USER option with the same usercondition name. The specified usercondition can be any symbol, including those specified as possible values for condition.

Any ON or OFF reference to a condition trap replaces the previous state (ON, OFF, or DELAY, and any trapname) of that condition trap. Thus, a CALL ON HALT replaces any current SIGNAL ON HALT (and a SIGNAL ON HALT replaces any current CALL ON HALT), a CALL ON or SIGNAL ON with a new trap name replaces any previous trap name, and any OFF reference disables the trap for CALL or SIGNAL.

11.1. Action Taken when a Condition Is Not Trapped

When a condition trap is currently disabled (OFF) and the specified condition occurs, the default action depends on the condition:

• For HALT and NOMETHOD, a SYNTAX condition is raised with the appropriate Rexx error number.

• For SYNTAX conditions, the clause in error is terminated, and a SYNTAX condition is propagated to each CALL instruction, INTERPRET instruction, message instruction, or clause with function or message invocations active at the time of the error, terminating each instruction if a SYNTAX trap is not active at the instruction level. If the SYNTAX condition is not trapped at any of the higher levels, processing stops, and a message (see Appendix C, Error Numbers and Messages) describing the nature of the event that occurred usually indicates the condition.

• For all other conditions, the condition is ignored and its state remains OFF.

11.2. Action Taken when a Condition Is Trapped

When a condition trap is currently enabled (ON) and the specified condition occurs, a CALL trapname or SIGNAL trapname instruction is processed automatically. You can specify the trapname after the NAME subkeyword of the CALL ON or SIGNAL ON instruction. If you do not specify a trapname, the name of the condition itself (for example, ERROR or FAILURE) is used.

For example, the instruction call on error enables the condition trap for the ERROR condition. If the condition occurred, then a call to the routine identified by the name ERROR is made. The instruction call on error name commanderror would enable the trap and call the routine COMMANDERROR if the condition occurred, and the caller usually receives an indication of failure.

The sequence of events, after a condition has been trapped, varies depending on whether a SIGNAL or CALL is processed:

• If the action taken is a SIGNAL, execution of the current instruction ceases immediately, the condition is disabled (set to OFF), and SIGNAL proceeds as usually.

If any new occurrence of the condition is to be trapped, a new CALL ON or SIGNAL ON instruction for the condition is required to re-enable it when the label is reached. For example, if SIGNAL ON SYNTAX is enabled when a SYNTAX condition occurs, a usual syntax error termination occurs if the SIGNAL ON SYNTAX label name is not found.

• If the action taken is a CALL, the CALL trapname proceeds in the usual way when the instruction completes. The call does not affect the special variable RESULT. If the routine should RETURN any data, that data is ignored.
When the condition is raised, and before the CALL is made, the condition trap is put into a delayed state. This state persists until the RETURN from the CALL, or until an explicit CALL (or SIGNAL) ON (or OFF) is made for the condition. This delayed state prevents a premature condition trap at the start of the routine called to process a condition trap. When a condition trap is in the delayed state, it remains enabled, but if the condition is raised again, it is either ignored (for ERROR and FAILURE) or (for the other conditions) any action (including the updating of the condition information) is delayed until one of the following events occurs:

1. A CALL ON or SIGNAL ON for the delayed condition is processed. In this case, a CALL or SIGNAL takes place immediately after the new CALL ON or SIGNAL ON instruction has been processed.

2. A CALL OFF or SIGNAL OFF for the delayed condition is processed. In this case, the condition trap is disabled and the default action for the condition occurs at the end of the CALL OFF or SIGNAL OFF instruction.

3. A RETURN is made from the subroutine. In this case, the condition trap is no longer delayed and the subroutine is called again immediately.

On RETURN from the CALL, the original flow of execution is resumed, that is, the flow is not affected by the CALL.

Notes:

1. In all cases, the condition is raised immediately upon detection. If SIGNAL ON traps the condition, the current instruction is ended, if necessary. Therefore, the instruction during which an event occurs can only be partly processed. For example, if SYNTAX is raised during the evaluation of the expression in an assignment, the assignment does not take place. Note that the CALL for traps for which CALL ON is enabled can only occur at clause boundaries. If these conditions arise in the middle of an INTERPRET instruction, execution of INTERPRET can be interrupted and resumed later. Similarly, other instructions, for example DO or SELECT, can be temporarily interrupted by a CALL at a clause boundary.

2. The state (ON, OFF, or DELAY, and any trapname) of each condition trap is saved on entry to a subroutine and is then restored on RETURN. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions set up by the caller. See Section 2.3, “CALL” for details of other information that is saved during a subroutine call.

3. The state of condition traps is not affected when an external routine is called by a CALL, even if the external routine is a Rexx program. On entry to any Rexx program, all condition traps have an initial setting of OFF.

4. While user input is processed during interactive tracing, all condition traps are temporarily set OFF. This prevents any unexpected transfer of control—for example, should the user accidentally use an uninitialized variable while SIGNAL ON NOVALUE is active. For the same reason, a syntax error during interactive tracing does not cause the exit from the program but is trapped specially and then ignored after a message is given.

5. The system interface detects certain execution errors either before the execution of the program starts or after the program has ended. SIGNAL ON SYNTAX cannot trap these errors.

Note that a label is a clause consisting of a single symbol followed by a colon. Any number of successive clauses can be labels; therefore, several labels are allowed before another type of clause.
11.3. Condition Information

When a condition is trapped and causes a SIGNAL or CALL, this becomes the current trapped condition, and certain condition information associated with it is recorded. You can inspect this information by using the \texttt{CONDITION} built-in function.

The condition information includes:

- The name of the current trapped condition
- The name of the instruction processed as a result of the condition trap (CALL or SIGNAL)
- The status of the trapped condition
- A descriptive string (see Section 11.3.1, “Descriptive Strings”) associated with that condition
- Optional additional object information (see Section 11.3.2, “Additional Object Information”)

The current condition information is replaced when control is passed to a label as the result of a condition trap (CALL ON or SIGNAL ON). Condition information is saved and restored across subroutine or function calls, including one because of a CALL ON trap and across method invocations. Therefore, a routine called by CALL ON can access the appropriate condition information. Any previous condition information is still available after the routine returns.

11.3.1. Descriptive Strings
The descriptive string varies, depending on the condition trapped:

\textbf{ERROR}
- The string that was processed and resulted in the error condition.

\textbf{FAILURE}
- The string that was processed and resulted in the failure condition.

\textbf{HALT}
- Any string associated with the halt request. This can be the null string if no string was provided.

\textbf{LOSTDIGITS}
- The number with excessive digits that caused the LOSTDIGITS condition.

\textbf{NOMETHOD}
- The name of the method that could not be found.

\textbf{NOSTRING}
- The readable string representation of the object causing the NOSTRING condition.

\textbf{NOTREADY}
- The name of the stream being manipulated when the error occurred and the NOTREADY condition was raised. If the stream was a default stream with no defined name, then the null string might be returned.

\textbf{NOVALUE}
- The derived name of the variable whose attempted reference caused the NOVALUE condition.
Additional Object Information

SYNTAX
Any string the language processor associated with the error. This can be the null string if you did
not provide a specific string. Note that the special variables RC and SIGL provide information on
the nature and position of the processing error. You can enable the SYNTAX condition trap only by
using SIGNAL ON.

USER
Any string specified by the DESCRIPTION option of the RAISE instruction that raised the
condition. If a description string was not specified, a null string is used.

11.3.2. Additional Object Information
The language processor can provide additional information, depending on the condition trapped:

NOMETHOD
The object that raised the NOMETHOD condition.

NOSTRING
The object that caused the NOSTRING condition.

NOTREADY
The stream object that raised the NOTREADY condition.

SYNTAX
An array containing the objects substituted into the secondary error message (if any) for the
syntax error. If the message did not contain substitution values, a zero element array is used.

USER
Any object specified by an ADDITIONAL or ARRAY option of the RAISE instruction that raised the
condition.

11.3.3. The Special Variable RC
When an ERROR or FAILURE condition is trapped, the Rexx special variable RC is set to the
command return code before control is transferred to the target label (whether by CALL or by
SIGNAL).

Similarly, when SIGNAL ON SYNTAX traps a SYNTAX condition, the special variable RC is set to the
syntax error number before control is transferred to the target label.

11.3.4. The Special Variable SIGL
Following any transfer of control because of a CALL or SIGNAL, the program line number of the
clause causing the transfer of control is stored in the special variable SIGL. If the transfer of control
is because of a condition trap, the line number assigned to SIGL is that of the last clause processed
(at the current subroutine level) before the CALL or SIGNAL took place. The setting of SIGL is
especially useful after a SIGNAL ON SYNTAX trap when the number of the line in error can be used,
for example, to control a text editor. Typically, code following the SYNTAX label can PARSE SOURCE
to find the source of the data and then call an editor to edit the source file, positioned at the line in
error. Note that in this case you might have to run the program again before any changes made in the
editor can take effect.

Alternatively, SIGL can help determine the cause of an error (such as the occasional failure of a
function call) as in the following example:


**Example 11.2. SIGL**

```
signal on syntax
a = a + 1    /* This is to create a syntax error */
say "SYNTAX error not raised"
exit

/* Standard handler for SIGNAL ON SYNTAX */
syntax:
say "Rexx error" rc "in line" sigl":" "ERRORTEXT"(rc)
say "SOURCELINE"(sigl)
trace ?r; nop
```

This code first displays the error code, line number, and error message. It then displays the line in error, and finally drops into debug mode to let you inspect the values of the variables used at the line in error.

### 11.3.5. Condition Object

A condition object is a Directory returned by the Object option of the CONDITION built-in function. This directory contains all information currently available on a trapped condition. The information varies with the trapped condition. `.nil` is returned for any entry not available to the condition. The following entries can be found in a condition object:

- **ADDITIONAL**
  The additional information object associated with the condition. This is the same object that the Additional option of the CONDITION built-in function returns. The ADDITIONAL information may be specified with the ADDITIONAL or ARRAY options of the RAISE instruction.

- **DESCRIPTION**
  The string describing the condition. The Description option of the CONDITION built-in function also returns this value.

- **INSTRUCTION**
  The keyword for the instruction executed when the condition was trapped, either CALL or SIGNAL. The Instruction option of the CONDITION built-in function also returns this value.

- **CONDITION**
  The name of the trapped condition. The Condition name option of the CONDITION built-in function also returns this value.

- **RESULT**
  Any result specified on the RETURN or EXIT options of a RAISE instruction.

- **RC**
  The major Rexx error number for a SYNTAX condition. This is the same error number assigned to the special variable RC.

- **CODE**
  The detailed identification of the error that caused a SYNTAX condition. This number is a nonnegative number in the form `nn.nn`. The integer portion is the Rexx major error number (the same value as the RC entry). The fractional portion is a subcode that gives a precise indication of the error that occurred.
ERRORTEXT
    The primary error message for a SYNTAX condition. This is the same message available from the
    ERRORTEXT built-in function.

MESSAGE
    The secondary error message for a SYNTAX condition. The message also contains the content of
    the ADDITIONAL information.

PACKAGE
    The Package object associated with the program where a condition was raised.

POSITION
    The line number in source code at which a condition was raised.

PROGRAM
    The name of the program where a condition was raised.

STACKFRAMES
    A single-index list of StackFrame instances.

TRACEBACK
    A single-index list of formatted traceback lines.

PROPAGATED
    .false if the condition was raised at the same level as the condition trap or the value .true if
    the condition was reraised with RAISE PROPAGATE.
Concurrenty

Conceptually, each Rexx object is like a small computer with its own processor to run its methods, its memory for object and method variables, and its communication links to other objects for sending and receiving messages. This is object-based concurrency. It lets more than one method run at the same time. Any number of objects can be active (running) at the same time, exchanging messages to communicate with, and synchronize, each other.

12.1. Early Reply

Early reply provides concurrent processing. A running method returns control, and possibly a result, to the point from which it was called; meanwhile it continues running. The following figure illustrates this concept.

Method A

```
V-B
  do 3
  say 'Hello'
end
```

Method B

```
Reply earlyresult
  do 3
  say 'Goodbye'
end
```

Possible Output

```
Hello
Hello
Goodbye
Hello
Goodbye
```

Figure 12.1. Early Reply

Method A includes a call to Method B. Method B contains a REPLY instruction. This returns control and a result to method A, which continues processing with the line after the call to Method B. Meanwhile, Method B also continues running.

The chains of execution represented by method A and method B are called activities. An activity is a thread of execution that can run methods concurrently with methods on other activities.

An activity contains a stack of invocations that represent the Rexx programs running on the activity. An invocation can be a main program invocation, an internal function or subroutine call, an external function or subroutine call, an INTERPRET instruction, or a message invocation. An invocation is activated when an executable unit is invoked and removed (popped) when execution completes. In Figure 12.1, "Early Reply", the programs begins with a single activity. The activity contains a single invocation, method A. When method A invokes method B, a second invocation is added to the activity.

When method B issues a REPLY, a new activity is created (activity 2). Method B's invocation is removed from activity 1, and pushed on to activity 2. Because activities can execute concurrently, both method A and method B continue processing. The following figures illustrate this concept.
Here is an example of using early reply to run methods concurrently.

Example 12.1. REPLY instruction

```verbatim
/* Example of early reply */

object1 = .example-new
object2 = .example-new

say object1-repeat(10, "Object 1 running")
say object2-repeat(10, "Object 2 running")
say "Main ended."
exit

::class example
::method repeat
use arg reps,msg
reply "Repeating" msg"," reps "times."
do reps
    say msg
end
```
12.2. Message Objects

A Message object is an intermediary between two objects that enables concurrent processing. All objects inherit the start method from the Object class. To obtain a message object, an object sends a start message to the object to which the message object will convey a message. The message is an argument to the start message as in the following example:

```
a=p~start("REVERSE")
```

This line of code creates a message object, A, and sends it a start message. The message object then sends the REVERSE message to object P. Object P receives the message, performs any needed processing, and returns a result to message object A. Meanwhile the object that obtained message object A continues its processing. When message object A returns, it does not interrupt the object that obtained it. It waits until this object requests the information. Here is an example of using a message object to run methods concurrently.

Example 12.2. Message object usage

```
/* Example of using a message object */
object1 = .example~new
object2 = .example~new

a = object1~start("REPEAT",10,"Object 1 running")
b = object2~start("REPEAT",10,"Object 2 running")

say a~result
say b~result
say "Main ended."
exit

::class example
::method repeat
use arg reps,msg
do reps
    say msg
end
return "Repeated" msg"," reps "times."
```

12.3. Default Concurrency

The instance methods of a class use the EXPOSE instruction to define a set of object variables. This collection of variables belonging to an object is called its object variable pool. The methods a class defines and the variables these methods can access is called a scope. Rexx's default concurrency exploits the idea of scope. The object variable pool is a set of object subpools, each representing the set of variables at each scope of the inheritance chain of the class from which the object was created. Only methods at the same scope can access object variables at any particular scope. This prevents any name conflicts between classes and subclasses, because the object variables for each class are in different scopes.

If you do not change the defaults, only one method of a given scope can run on a single object at a time. Once a method is running on an object, the language processor blocks other methods on other activities from running in the same object at the same scope until the method that is running completes. Thus, if different activities send several messages within a single scope to an object the methods run sequentially.
The next example shows how the default concurrency works.

**Example 12.3. Default concurrency for methods**

```clike
/* Example of default concurrency for methods of different scopes */
object1 = .subexample~new
say object1~repeat(8, "Object 1 running call 1")  /* These calls run */
say object1~repeater(8, "Object 1 running call 2")  /* concurrently */
say "Main ended."
exit
::class example
::method repeat
use arg reps,msg
reply "Repeating" msg"," reps "times."
do reps
  say msg
end
::class subexample subclass example
::method repeater
use arg reps,msg
reply "Repeating" msg"," reps "times."
do reps
  say msg
end
```

The preceding example produces output such as the following:

Repeating Object 1 running call 1, 8 times.
Object 1 running call 1
Repeating Object 1 running call 2, 8 times.
Object 1 running call 1
Object 1 running call 2
Main ended.
Object 1 running call 1
Object 1 running call 2
Object 1 running call 1
Object 1 running call 2
Object 1 running call 1
Object 1 running call 2
Object 1 running call 1
Object 1 running call 2
Object 1 running call 1
Object 1 running call 2

The following example shows that methods of the same scope do not run concurrently by default.

**Example 12.4. Default concurrency for methods**

```clike
/* Example of methods with the same scope not running concurrently*/
object1 = .example~new
say object1~repeat(10,"Object 1 running call 1")  /* These calls */
say object1~repeat(10,"Object 1 running call 2")  /* cannot run */
```
The REPEAT method includes a REPLY instruction, but the methods for the two REPEAT messages in the example cannot run concurrently. This is because REPEAT is called twice at the same scope and requires exclusive access to the object variable pool. The REPLY instruction causes the first REPEAT message to transfer its exclusive access to the object variable pool to a new activity and continue execution. The second REPLY message also requires exclusive access and waits until the first method completes.

If the original activity has more than one method active (nested method calls) with exclusive variable access, the first REPLY instruction is unable to transfer its exclusive access to the new activity and must wait until the exclusive access is again available. This may allow another method on the same object to run while the first method waits for exclusive access.

12.3.1. Sending Messages within an Activity
Whenever a message is invoked on an object, the activity acquires exclusive access (a lock) for the object's scope. Other activities that send messages to the same object that required the locked scope waits until the first activity releases the lock.

Suppose object A is running method Y, which includes:

```
self~z
```

Sequential processing does not allow method Z to begin until method Y has completed. However, method Y cannot complete until method Z runs. A similar situation occurs when a subclass's overriding method does some processing and passes a message to its superclasses' overriding method. Both cases require a special provision: If an invocation running on an activity sends another message to the same object, this method is allowed to run because the activity has already acquired the lock for the scope. This allows nested, nonconcurrent method invocations on a single activity without causing a deadlock situation. The language processor regards these additional messages as subroutine calls.

Here is an example showing the special treatment of single activity messages. The REPEATER and REPEAT methods have the same scope. REPEAT runs on the same object at the same time as the REPEATER method because a message to SELF runs the REPEAT method. The language processor treats this as a subroutine call rather than as concurrently running two methods.

```
/* Example 12.5. Sending a message to SELF */

object1 = .example~new
object2 = .example~new

say object1~repeater(10, "Object 1 running")
say object2~repeater(10, "Object 2 running")
say "Main ended."
```
The activity locking rules also allow indirect object recursion. The following figure illustrates indirect object recursion.

**Figure 12.4. Indirect Object Recursion**

Method M in object A sends object B a message to run method N. Method N sends a message to object A, asking it to run method O. Meanwhile, method M is still running in object A and waiting for a result from method N. A deadlock would result. Because the methods are all running on the same activity, no deadlock occurs.

### 12.4. Using Additional Concurrency Mechanisms

Rexx has additional concurrency mechanisms that can add full concurrency so that more than one method of a given scope can run in an object at a time:

- The SETUNGUARDED method of the Method class and the UNGUARDED option of the METHOD directive provide unconditional concurrency
- GUARD OFF and GUARD ON control a method's exclusive access to an object's scope

#### 12.4.1. SETUNGUARDED Method and UNGUARDED Option

The SETUNGUARDED method of the Method class and the UNGUARDED option of the ::METHOD directive control locking of an object's scope when a method is invoked. Both let a method run even if another method is active on the same object.

Use the SETUNGUARDED method or UNGUARDED option only for methods that do not need exclusive use of their object variable pool, that is, methods whose execution can interleave with another method's execution without affecting the object's integrity. Otherwise, concurrent methods can produce unexpected results.

To use the SETUNGUARDED method for a method you have created with the NEW method of the Method class, you specify:
GUARD ON and GUARD OFF

methodname~SETUNGUARDED

(See Section 5.1.3.19, “setUnguarded” for details about SETUNGUARDED.)

Alternately, you can define a method with the ::METHOD directive, specifying the UNGUARDED option:

::METHOD methodname UNGUARDED

12.4.2. GUARD ON and GUARD OFF

You might not be able to use the SETUNGUARDED method or UNGUARDED option in all cases. A method might need exclusive use of its object variables, then allow methods on other activities to run, and perhaps later need exclusive use again. You can use GUARD ON and GUARD OFF to alternate between exclusive use of an object’s scope and allowing other activities to use the scope.

By default, a method must wait until a currently running method is finished before it begins. GUARD OFF lets another method (running on a different activity) that needs exclusive use of the same object variables become active on the same object. See Section 2.9, “GUARD” for more information.

12.4.3. Guarded Methods

Concurrency requires the activities of concurrently running methods to be synchronized. Critical data must be safeguarded so diverse methods on other activities do not perform concurrent updates. Guarded methods satisfy both these needs.

A guarded method combines the UNGUARDED option of the ::METHOD directive or the SETUNGUARDED method of the Method class with the GUARD instruction.

The UNGUARDED option and the SETUNGUARDED method both provide unconditional concurrency. Including a GUARD instruction in a method makes concurrency conditional:

GUARD ON WHEN expression

If the expression on the GUARD instruction evaluates to .true, the method continues to run. If the expression on the GUARD instruction evaluates to .false, the method does not continue running. GUARD reevaluates the expression whenever the value of an exposed object variable changes. When the expression evaluates to 1, the method resumes running. You can use GUARD to block running any method when proceeding is not safe. (See Section 2.9, “GUARD” for details about GUARD.)

Note

It is important to ensure that you use an expression that can be fulfilled. If the condition expression cannot be met, GUARD ON WHEN puts the program in a continuous wait condition. This can occur in particular when several activities run concurrently. In this case, a second activity can make the condition expression invalid before GUARD ON WHEN can use it.
To avoid this, ensure that the GUARD ON WHEN statement is executed before the condition is set to true. Keep in mind that the sequence of running activities is not determined by the calling sequence, so it is important to use a logic that is independent of the activity sequence.

### 12.4.4. Additional Examples

The following example uses REPLY in a method for a write-back cache.

#### Example 12.6. Method Write_Back

```rexx
/* Method Write_Back */
use arg data    /* Save data to be written */
reply 0         /* Tell the sender all was OK */
self~disk_write(data) /* Now write the data */
```

The REPLY instruction returns control to the point at which method Write_Bakc was called, returning the result 0. The caller of method Write_Bakc continues processing from this point; meanwhile, method Write_Bakc also continues processing.

The following example uses a message object. It reads a line asynchronously into the variable `nextline`:

#### Example 12.7. Message to INFILE

```rexx
mymsg = infile~start("READLINE") /* Gets message object to carry */
/* message to INFILE */
/* do other work */
nextline=mymsg~result /* Gets result from message object */
```

This creates a message object that waits for the read to finish while the sender continues with other work. When the line is read, the `mymsg` message object obtains the result and holds it until the sender requests it.

Semaphores and monitors (bounded buffers) synchronize concurrency processes. Giving readers and writers concurrent access is a typical concurrency problem. The following sections show how to use guarded methods to code semaphore and monitor mechanisms and to provide concurrency for readers and writers.

#### 12.4.4.1. Semaphores

A semaphore is a mechanism that controls access to resources, for example, preventing simultaneous access. Synchronization often uses semaphores. Here is an example of a semaphore class:

#### Example 12.8. Example of a Rexx Semaphore Class

```rexx
/* A Rexx Semaphore Class. */
/* This file implements a semaphore class in Rexx. The class is defined to */
/* the Global Rexx Environment. The following methods are defined for */
/* this class: */
/* init - Initializes a new semaphore. Accepts the following positional */
/* parameters: */
/* 'name' - global name for this semaphore */
/* if named default to set name in */
```
the class semDirectory

noShare - do not define named semaphore

in class semDirectory

Initial state (0 or 1)

setInitialState - Allow for subclass to have some post-initialization, and do setup based on initial state of semaphore

Waiting - Is the number of objects waiting on this semaphore.

Shared - Is this semaphore shared (Global).

Named - Is this semaphore named.

Name - Is the name of a named semaphore.

setSem - Sets the semaphore and returns previous state.

resetSem - Sets state to unSet.

querySem - Returns current state of semaphore.

SemaphoreMeta - Is the metaclass for the semaphore classes. This class is set up so that when a namedSemaphore is shared, it maintains these named/shared semaphores as part of its state. These semaphores are maintained in a directory, and an UNKNOWN method is installed on the class to forward unknown messages to the directory. In this way the class can function as a class and "like" a directory, so [] syntax can be used to retrieve a semaphore from the class.

The following are in the subclass EventSemaphore.

Post - Posts this semaphore.

Query - Queries the number of posts since the last reset.

Reset - Resets the semaphore.

Wait - Waits on this semaphore.

The following are in the subclass MutexSemaphore

requestMutex - Gets exclusive use of semaphore.

releaseMutex - Releases to allow someone else to use semaphore.

NOTE: Currently anyone can issue a release (need not be the owner).

************************************************************/

===         Start of Semaphore class.                        =====

************************************************************/

::class SemaphoreMeta subclass class

::method init

expose semDict

/* Be sure to initialize parent */

.message-new(self, .array-of("INIT", super), "a", arg(1,"a"))-send

semDict = .directory-new

::method unknown

expose semDict

use arg msgName, args

/* Forward all unknown messages */

/* to the semaphore dictionary */

.message-new(semDict, msgName, "a", args)-send

if var("RESULT") then

return result

else

return

::class Semaphore subclass object metaclass SemaphoreMeta

::method init

expose sem waits shared name

use arg semname, shr, state

waits = 0 /* No one waiting */

name = "" /* Assume unnamed */
shared = 0                                /* Assume not shared                */
sem = 0                                   /* Default to not posted            */

if state = 1 Then                         /* Should initial state be set?     */
    sem = 1
endif

if VAR("SEMNAME") & semname \= "" Then Do
    name = semname                          /* Yes, so set the name             */
endif

if shr \= "NOSHARE" Then Do             /* Do we want to share this sem?    */
    shared = 1                            /* Yes, mark it shared              */
    self~class[name] = self
endif
End

End

self~setInitialState(sem)                 /* Initialize initial state         */

::method setInitialState
/* This method intended to be               */
nop                                         /* overridden by subclasses          */
::method setSem
expose sem
oldState = sem
sem = 1                                     /* Set new state to 1               */
return oldState
::method resetSem
expose sem
sem = 0
return 0
::method querySem
expose sem
return sem
::method shared
expose shared
return shared                               /* Return .true or .false            */
::method named
expose name
if name = "" Then return 0                  /* No, not named                     */
else return 1                               /* Yes, it is named                  */
::method name
expose name
return name                                 /* Return name or ""                 */
::method incWaits
expose waits
waits = waits + 1                           /* One more object waiting           */
::method decWaits
expose Waits
waits = waits - 1                           /* One object less waiting           */
::method Waiting
expose Waits
return waits                                /* Return number of objects waiting  */
/*  ===================================================================================  */
/*  ===          Start of EventSemaphore class.                                        ===  */
/*  ===================================================================================  */
::class EventSemaphore subclass Semaphore public
::method setInitialState
  expose posted posts
  use arg posted

  if posted  then posts = 1
  else posts = 0
::method post
  expose posts posted

  self-setSem  /* Set semaphore state */
  posted = 1   /* Mark as posted */
  reply
  posts = posts + 1 /* Increase the number of posts */
::method wait
  expose posted

  self-incWaits /* Increment number waiting */
  guard off
  guard on when posted /* Now wait until posted */
  reply /* Return to caller */
  self-decWaits /* Cleanup, 1 less waiting */
::method reset
  expose posted posts

  posted = self-resetSem /* Reset semaphore */
  reply /* Do an early reply */
  posts = 0 /* Reset number of posts */
::method query
  expose posts

  /* Return number of times */
  return posts /* Semaphore has been posted */
/*===================================================================*/
/*===         Start of MutexSemaphore class.                             */
/*===================================================================*/
::class MutexSemaphore subclass Semaphore public
::method setInitialState
  expose owned
  use arg owned

::method requestMutex
  expose Owned

  Do forever /* Do until we get the semaphore */
    owned = self-setSem /* Was semaphore already set? */
    if Owned = 0 /* Wasn't owned; we now have it */
      Then leave /* Go up and see if we can get it */
    else Do
      self-incWaits /* Turn off guard status to let */
      /* others come in */
      guard off
      /* Wait until not owned and get */
      /* guard */
      guard on when \Owned /* One less waiting for MUTEX */
      self-decWaits /* Go up and see if we can get it */
    End
End
::method releaseMutex
  expose owned

  /* Do an early reply */
12.4.4.2. Monitors (Bounded Buffer)

A monitor object consists of a number of client methods, WAIT and SIGNAL methods for client methods to use, and one or more condition variables. Guarded methods provide the functionality of monitors. Do not confuse this with the Monitor class (see Section 5.4.10, "Monitor Class").

Example 12.9. Example of a Rexx Monitor Class

```rexx
::method init
 /* Initialize the bounded buffer */
expose size in out n
use arg size
in = 1
out = 1
n = 0

::method append unguarded
 /* Add to the bounded buffer if not full */
expose n size b. in
guard on when n < size
use arg b.in
in = in//size+1
n = n+1

::method take
 /* Remove from the bounded buffer if not empty */
expose n b. out size
guard on when n > 0
reply b.out
out = out//size+1
n = n-1
```

12.4.4.3. Readers and Writers

The concurrency problem of the readers and writers requires that writers exclude writers and readers, whereas readers exclude only writers. The UNGUARDED option is required to allow several concurrent readers.

Example 12.10. Example of a Rexx Readers and Writers

```rexx
::method init
expose readers writers
readers = 0
writers = 0

::method read unguarded
 /* Read if no one is writing */
expose writers readers
guard on when writers = 0
```
readers = readers + 1
guard off

/* Read the data */
say "Reading (writers:" writers", readers:" readers")..

guard on
readers = readers - 1

::method write unguarded
/* Write if no-one is writing or reading */
expose writers readers
guard on when writers + readers = 0
writers = writers + 1

/* Write the data */
say "Writing (writers:" writers", readers:" readers")..

writers = writers - 1
The Security Manager

The security manager provides a special environment that is safe even if agent programs try to perform unexpected actions. The security manager is called if an agent program tries to:

- Call an external function
- Use a host command
- Use the ::REQUIRES directive
- Access the .LOCAL directory
- Access the .ENVIRONMENT directory
- Use a stream name in the input and output built-in functions (CHARIN, CHAROUT, CHARST, LINEIN, LINEOUT, LINES, and STREAM)
- Send a message for a protected method to an object

13.1. Calls to the Security Manager

When the language processor reaches any of the defined security checkpoints, it sends a message to the security manager for the particular checkpoint. The message has a single argument, a directory of information that pertains to the checkpoint. If the security manager chooses to handle the action instead of the language processor, the security manager uses the checkpoint information directory to pass information back to the language processor.

Security manager methods must return a value of either .false or .true to the language processor. A value of .false indicates that the program is authorized to perform the indicated action. In this case, processing continues as usual. A value of .true indicates that the security manager performed the action itself. The security manager sets entries in the information directory to pass results for the action back to the language processor. The security manager can also use the RAISE instruction to raise a program error for a prohibited access. Error message 98.948 indicates authorization failures.

The defined checkpoints, with their arguments and return values, are:

**CALL**

sent for all external function calls. The information directory contains the following entries:
- NAME
  - The name of the invoked function.
- ARGUMENTS
  - An array of the function arguments.

When the CALL method returns .false, indicating that it handled the external call, the entry RESULT (if any) in the information directory is used as the function call result.

**COMMAND**

sent for all host command instructions. The information directory contains the following entries:
- COMMAND
  - The string that represents the host command.
- ADDRESS
  - The name of the target ADDRESS environment for the command.
When the COMMAND method returns `.false`, indicating that it handled the command, the 
security manager uses the following information directory entries to return the command results:

**RC**
The command return code. If the entry is not set, a return code of `.true` is used.

**FAILURE**
If a FAILURE entry is added to the information directory, a Rexx FAILURE condition is raised.

**ERROR**
If an ERROR entry is added to the information directory, a Rexx ERROR condition is raised. 
The ERROR condition is raised only if the FAILURE entry is not set.

**REQUIRES**
Sent whenever a `::REQUIRES` directive in the file is processed. The information directory contains 
the following entry:

**NAME**
The name of the file specified on the `::REQUIRES` directive.

When the REQUIRES method returns `.false`, indicating that it handled the request, the 
entry NAME in the information directory is used as the actual file to load for the request. The 
REQUIRES method can also provide a security manager to be used for the program loaded by 
the `::REQUIRES` directive by setting the information directory entry SECURITYMANAGER to the 
desired security manager object.

**LOCAL**
Sent whenever Rexx is going to access an entry in the `.LOCAL` directory as part of the resolution 
of the environment symbol name. The information directory contains the following entry:

**NAME**
The name of the target directory entry.

When the LOCAL method returns `.false`, indicating that it handled the request, the information 
directory entry RESULT contains the directory entry. When RESULT is not set and the method 
returns `.false`, this is the same as a failure to find an entry in the `.LOCAL` directory. Rexx 
continues with the next step in the name resolution.

**ENVIRONMENT**
Sent whenever Rexx is going to access an entry in the `.ENVIRONMENT` directory as part of the 
resolution of the environment symbol name. The information directory contains the following entry:

**NAME**
The name of the target directory entry.

When the ENVIRONMENT method returns `.false`, indicating that it handled the request, the information 
directory entry RESULT contains the directory entry. When RESULT is not set and the method 
returns `.false`, this is the same as a failure to find an entry in the `.ENVIRONMENT` 
directory. Rexx continues with the next step in the name resolution.

**STREAM**
Sent whenever one of the Rexx input and output built-in functions (CHARIN, CHAROUT, CHAR, 
LINEIN, LINEOUT, LINES, or STREAM) needs to resolve a stream name. The information 
directory contains the following entry:

**NAME**
The name of the target stream.
When the STREAM method returns `.false`, indicating that it handled the request, the information directory entry STREAM must be set to an object to be used as the stream target. This should be a stream object or another object that supports the Stream class methods.

**METHOD**

sent whenever a secure program attempts to send a message for a protected method (see the `::METHOD` directive [Section 3.5, "::METHOD"](##)) to an object. The information directory contains the following entries:

- **OBJECT**
  - The object the protected method is issued against.

- **NAME**
  - The name of the protected method.

- **ARGUMENTS**
  - An array containing the method arguments.

When the METHOD method returns `.false`, indicating that it handled the request, the entry RESULT (if any) in the information directory is used as the method result.

### 13.1.1. Example

The following agent program includes all the actions for which the security manager defines checkpoints (for example, by calling an external function).

**Example 13.1. Agent Program**

```/* Agent */
"echo Hello There"
call rxfuncadd "rxcalc_sqrt", "rxmath", "rxcalc_sqrt"
say result
say syssleep(1)
say linein("./profile")
say .methods
$json~define(""
::requires json.cls
```

The following server implements the security manager with three levels of security. For each action the security manager must check (for example, by calling an external routine):

1. The audit manager (Dumper class) writes a record of the event but then permits the action.
2. The closed cell manager (noWay class) does not permit the action to take place and raises an error.
3. The replacement execution environment (Replacer class, a subclass of the noWay class) replaces the prohibited action with a different action.

**Example 13.2. Example of Server Implementing Security Manager**

```/* Server implements security manager */
parse arg program
routine = .Routine~newFile(program)
say "Calling program " program " with an audit manager:"
pull
routine~setSecurityManager(.Dumper~new(.stderr))
routine~call
```
Example

say "Calling program" program "with a function replacement execution environment:"
pull
routine-setSecurityManager(.Replacer-new)
routine-call
say "Calling program" program "with a closed cell manager:"
pull
routine-setSecurityManager(.noWay-new)
signal on syntax
routine-call
exit

syntax:
say "Agent program terminated with an authorization failure"
say condition("additional")
exit

-- A Security Manager that keeps an audit record, and permits the request to run
::class Dumper public
::method init
-- save our target stream for output
expose stream
use arg stream

-- unknown will trap all Security Manager checkpoints
::method unknown
expose stream
use arg name, args
-- write an audit record
stream-lineout(.DateTime-new "Called for event" name)
info = args[1]
-- info directory is the first arg
do name over info
-- write the info directory contents
stream-lineout(.DateTime-new "Info item" name":" info[name])
end
return 0

-- A closed cell Security Manager blocking all requests
::class noWay
-- unknown will trap all Security Manager checkpoints
::method unknown
-- raise an error for each checkpoint
raise syntax 98.948 array(arg(1) "blocked by Security manager")

-- A Security Manager that replaces prohibited actions with a different one
::class Replacer subclass noWay -- inherit restrictive UNKNOWN method
-- command checkpoint
::method command
use arg info
info-rc = 1234
-- replace the command return code
info-failure = .true
-- raise a FAILURE condition
return 1

-- external call checkpoint
::method call
use arg info
info-result = "blocked"
return 1

-- STREAM checkpoint
::method stream
use arg info
-- always replace with a different stream
info-stream = .stream-new("SecurityManager.txt")
return 1

-- .local variable lookup
::method local
   return 1 -- handle. but return no value

-- .environment variable lookup
::method environment
   return 1 -- handle. but return no value

-- protected method invocation
::method method
   use arg info
   info-result = "blocked"
   return 1 -- we handled this

-- ::REQUIRES directive
::method requires
   use arg info
   info-name = "SecurityManager.cls"
   info-securitymanager = self -- load under this authority
   return 1 -- we handled this
Chapter 14.

Input and Output Streams

Rexx defines Stream class methods to handle input and output and maintains the I/O functions for input and output externals. Using a mixture of Rexx I/O methods and Rexx I/O functions can cause unpredictable results. For example, using the LINEOUT method and the LINEOUT function on the same persistent stream object can cause overlays.

When a Rexx I/O function creates a stream object, the language processor maintains the stream object. When a Rexx I/O method creates a stream object, it is returned to the program to be maintained. Because of this, when Rexx I/O methods and Rexx I/O functions referring to the same stream are in the same program, there are two separate stream objects with different read and write pointers. The program needs to synchronize the read and write pointers of both stream objects, or overlays occur.

To obtain a stream object (for example, MYFIL), you could use:

Example 14.1. Obtaining a stream object

MyStream = .stream~new("MYFIL")

You can manipulate stream objects with character or line methods:

Example 14.2. Performing input on a stream object

nextchar = MyStream~charin()
nextline = MyStream~linein()

In addition to stream objects, the language processor defines an external data queue object for interprogram communication. This queue object understands line functions only.

A stream object can have a variety of sources or destinations including files, serial interfaces, displays, or networks. It can be transient or dynamic, for example, data sent or received over a serial interface, or persistent in a static form, for example, a disk file.

Housekeeping for stream objects (opening and closing files, for example) is not explicitly part of the language definition. However, Rexx provides methods, such as CHARIN and LINEIN, that are independent of the operating system and include housekeeping. The COMMAND method provides the stream_command argument for those situations that require more granular access to operating system interfaces.

14.1. The Input and Output Model

The model of input and output for Rexx consists of the following logically distinct parts:

- One or more input stream objects
- One or more output stream objects
- One or more external data queue objects

The Rexx methods, instructions, and built-in routines manipulate these elements as follows.
14.1.1. Input Streams

Input to Rexx programs is in the form of a serial character stream generated by user interaction or has the characteristics of one generated this way. You can add characters to the end of some stream objects asynchronously; other stream objects might be static or synchronous.

The methods and instructions you can use on input stream objects are:

- **CHARIN method**—reads input stream objects as characters.
- **LINEIN method**—reads input stream objects as lines.
- **PARSE PULL and PULL instructions**—read the default input stream object (.INPUT), if the external data queue is empty. PULL is the same as PARSE UPPER PULL.
- **PARSE LINEIN instruction**—reads lines from the default input stream object regardless of the state of the external data queue. Usually, you can use PULL or PARSE PULL to read the default input stream object.

In a persistent stream object, the Rexx language processor maintains a current read position. For a persistent stream:

- The CHARS method returns the number of characters currently available in an input stream object from the read position through the end of the stream (including any line-end characters).
- The LINES method determines if any data remains between the current read position and the end of the input stream object.
- You can move the read position to an arbitrary point in the stream object with:
  - The SEEK or POSITION method of the Stream class
  - The COMMAND method’s SEEK or POSITION argument
  - The start argument of the CHARIN method
  - The line argument of the LINEIN method

When the stream object is opened, this position is the start of the stream.

In a transient stream, no read position is available. For a transient stream:

- The CHARS and LINES methods attempt to determine if data is present in the input stream object. These methods return the value 1 for a device if data is waiting to be read or a determination cannot be made. Otherwise, these methods return 0.
- The SEEK and POSITION methods of the Stream class and the COMMAND method’s SEEK and POSITION arguments are not applicable to transient streams.

14.1.2. Output Streams

Output stream methods provide for output from a Rexx program. Output stream methods are:

- **SAY instruction**—writes to the default output stream object (.OUTPUT).
- **CHAROUT method**—writes in character form to either the default or a specified output stream object.
• LINEOUT method—writes in lines to either the default or a specified output stream object.

LINEOUT and SAY write line-end characters at the end of each line. Depending on the operating system or hardware, other modifications or formatting can be applied; however, the output data remains a single logical line.

The Rexx language processor maintains the current write position in a stream. It is separate from the current read position. Write positioning is usually at the end of the stream (for example, when the stream object is first opened), so that data can be appended to the end of the stream. For persistent stream objects, you can set the write position to the beginning of the stream to overwrite existing data by giving a value of 1 for the CHAROUT start argument or the LINEOUT line argument. You can also use the CHAROUT start argument, the LINEOUT line argument, the SEEK or POSITION method, or the COMMAND method's SEEK or POSITION stream_command to direct sequential output to some arbitrary point in the stream.

Note

Once data is in a transient output stream object (for example, a network or serial link), it is no longer accessible to Rexx.

14.1.3. External Data Queue

Rexx provides queuing services entirely separate from interprocess communications queues.

The external data queue is a list of character strings that only line operations can access. It is external to Rexx programs in that other Rexx programs can have access to the queue.

The external data queue forms a Rexx-defined channel of communication between programs. Data in the queue is arbitrary; no characters have any special meaning or effect.

Apart from the explicit Rexx operations described here, no detectable change to the queue occurs while a Rexx program is running, except when control leaves the program and is manipulated by external means (such as when an external command or routine is called).

There are two kinds of queues in Rexx. Both kinds are accessed and processed by name.

14.1.3.1. Unnamed Queues

One unnamed queue is automatically provided for each Rexx program in operation. Its name is always "QUEUE:“, and the language processor creates it when Rexx is called and no queue is currently available. All processes that are children of the process that created the queue can access it as long as the process that created it is still running. However, other processes cannot share the same unnamed queue. The queue is deleted when the process that created it ends.

14.1.3.2. Named Queues

Your program creates (and deletes) named queues. You can name the queue yourself or leave the naming to the language processor. Your program must know the name of the queue to use a named queue. To obtain the name of the queue, use the RXQUEUE function:
This sets the new queue name and returns the name of the previous queue.

The following Rexx instructions manipulate the queue:

- **PULL** or **PARSE PULL**—reads a string from the head of the queue. If the queue is empty, these instructions take input from .INPUT.
- **PUSH**—stacks a line on top of the queue (LIFO).
- **QUEUE**—adds a string to the tail of the queue (FIFO).

Rexx functions that manipulate **QUEUE** as a device name are:

- **LINEIN(“QUEUE:“)**—reads a string from the head of the queue. If the queue is empty the program waits for an entry to be placed on the queue.
- **LINEOUT(“QUEUE:“,”string“)**—adds a string to the tail of the queue (FIFO).
- **QUEUED**—returns the number of items remaining in the queue.

Here is an example of using a queue:

```
Example 14.3. Sample Rexx Procedure Using a Queue

/*                                                                            */
/* push/pull WITHOUT multiprogramming support                                 */
/*                                                                            */
push date() time()                                /* push date and time       */
do 1000                                           /* let's pass some time     */
nop                                             /* doing nothing            */
end                                               /* end of loop              */
pull a b                                          /* pull them                */
say "Pushed at " a b ", Pulled at " date() time() /* say now and then         */

/*                                                                            */
/* push/pull WITH multiprogramming support                                   */
/* (no error recovery, or unsupported environment tests)                    */
/*                                                                            */
newq = RXQUEUE("Create")                          /* create a unique queue    */
oq = RXQUEUE("Set",newq)                          /* establish new queue      */
push date() time()                                /* push date and time       */
do 1000                                           /* let's spend some time    */
nop                                             /* doing nothing            */
end                                               /* end of loop              */
pull a b                                          /* get pushed information   */
say "Pushed at " a b ", Pulled at " date() time() /* tell user               */
call RXQUEUE "Delete",newq                   /* destroy unique queue created */
call RXQUEUE "Set",oq               /* reset to default queue (not required) */
```

Special considerations:

- External programs that must communicate with a Rexx procedure through defined data queues can use the Rexx-provided queue or the queue that QUEUE: references (if the external program runs in a child process), or they can receive the data queue name through some interprocess communication technique, including argument passing, placement on a prearranged logical queue, or the use of usual interprocess communication mechanisms (for example, pipes, shared memory, or IPC queues).
• Named queues are available across the entire system. Therefore, the names of queues must be unique within the system. If a queue named anyque exists, using the following function:

```rx
newqueue = RXQUEUE("Create", "ANYQUE")
```
results in an error.

### 14.1.3.3. Multiprogramming Considerations

The top-level Rexx program in a process tree owns an unnamed queue. However, any child process can modify the queue at any time. No specific process or user owns a named queue. The operations that affect the queue are atomic—the subsystem serializes the resource so that no data integrity problems can occur. However, you are responsible for the synchronization of requests so that two processes accessing the same queue get the data in the order it was placed on the queue.

A specific process owns (creates) an unnamed queue. When that process ends, the language processor deletes the queue. Conversely, the named queues created with `RxQueue("Create", queuename)` exist until you explicitly delete them. The end of a program or procedure that created a named queue does not force the deletion of the private queue. When the process that created a queue ends, any data on the queue remains until the data is read or the queue is deleted. (The function call `RxQueue("Delete", queuename)` deletes a queue.)

If a data queue is deleted by its creator, a procedure, or a program, the items in the queue are also deleted.

### 14.1.4. Default Stream Names

A stream name can be a file, a queue, a pipe, or any device that supports character-based input and output. If the stream is a file or device, the name can be any valid file specification.

Windows and Unix-like systems define three default streams:

- stdin (file descriptor 0) - standard input
- stdout (file descriptor 1) - standard output
- stderr (file descriptor 2) - standard error (output)

Rexx provides `.INPUT` and `.OUTPUT` public objects. They default to the default input and output streams of the operating system. The appropriate default stream object is used when the call to a Rexx I/O function includes no stream name. The following Rexx statements write a line to the default output stream of the operating system:

```rx
Lineout(,"Hello World")
.Output~lineout("Hello World")
```

Rexx reserves the names `STDIN`, `STDOUT`, and `STDERR` to allow Rexx functions to refer to these stream objects. The checks for these names are not case-sensitive; for example, `STDIN, stdin, and stdin` all refer to the standard input stream object. If you need to access a file with one of these names, qualify the name with a directory specification, for example, `\stdin`.

Rexx also provides access to arbitrary file descriptors that are already open when Rexx is called. The stream name used to access the stream object is `HANDLE:x`. `x` is the number of the file descriptor you wish to use. You can use `HANDLE:x` as any other stream name; it can be the receiver of a Stream class method. If the value of `x` is not a valid file descriptor, the first I/O operation to that object fails.
Notes:

1. Once you close a `HANDLE:x` stream object, you cannot reopen it.

2. `HANDLE:x` is reserved. If you wish to access a file or device with this name, include a directory specification before the name. For example, `\HANDLE:x` accesses the file HANDLE:x in the current directory.

3. Programs that use the .INPUT and .OUTPUT public objects are independent of the operating environment.

14.1.5. Line versus Character Positioning

Rexx lets you move the read or write position of a persistent stream object to any location within the stream. You can specify this location in terms of characters or lines.

Character positioning is based upon the view of a stream as a simple collection of bytes of data. No special meaning is given to any single character. Character positioning alone can move the stream pointer. For example:

```
MyStream~charin(10,0)
```

moves the stream pointer so that the tenth character in MyStream is the next character read. But this does not return any data. If MyStream is opened for reading or writing, any output that was previously written but is still buffered is eliminated. Moving the write position always causes any buffered output to be written.

Line positioning views a stream as a collection of lines of data. There are two ways of positioning by lines. If you open a stream in binary mode and specify a record length of `x` on the open, a line break occurs every `x` characters. Line positioning in this case is an extension of character positioning. For example, if you open a stream in binary mode with record length 80, then the following two lines are exactly equivalent.

```
MyStream~command(position 5 read line)
MyStream~command(position 321 read char)
```

Remember that streams and other Rexx objects are indexed starting with one rather than zero.

The second way of positioning by lines is for non-binary streams. *Line-end* characters separate lines in non-binary streams. Because the line separator is contained within the stream, ensure accurate line positioning. For example, it is possible to change the line number of the current read position by writing extra line-end characters ahead of the read position or by overwriting existing line-end characters. Thus, line positioning in a non-binary stream object has the following characteristics:

- To do line positioning, it is necessary to read the stream in circumstances such as switching from character methods to line methods or positioning from the end of the stream.

- If you rewrite a stream at a point prior to the read position, the line number of the current read position could become inaccurate.

Note that for both character and line positioning, the index starts with one rather than zero. Thus, character position 1 and line position 1 are equivalent, and both point to the top of the persistent stream object. The Rexx I/O processing uses certain optimizations for positioning. These require that no other process is writing to the stream concurrently and no other program uses or manipulates the
same low-level drive, directory specification, and file name that the language processor uses to open
the file. If you need to work with a stream in these circumstances, use the system I/O functions.

14.2. Implementation

Usually, the dialog between a Rexx program and you as the user takes place on a line-by-
line basis and is, therefore, carried out with the SAY, PULL, or PARSE PULL instructions. This
technique considerably enhances the usability of many programs, because they can be converted to
programmable dialogs by using the external data queue to provide the input you generally type. Use
the PARSE LINEIN instruction only when it is necessary to bypass the external data queue.

When a dialog is not on a line-by-line basis, use the serial interfaces the CHARIN and CHAROUT
methods provide. These methods are important for input and output in transient stream objects, such
as keyboards, printers, or network environments.

Opening and closing of persistent stream objects, such as files, is largely automatic. Generally the first
CHARIN, CHAROUT, CHARs, LINEIN, LINEOUT, or LINES message sent to a stream object opens
that stream object. It remains open until you explicitly close it with a CHAROUT or LINEOUT or until
the program ends. Using the LINEOUT method with only the name of a stream object (and no output
string or line) closes the named stream object. The Stream class also provides OPEN and CLOSE
methods and the COMMAND method, which can explicitly open or close a stream object.

If you open a stream with the CHARIN, CHAROUT, LINEIN, or LINEOUT methods, it is opened for
both reading and writing, if possible. You can use the OPEN method or the COMMAND method to
open a stream for read-only or write-only operations.

14.3. Operating System Specifics

The COMMAND method of the Stream class determines the state of an input or output stream object
and carries out specific operations (see Section 5.2.4.8, “command”). It allows Rexx programs to open
and close selected stream objects for read-only, write-only, or read and write operations, to move the
read and write position within a stream object, to control the locking and buffering characteristics, and
to obtain information (such as the size and the date of the last update).

14.4. Examples of Input and Output

In most circumstances, communication with a user running a Rexx program uses the default input and
output stream objects. For a question and answer dialog, the recommended technique is to use the
SAY and PULL instructions on the .INPUT and .OUTPUT objects. (You can use PARSE PULL if case-
sensitive input is needed.)

It is generally necessary to write to, or read from, stream objects other than the default. For example,
the following program copies the contents of one stream to another.

Example 14.4. FILECOPY program

/* FILECOPY.CMD */
/* This routine copies, as lines, the stream or */
/* file that the first argument names to the stream */
/* or file the second argument names. It is assumed */
/* that the name is not an object, as it could be */
/* if it is passed from another Rexx program. */
Errors during Input and Output
parse arg inputname, outputname
inputobject = .stream~new(inputname)
outputobject = .stream~new(outputname)
signal on notready
do forever
outputobject~lineout(inputobject~linein)
end
exit
notready:
return

As long as lines remain in the named input stream, a line is read and is then immediately written to the
named output stream. This program is easy to change so that it filters the lines before writing them.
The following example illustrates how character and line operations can be mixed in a communications
program. It converts a character stream into lines.

Example 14.5. COLLECT program
/* COLLECT.CMD */
/* This routine collects characters from the stream */
/* the first argument names until a line is
*/
/* complete, and then places the line on the
*/
/* external data queue.
*/
/* The second argument is a single character that
*/
/* identifies the end of a line.
*/
parse arg inputname, lineendchar
inputobject = .stream~new(inputname)
buffer=""
/* zero-length character accumulator */
do forever
nextchar=inputobject~charin
if nextchar=lineendchar then leave
buffer=buffer||nextchar
/* add to buffer */
end
queue buffer /* place it on the external data queue */

Here each line is built up in a variable called BUFFER. When the line is complete (for example, when
the user presses the Enter key) the loop ends and the language processor places the contents of
BUFFER on the external data queue. The program then ends.

14.5. Errors during Input and Output
The Rexx language offers considerable flexibility in handling errors during input or output. This is
provided in the form of a NOTREADY condition that the CALL ON and SIGNAL ON instructions can
trap. The STATE and DESCRIPTION methods can elicit further information.
When an error occurs during an input or output operation, the function or method called usually
continues without interruption (the output method returns a nonzero count). Depending on the nature
of the operation, a program has the option of raising the NOTREADY condition. The NOTREADY
condition is similar to the ERROR and FAILURE conditions associated with commands in that it does
626


not cause a terminating error if the condition is raised but is not trapped. After NOTREADY has been raised, the following possibilities exist:

- If the NOTREADY condition is not trapped, processing continues without interruption. The NOTREADY condition remains in the OFF state.

- If SIGNAL ON NOTREADY traps the NOTREADY condition, the NOTREADY condition is raised. Processing of the current clause stops immediately, and the SIGNAL takes place as usual for condition traps.

- If CALL ON NOTREADY traps the NOTREADY condition, the NOTREADY condition is raised, but execution of the current clause is not halted. The NOTREADY condition is put into the delayed state, and processing continues until the end of the current clause. While processing continues, input methods that refer to the same stream can return the null string and output methods can return an appropriate count, depending on the form and timing of the error. At the end of the current clause, the CALL takes place as usual for condition traps.

- If the NOTREADY condition is in the DELAY state (CALL ON NOTREADY traps the NOTREADY condition, which has already been raised), processing continues, and the NOTREADY condition remains in the DELAY state.

After the NOTREADY condition has been raised and is in DELAY state, the "0" option of the CONDITION function returns the stream object being processed when the stream error occurred.

The STATE method of the Stream class returns the stream object state as ERROR, NOTREADY, or UNKNOWN. You can obtain additional information by using the DESCRIPTION method of the Stream class.

**Note**

SAY .OUTPUT and PULL .INPUT never raise the NOTREADY condition. However, the STATE and DESCRIPTION methods can return NOTREADY.

### 14.6. Summary of Rexx I/O Instructions and Methods

The following lists Rexx I/O instructions and methods:

- **Stream method** charIn
- **Stream method** charOut
- **Stream method** chars
- **Stream method** close
- **Stream method** command
- **Stream method** description
- **Stream method** flush
- **Stream method** lineIn
- **Stream method** lineOut
- **Stream method** lines
- **Stream method** makeArray
- **Stream method** open
- instruction **PARSE LINEIN**
- instruction **PARSE PULL**
- **Stream method** position
Summary of Rexx I/O Instructions and Methods

• instruction **PULL**
• instruction **PUSH**
• Stream method **qualify**
• Stream method **query**
• instruction **QUEUE**
• built-in function **QUEUED**
• instruction **SAY**
• Stream method **seek**
• Stream method **state**
**Chapter 15.**

**Debugging Aids**

In addition to the `TRACE` instruction there are the following debugging aids.

### 15.1. Interactive Debugging of Programs

The debug facility permits interactively controlled execution of a program. Adding the prefix character `?` to the `TRACE` instruction, the `TRACE` function, or `TRACE` keyword of the `::OPTIONS` directive. For example, `TRACE ?I, TRACE(?I), or ::OPTIONS TRACE ?I` turns on interactive debugging and indicates to the user that interactive debugging is active. Further `TRACE` instructions in the program are ignored, and the language processor pauses after nearly all instructions that are traced at the console (see Section 15.2, “Debugging Aids” for the exceptions). When the language processor pauses, the following debug actions are available:

- Entering a null line causes the language processor to continue with the execution until the next pause for debugging input. Repeatedly entering a null line, therefore, steps from pause point to pause point. For `TRACE ?A`, for example, this is equivalent to single-stepping through the program.

- Entering an equal sign (=) with no surrounding whitespace causes the language processor to reexecute the clause last traced. For example, if an IF clause is about to take the wrong branch, you can change the value of the variables on which it depends, and then reexecute it.

Once the clause has been reexecuted, the language processor pauses again.

- Anything else entered is treated as a line of one or more clauses, and processed immediately (that is, as though DO; line; END; had been inserted in the program). The same rules apply as for the `INTERPRET` instruction (for example, DO-END constructs must be complete). If an instruction contains a syntax error, a standard message is displayed and you are prompted for input again. Similarly, all other `SIGNAL` conditions are disabled while the string is processed to prevent unintentional transfer of control.

During interpretation of the string, no tracing takes place, except that nonzero return codes from commands are displayed. The special variable `RC` and the environment symbol `.RS` are not set by commands executed from the string. Once the string has been processed, the language processor pauses again for further debugging input.

Interactive debug is turned off in either of the following cases:

- A `TRACE` instruction uses the `?` prefix while interactive debug is in effect
- At any time, if `TRACE 0` or `TRACE` with no options is entered

### 15.2. Debugging Aids

The numeric form of the `TRACE` instruction can be used to allow sections of the program to be executed without pause for debugging input. `TRACE n` (that is, a positive result) allows execution to continue, skipping the next `n` pauses (when interactive debugging is or becomes active). `TRACE -n` (that is, a negative result) allows execution to continue without pause and with tracing inhibited for `n` clauses that would otherwise be traced. The trace action a `TRACE` instruction selects is saved and restored across subroutine calls. This means that if you are stepping through a program (for example, after using `TRACE ?R` to trace results) and then enter a subroutine in which you have no interest, you can enter `TRACE 0` to turn off tracing. No further instructions in the subroutine are traced, but on return to the caller, tracing is restored.
Similarly, if you are interested only in a subroutine, you can put a trace \texttt{?R} instruction at its start. Having traced the routine, the original status of tracing is restored and, if tracing was off on entry to the subroutine, tracing and interactive debugging are turned off until the next entry to the subroutine.

Because any instructions can be executed in interactive debugging you have considerable control over the execution.

The following are some examples:

<table>
<thead>
<tr>
<th>Example 15.1. Character inout and output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Say expr /* displays the result of evaluating the */ expression */</td>
</tr>
<tr>
<td>name=expr /* alters the value of a variable */</td>
</tr>
<tr>
<td>Trace O /* (or Trace with no options) turns off */ interactive debugging and all tracing */</td>
</tr>
<tr>
<td>Trace ?A /* turns off interactive debugging but */ continues tracing all clauses */</td>
</tr>
<tr>
<td>exit /* terminates execution of the program */</td>
</tr>
<tr>
<td>do i=1 to 10; say stem.i; end /* displays ten elements of the array stem. */</td>
</tr>
</tbody>
</table>

Exceptions: Some clauses cannot safely be reexecuted, and therefore the language processor does not pause after them, even if they are traced. These are:

- Any repetitive DO clause, on the second or subsequent time around the loop.
- All END clauses.
- All THEN, ELSE, OTHERWISE, or null clauses.
- All RETURN and EXIT clauses.
- All SIGNAL clauses (but the language processor pauses after the target label is traced).
- Any clause that causes a syntax error. They can be trapped by SIGNAL ON SYNTAX, but cannot be reexecuted.

A pause occurs after a REPLY instruction, but the REPLY instruction cannot be reexecuted.

\section*{15.3. RXTRACE Variable}

When the interpreter starts, it checks the setting of the special environment variable, \texttt{RXTRACE}. If \texttt{RXTRACE} has been set to \texttt{ON} (not case-sensitive), each time when starting a new Rexx procedure, the interpreter starts in interactive debug mode as if the Rexx instruction \texttt{TRACE \ 'R'} had been the first interpretable instruction. All other settings of \texttt{RXTRACE} are ignored. \texttt{RXTRACE} is only checked when starting a new Rexx interpreter, it is not checked each time when starting a new Rexx procedure.

Use the \texttt{SET} command to set or query an environment variable or query all environment variables. To delete an environment variable, use \texttt{SET variable=}. 

630
Reserved Keywords

Keywords can be used as ordinary symbols in many unambiguous situations. The precise rules are given in this chapter.

The free syntax of Rexx implies that some symbols are reserved for use by the language processor in certain contexts.

Within particular instructions, some symbols can be reserved to separate the parts of the instruction. These symbols are referred to as keywords. Examples of Rexx keywords are the WHILE keyword in a DO instruction and the THEN keyword, which acts as a clause terminator in this case, following an IF or WHEN clause.

Apart from these cases, only simple symbols that are the first token in a clause and that are not followed by an "=" or ":" are checked to see if they are instruction keywords. The symbols can be freely used elsewhere in clauses without being understood as keywords.

Be careful with host commands or subcommands with the same name as Rexx keywords. To avoid problems, enclose at least the command or subcommand in quotation marks. For example:

"DELETE" Fn"."Ext

You can then also use the SIGNAL ON NOVALUE condition to check the integrity of an executable.

Alternatively, you can precede such command strings with two adjacent quotation marks to concatenate the null string to the beginning. For example:

""Erase Fn"."Ext

A third option is to enclose the entire expression, or the first symbol, in parentheses. For example:

(Erase Fn"."Ext)
Special Variables

A special variable can be set automatically during processing of a Rexx program. There are five special variables:

RC

is set to the return code from any executed command (including those submitted with the ADDRESS instruction). After the trapping of ERROR or FAILURE conditions, it is also set to the command return code. When the SYNTAX condition is trapped, RC is set to the syntax error number (1-99). RC is unchanged when any other condition is trapped.

Note

Commands executed manually during interactive tracing do not change the value of RC.

RESULT

is set by a RETURN instruction in a subroutine that has been called, or a method that was activated by a message instruction, if the RETURN instruction specifies an expression. If the RETURN instruction has no expression, RESULT is dropped (becomes uninitialized). Note that an EXIT or REPLY instruction also sets RESULT.

SELF

is set when a method is activated. Its value is the object that forms the execution context for the method (that is, the receiver object of the activating message). You can use SELF to:

• Run a method in an object in which a method is already running. For example, a Find_Clues method is running in an object called Mystery_Novel. When Find_Clues finds a clue, it sends a Read_Last_Page message to Mystery_Novel:

self~Read_Last_Page

• Pass references about an object to the methods of other objects. For example, a Sing method is running in object Song. The code Singer2~Duet(self) would give the Duet method access to the same Song.

SIGL

is set to the line number of the last instruction that caused a transfer of control to a label (that is, any SIGNAL, CALL, internal function call, or trapped condition). See Section 11.3.4, “The Special Variable SIGL”.

SUPER

is set when a method is activated. Its value is the class object that is the usual starting point for a superclass method lookup for the SELF object. This is the first immediate superclass of the class that defined the method currently running. (See Section 1.7, “Classes and Instances”) If the current method was defined by a class in the direct inheritance chain, SUPER will always refer to the immediate superclass of that class. If the current method is defined by a mixin class, the SUPER variable will always be the superclass of the mixin class.

The special variable SUPER lets you call a method in the superclass of an object. For example, the following Savings class has INIT methods that the Savings class, Account class, and Object class define.
Example 17.1. Using the SUPER variable

```ruby
::class Account
::method INIT
  expose balance
  use arg balance
  self~init:super /* Forwards to the Object INIT method */
::method TYPE
  return "an account"
::method name attribute
::class Savings subclass Account
::method INIT
  expose interest_rate
  use arg balance, interest_rate
  self~init:super(balance) /* Forwards to the Account INIT method */
::method type
  return "a savings account"
```

When the INIT method of the Savings class is called, the variable SUPER is set to the Account class object. The instruction:

```
self~init:super /* Forwards to the Object INIT method */
```

calls the INIT method of the Account class rather than recursively calling the INIT method of the Savings class. When the INIT method of the Account class is called, the variable SUPER is assigned to the Object class.

Example 17.2. Forwarding an event

```
self~init:super /* Forwards to the Object INIT method */
```

calls the INIT method that the Object class defines.

You can alter these variables like any other variable, but the language processor continues to set RC, RESULT, and SIGL automatically when appropriate. The EXPOSE, PROCEDURE, USE and DROP instructions also affect these variables.

Rexx also supplies functions that indirectly affect the execution of a program. An example is the name that the program was called by and the source of the program (which are available using the PARSE SOURCE instruction). In addition, PARSE VERSION makes available the language version and date of Rexx implementation that is running. The built-in functions ADDRESS, DIGITS, FUZZ, FORM, and TRACE return other settings that affect the execution of a program.
Useful Services
The following section describes useful commands and services.

18.1. Windows Commands
COPY
  copies files.
DELETE
  deletes files.
DIR
  displays disk directories.
ERASE
  erases files.
MODE
  controls input and output device characteristics.
PATH
  defines or displays the search path for commands and Rexx programs. See also Section 7.2.1, “Search Order”.
SET
  displays or changes Windows environment variables. See also Section 7.4.71, “VALUE”.

18.2. Linux Commands
Most Commonly used commands are:

cp
  copies files and directories.
mv
  moves files and directories.
rm
  removes files and directories.
ls
  displays files and directories.
echo $path
  defines or displays the search path for commands and Rexx programs. See also Section 7.2.1, “Search Order”.
env
  displays or changes Linux environment variables.

Any other Linux command can be used. For a description of these commands, see the respective Linux documentation (for example, man-pages).
18.3. Subcommand Handler Services

For a complete subcommand handler description, see the Open Object Rexx: Application Programming Interfaces.

18.3.1. The RXSUBCOM Command

The RXSUBCOM command registers, drops, and queries Rexx subcommand handlers. A Rexx procedure or script file can use RXSUBCOM to register dynamic-link library subcommand handlers. Once the subcommand handler is registered, a Rexx program can send commands to the subcommand handler with the Rexx ADDRESS instruction. For example, Rexx Dialog Manager programs use RXSUBCOM to register the ISPCIR subcommand handler.

Example 18.1. Command RXSUBCOM — REGISTER option

"RXSUBCOM REGISTER ISPCIR ISPCIR ISPCIR"
Address ispcir

See Section 2.1, “ADDRESS” for details of the ADDRESS instruction.

18.3.1.1. RXSUBCOM REGISTER

RXSUBCOM REGISTER registers a dynamic-link library subcommand handler. This command makes a command environment available to Rexx.

RXSUBCOM REGISTER enname dllname procname

Parameters:

envname
The subcommand handler name. The Rexx ADDRESS instruction uses envname to send commands to the subcommand handler.

dllname
The name of the dynamic-link library file containing the subcommand handler routine.

procname
The name of the dynamic-link library procedure within dllname that Rexx calls as a subcommand handler.

Return codes:

0
The command environment has been registered.

10
A duplicate registration has occurred. An envname subcommand handler in a different dynamic-link library has already been registered. Both the new subcommand handler and the existing subcommand handler can be used.

30
The registration has failed. Subcommand handler envname in library dllname is already registered.
RXSUBCOM was unable to obtain the memory necessary to register the subcommand handler.

-1
A parameter is missing or incorrectly specified.

18.3.1.2. RXSUBCOM DROP

RXSUBCOM DROP deregisters a subcommand handler.

Parameters:

envname
The name of the subcommand handler.

dllname
The name of the dynamic-link file containing the subcommand handler routine.

Return codes:

0
The subcommand handler was successfully deregistered.

30
The subcommand handler does not exist.

40
The environment was registered by a different process as RXSUBCOM_NONDROP.

-1
A parameter is missing or specified incorrectly.

18.3.1.3. RXSUBCOM QUERY

RXSUBCOM QUERY checks the existence of a subcommand handler. The query result is returned.

Parameters:

envname
The name of the subcommand handler.

dllname
The name of the dynamic-link file containing the subcommand handler routine.

Return codes:

0
The subcommand handler is registered.
The subcommand handler is not registered.

-1
A parameter is missing or specified incorrectly.

18.3.1.4. RXSUBCOM LOAD

RXSUBCOM LOAD loads a subcommand handler dynamic-link library.

Parameters:

`envname`
- The name of the subcommand handler.

`dllname`
- The name of the dynamic-link file containing the subcommand handler routine.

Return codes:

0
The dynamic-link library was located and loaded successfully.

50
The dynamic-link library was not located or could not be loaded.

-1
A parameter is missing or incorrectly specified.

18.4. The RXQUEUE Filter

The RXQUEUE filter usually operates on the default queue named `SESSION`. However, if an environment variable named RXQUEUE exists, the RXQUEUE value is used for the queue name.

For a full description of Rexx queue services for applications programming, see Section 14.1.3, "External Data Queue".

Parameters:

`queue name`
- The name of the named Rexx queue to stack or queue items to. If no `queue name` is given, `SESSION` is used.
The RXQUEUE Filter

/FIFO
queues items from STDIN first in, first out (FIFO) on the Rexx queue. This is the default.

/LIFO
stacks items from STDIN last in, first out (LIFO) on the Rexx queue.

/CLEAR
removes all lines from the Rexx queue.

RXQUEUE takes output lines from another program and places them on a Rexx queue. There is currently a limit of 65472 characters for a single line. If a line contains more characters than the limit, those characters are discarded.

A Rexx procedure can use RXQUEUE to capture operating system command and program output for processing. RXQUEUE can direct output to any Rexx queue, either FIFO (first in, first out) or LIFO (last in, first out).

RXQUEUE uses the environment variable RXQUEUE for the default queue name. When RXQUEUE does not have a value, RXQUEUE uses SESSION for the queue name.

The following example obtains the Windows version number with RXQUEUE:

Example 18.2. Command RXQUEUE

```rexx
/* Sample program to show simple use of RXQUEUE */
/* Find out the Windows version number, using the */
/* VER command. VER produces two lines of */
/* output; one blank line, and one line with the*/
/* format "The Windows Version is n.nn" */
"VER |RXQUEUE" /* Put the data on the Queue */
pull . /* Get and discard the blank line */
Pull . "VERSION" number "]" /* The bracket is required for Windows 95, not for Windows NT */
Say "We are running on Windows Version" number
```

Note that the syntax of the version string that is returned by Windows can vary, so the parsing syntax for retrieving the version number may be different.

The following example processes output from the DIR command:

Example 18.3. Command RXQUEUE

```rexx
/* Sample program to show how to use the RXQUEUE filter */
/* This program filters the output from a DIR command, */
/* ignoring small files. It displays a list of the */
/* large files, and the total of the sizes of the large */
/* files. */
size_limit = 10000 /* The dividing line */
/* between large and small*/
size_total = 0 /* Sum of large file sizes*/
NUMERIC DIGITS 12 /* Set up to handle very */
/* large numbers */
/* Create a new queue so that this program cannot */
/* interfere with data placed on the queue by another */
/* program. */
```
queue_name = rxqueue("Create")
Call rxqueue "Set", queue_name

"DIR /N | RXQUEUE" queue_name

Do 5
   Pull .  /* discard header line */
End

/* Now all the lines are file or directory lines, */
/* except for one at the end. */

Do queued() - 1  /* loop for lines we want */
   Parse Pull . size . name . /* get one name and size */
   /* If the size field says "<DIR>", we ignore this */
   /* line. */
   If size <> "<DIR>" Then
      /* Now check size, and display */
      If size > size_limit Then Do
         Say format(size,12) name
         size_total = size_total + size
      End
   End
End

Say "The total size of those files is" size_total

/* Now we are done with the queue. We delete it, which */
/* discards the line remaining in it. */

Call rxqueue "DELETE", queue_name

18.5. Distributing Programs without Source

Open Object Rexx comes with a utility called rexxc. You can use this utility to produce versions of your programs that do not include the original program source. You can use these programs to replace any Rexx program file that includes the source, with the following restrictions:

1. The SOURCELINE built-in function returns 0 for the number of lines in the program and raises an error for all attempts to retrieve a line.

2. A sourceless program may not be traced. The TRACE instruction runs without error, but no tracing of instruction lines, expression results, or intermediate expression values occurs.

The syntax of the rexxc utility is:

```
rexxc inputfile outputfile -s -e
```

If you specify the outputfile, the language processor processes the inputfile and writes the executable version of the program to the outputfile. If the outputfile already exists, it is replaced.

If the language processor detects a syntax error while processing the program, it reports the error and stops processing without creating a new output file. If you omit the outputfile, the language processor performs a syntax check on the program without writing the executable version to a file.

You can use the -s option to suppress the display of the information about the interpreter used, and the -e option to create the outputfile in base64-encoded format, suitable when a 7-bit encoding is required.
On Windows, the `-s` and `-e` options can alternatively be specified as `/s` and `/e`.

**Note**

You can use the in-storage capabilities of the RexxStart programming interface to process the file image of the output file.
Appendix A. Using DO and LOOP

This appendix provides you with additional information about the DO and LOOP keyword instructions.

A.1. Simple DO Group

If you specify neither repetitor nor conditional, the DO construct only groups a number of instructions together. They are processed once. For example:

Example A.1. Simple DO block

/* The two instructions between DO and END are both */
/* processed if A has the value "3". */
If a=3 then Do
  a=a+2
  Say "Smile!"
End

A.2. Repetitive Loops

If a DO or LOOP instruction has a repetitor phrase, a conditional phrase, or both, the group of instructions forms a repetitive loop. The instructions are processed according to the repetitor phrase, optionally modified by the conditional phrase. (See Section A.3, "Conditional Phrases (WHILE and UNTIL)".)

A.2.1. Simple Repetitive Loops

A simple repetitive loop is a repetitive loop in which the repetitor phrase is an expression that evaluates to a count of the iterations.

If repetitor is omitted but there is a conditional or if the repetitor is FOREVER, the group of instructions is processed until the condition is satisfied or a Rexx instruction ends the loop (for example, LEAVE).

In the simple form of a repetitive loop, expr is evaluated immediately (and must result in a positive whole number or zero), and the loop is then processed that many times.

Example A.2. Simple LOOP block

/* This displays "Hello" five times */
Loop 5
  say "Hello"
end

Note that, similar to the distinction between a command and an assignment, if the first token of expr is a symbol and the second token is (or starts with) =, the controlled form of repetitor is expected.

A.2.2. Controlled Repetitive Loops

The controlled form specifies control1, a control variable that is assigned an initial value (the result of expr, formatted as though 0 had been added) before the first execution of the instruction list.
variable is then stepped by adding the result of exprb before the second and subsequent times that the instruction list is processed.

The instruction list is processed repeatedly as long as the end condition (determined by the result of exprt) is not met. If exprb is positive or 0, the loop is ended when control1 is greater than exprt. If negative, the loop is ended when control1 is less than exprt.

The expri, exprt, and exprb options must result in numbers. They are evaluated only once, before the loop begins and before the control variable is set to its initial value. The default value for exprb is 1. If exprt is omitted, the loop runs infinitely unless some other condition stops it.

---

**Example A.3. Simple LOOP block with conditions**

```plaintext
Loop I=3 to -2 by -1 /* Displays: */
say i /* 3 */
   /*  2 */
end /*  1 */
   /*  0 */
   /* -1 */
   /* -2 */
```

The numbers do not have to be whole numbers:

**Example A.4. Simple LOOP block with conditions**

```plaintext
I=0.3 /* Displays: */
Do Y=I to I+4 by 0.7 /* 0.3 */
say Y /* 1.0 */
   /* 1.7 */
end /* 2.4 */
   /* 3.1 */
   /* 3.8 */
```

The control variable can be altered within the loop, and this can affect the iteration of the loop. Altering the value of the control variable is not considered good programming practice, though it can be appropriate in certain circumstances.

Note that the end condition is tested at the start of each iteration (and after the control variable is stepped, on the second and subsequent iterations). Therefore, if the end condition is met immediately, the group of instructions can be skipped entirely. Note also that the control variable is referred to by name. If, for example, the compound name A.I is used for the control variable, altering I within the loop causes a change in the control variable.

The execution of a controlled loop can be limited further by a FOR phrase. In this case, you must specify exprf, and it must evaluate to a positive whole number or zero. This acts like the repetition count in a simple repetitive loop, and sets a limit to the number of iterations around the loop if no other condition stops it. Like the TO and BY expressions, it is evaluated only once—when the instruction is first processed and before the control variable receives its initial value. Like the TO condition, the FOR condition is checked at the start of each iteration.

**Example A.5. Simple LOOP block with limits**

```plaintext
Loop Y=0.3 to 4.3 by 0.7 for 3 /* Displays: */
say Y /* 0.3 */
   /* 1.0 */
end /* 1.0 */
```
In a controlled loop, the *control1* name describing the control variable can be specified on the END clause. This *name* must match *control1* in the DO or LOOP clause in all respects except the case (note that no substitution for compound variables is carried out). Otherwise, a syntax error results. This enables the nesting of loops to be checked automatically, with minimal overhead.

**Example A.6. Simple LOOP block with limits**

```plaintext
Loop K=1 to 10
  ...
End k  /* Checks that this is the END for K loop */
```

**Note**

The NUMERIC settings can affect the successive values of the control variable because Rexx arithmetic rules apply to the computation of stepping the control variable.

### A.2.3. Repetitive Loops over Collections

A collection loop specifies a control variable, *control2*, which receives a different value on each repetition of the loop. (For more information on *control2*, see Section 2.4, “DO”.) These different values are taken from successive values of *collection*. The *collection* is any expression that evaluates to an object that provides a `makeArray` method, including stem variables. The collection returned determines the set of values and their order. Array, List, and Queue items return an array with the items in the appropriate order, as do Streams. Tables, Stems, Directories, etc. are not ordered so the items get placed in the array in no particular order.

If the collection is a stem variable, the values are the tail names that have been explicitly assigned to the given stem. The order of the tail names is unspecified, and a program should not rely on any order.

For other collection objects, the `makeArray` method of the specific collection class determines the values assigned to the control variable.

All values for the loop iteration are obtained at the beginning of the loop. Therefore, changes to the target collection object do not affect the loop iteration. For example, using DROP to change the set of tails associated with a stem or using a new value as a tail does not change the number of loop iterations or the values over which the loop iterates.

As with controlled repetition, you can specify the symbol that describes the control variable on the END clause. The control variable is referenced by name, and you can change it within the loop (although this would not usually be useful). You can also specify the control variable name on an ITERATE or LEAVE instruction.

**Example A.7. Simple LOOP block over a collection**

```plaintext
Astem.=0
Astem.3="CCC"
Astem.24="XXX"
```
A.2.4. Repetitive Loops over Suppliers

A supplier loop specifies one or two control variables, `index`, and `item`, which receive a different value on each repetition of the loop. (For more information on `index` and `item`, see Section 2.4, “DO”.) These different values are taken from successive values of `supplier`. The `supplier` is any expression that evaluates to an object that provides a `supplier` method. The supplier returned determines the set of values and their order.

All values for the loop iteration are obtained at the beginning of the loop. Therefore, changes to the target object do not affect the loop iteration.

If you want to specify a `name` on the END clause or on an ITERATE or LEAVE instruction, you have to specify the LABEL keyword.

Example A.8. LOOP WITH over a supplier

```
loop with index name item method over .numericComparator-methods(.nil)
    say name method-isPrivate method-isProtected method-isGuarded
end
```

This example can produce:

```
COMPARE 0 0 1
INIT 0 0 1
```

A.3. Conditional Phrases (WHILE and UNTIL)

A conditional phrase can modify the iteration of a repetitive loop. It can cause the termination of a loop. It can follow any of the forms of `repetitor` (none, FOREVER, simple, controlled, OVER, or WITH). If you specify WHILE or UNTIL, `exprw` or `expru`, respectively, is evaluated after each loop using the latest values of all variables, and the loop is ended if `exprw` evaluates to 0 or `expru` evaluates to 1.

For a WHILE loop, the condition is evaluated at the top of the group of instructions. For an UNTIL loop, the condition is evaluated at the bottom—before the control variable has been stepped.
Example A.9. Simple LOOP block with conditions

```
Loop I=1 to 10 by 2 until i>6
    say i
end
/* Displays: "1" "3" "5" "7" */
```

Note

Using the LEAVE or ITERATE instructions can also modify the execution of repetitive loops.

A.4. LABEL Phrase

The LABEL phrase may be used to specify a name for the DO or LOOP. The label can optionally be used for

- a LEAVE instruction to identify the block to be left
- an ITERATE instruction to identify the loop to be iterated
- the END clause of the block, for additional checking.

Example A.10. Simple LOOP block with label

```
Loop label outer I=1 to 10 by 2
    ... if i > 5 then do label inner
        ... if a = b then leave inner
        ... if c = b then iterate outer
    end inner
    ... say i
end outer
/* Displays: "1" "3" "5" "7" */
```

In this example, the LEAVE instruction will exit the inner DO block. The ITERATE instruction will iterate the outer LOOP instruction.

If a LABEL phrase is used on a DO or LOOP, it overrides any name derived from any control variable name. That is, if label is used, the control variable cannot be used on a LEAVE, ITERATE, or END instruction to identify the block. Only the label name is valid.

A.5. COUNTER Phrase

The COUNTER phrase may be used on repetitive DO or LOOP instructions to specify a ctr variable that provides a count value for each repetition. ctr starts with 1 and increases by 1 for each repetition.
After the loop finishes, \( c \) will have the count value of the last repetition, or, if the loop had zero repetitions, \( c \) will be 0.

Example A.11. Controlled DO loop with COUNTER

```plaintext
do counter ct x = -0.3 to 0.4 by 0.2
    say ct x
end
/* Displays: 1 -0.3, 2 -0.1, 3 0.1, 4 0.3 */
```
A.6. Conceptual Model of Loops

Evaluate `exprr+0` or `exprr+1` and then `exprt+0`, `exprb+0`, and `exprf+0` in order written.

Assign start value to control variable.

Use `TO` value (`exprt`) to test control variable for termination.

Use count of iterations (`exprr`) to test for termination.

Use `FOR` value (`exprf`) to test for termination.

Use `WHILE` expression (`expw`) to test for termination.

Process instructions in the `DO/LOOP` group.

Use `UNTIL` expression (`expru`) to test for termination.

Use `BY` value (`exprb`) to update control variable.

Discontinue processing of `DO/LOOP` group if `TO` value is exceeded.

Discontinue processing of `DO/LOOP` group if number of iterations is exceeded.

Discontinue processing of `DO/LOOP` group if `FOR` value is exceeded.

Discontinue processing of `DO/LOOP` group if `WHILE` condition is not met.

Discontinue processing of `DO/LOOP` group if `UNTIL` condition is met.

Figure A.1. Concept of a Loop
Figure A.2. Concept of Repetitive Loop over Collection
Appendix B. Migration

This appendix lists some differences between Open Object Rexx 5.0 and earlier versions of Open Object Rexx.

B.1. Incompatible ooRexx features

Functions or features which have been changed in ooRexx 5.0 in a way that will lead to incompatibilities with prior versions.

B.1.1. RexxUtil SysTempFileName

SysTempFileName on Unix-like platforms now behaves identically to SysTempFileName on Windows. It no longer uses only the first five characters of the file name part of the template, appending a random string to make it unique. It also no more prepends an operating system-chosen writable path if no path is given in the template. Instead it now uses the same filler-based mechanism as the Windows version does.

Existing ooRexx programs using SysTempFileName on Unix-like platforms will need to be amended.

B.2. Deprecated Rexx features

In exceptional circumstances, ooRexx may deprecate functions or features, which means the use of these functions is discouraged, documentation is no longer provided, and bug reports against deprecated functions will not be accepted. Reasons for deprecation include broken functionality, features that were never officially documented, or functions no longer required.

Although existing code using deprecated functions is expected to continue to work as-is, you are strongly encouraged to migrate to the replacement functionality, as these functions may be removed from future releases of the interpreter.

B.2.1. RexxUtil Semaphore functions

The following RexxUtil functions related to semaphore-processing have been deprecated.

SysCloseEventSem  SysPostEventSem
SysCloseMutexSem  SysPulseEventSem
SysCreateEventSem  SysReleaseMutexSem
SysCreateMutexSem  SysRequestMutexSem
SysOpenEventSem  SysResetEventSem
SysOpenMutexSem  SysWaitEventSem

They have been superseded by the new EventSemaphore Class and MutexSemaphore Class.

B.2.2. RexxUtil SysLoadFuncs/SysDropFuncs

The RexxUtil functions SysLoadFuncs and SysDropFuncs have been deprecated. Since ooRexx 4.0.0 these functions do nothing.

B.2.3. ::OPTIONS NOVALUE ERROR directive

The ::OPTIONS NOVALUE ERROR directive has been deprecated. It was renamed to ::OPTIONS NOVALUE SYNTAX.
B.2.4. Class ArgUtil
The ArgUtil class and all of it methods have been deprecated. It has been superseded by Validate Class.
Appendix C. Error Numbers and Messages

The error numbers produced by syntax errors during the processing of Rexx programs are all in the range 1 to 99. Errors are raised in response to conditions, for example, SYNTAX, NOMETHOD, and PROPAGATE. When the condition is SYNTAX, the value of the error number is placed in the variable RC when SIGNAL ON SYNTAX is trapped.

You can use the ERRORTEXT built-in function to return the text of an error message.

Some errors have associated subcodes. A subcode is a one- to three-digit decimal extension to the error number, for example, 115 in 40.115. When an error subcode is available, additional information that further defines the source of the error is given. The ERRORTEXT built-in function cannot retrieve the secondary message, but it is available from the condition object created when SIGNAL ON SYNTAX traps an error.

C.1. Error List

C.1.1. Error 3 - Failure during initialization.

Explanation:

The REXX program could not be read from the disk.

The associated subcodes are:

001 Failure during initialization: File "filename" is unreadable.

900 message.

901 Failure during initialization: Program "program" was not found.

902 Error writing output file "file".

903 Program "program_name" cannot be run by this version of the REXX interpreter.

905 Encoded compiled program "program_name" cannot be decoded.

C.1.2. Error 4 - Program interrupted.

Explanation:

The system interrupted the execution of your program because of an error or a user request.

The associated subcodes are:

001 Program interrupted with condition condition.
C.1.3. Error 5 - System resources exhausted.

Explanation:
While trying to execute a program, the language processor was unable to get the resources it needed to continue. For example, it could not get the space needed for its work areas or variables. The program that called the language processor might itself have already used up most of the available storage. Or a request for storage might have been for more than the implementation maximum.

The associated subcodes are:

900 message.

C.1.4. Error 6 - Unmatched "/*" or quote.

Explanation:
A comment or literal string was started but never finished. This could be because the language processor detected:
- The end of the program (or the end of the string in an INTERPRET instruction) without finding the ending "*/" for a comment or the ending quotation mark for a literal string
- The end of the line for a literal string.

The associated subcodes are:

001 Unmatched comment delimiter ("/*") on line line_number.
002 Unmatched single quote (').
003 Unmatched double quote ("").
900 message.

C.1.5. Error 7 - WHEN or OTHERWISE expected.

Explanation:
At least one WHEN construct (and possibly an OTHERWISE clause) is expected within a SELECT instruction. This message is issued if any other instruction is found or there is no WHEN construct before the OTHERWISE or all WHEN expressions are false and an OTHERWISE is not present. A common cause of this error is if you forget the DO and END around the list of instructions following a WHEN. For example:

<table>
<thead>
<tr>
<th>WRONG</th>
<th>RIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select</td>
<td>Select</td>
</tr>
<tr>
<td>When a=c then</td>
<td>When a=c then DO</td>
</tr>
<tr>
<td>Say 'A equals C'</td>
<td>Say 'A equals C'</td>
</tr>
<tr>
<td>exit</td>
<td>exit</td>
</tr>
</tbody>
</table>
The associated subcodes are:

001
   SELECT on line line_number requires WHEN.

002
   SELECT on line line_number requires WHEN, OTHERWISE, or END.

003
   All WHEN expressions of SELECT are false; OTHERWISE expected.

C.1.6. Error 8 - Unexpected THEN or ELSE.
Explanation:
A THEN or an ELSE clause was found that does not match a corresponding IF or WHEN clause. This often occurs because of a missing END or DO...END in the THEN part of a complex IF...THEN...ELSE construction. For example:

<table>
<thead>
<tr>
<th>WRONG</th>
<th>RIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>if a = b then do</td>
<td>if a = b then do</td>
</tr>
<tr>
<td>say &quot;equals&quot;</td>
<td>say &quot;equals&quot;</td>
</tr>
<tr>
<td>exit</td>
<td>exit</td>
</tr>
<tr>
<td>else</td>
<td>end</td>
</tr>
<tr>
<td>say &quot;not equals&quot;</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>say &quot;not equals&quot;</td>
</tr>
</tbody>
</table>

The associated subcodes are:

001
   THEN has no corresponding IF or WHEN clause.

002
   ELSE has no corresponding THEN clause.

C.1.7. Error 9 - Unexpected WHEN or OTHERWISE.
Explanation:
A WHEN or OTHERWISE was found outside of a SELECT construction. You might have accidentally enclosed the instruction in a DO...END construction by leaving out an END, or you might have tried to branch to it with a SIGNAL instruction (which does not work because the SELECT is then ended).

The associated subcodes are:

001
   WHEN has no corresponding SELECT.

002
   OTHERWISE has no corresponding SELECT.

C.1.8. Error 10 - Unexpected or unmatched END.
Explanation:
More ENDS were found in your program than DO, LOOP, or SELECT instructions, or the ENDS did not match the DO, LOOP, or SELECT instructions. This message also occurs if you try to transfer control into the middle of a loop using SIGNAL. In this case, the language processor does not expect the END because it did not process the previous DO instruction. Remember also that SIGNAL deactivates any current loops, so it cannot transfer control from one place inside a loop to another.

Another cause for this message is placing an END immediately after a THEN or ELSE subkeyword or specifying a name on the END keyword that does not match the name following DO or LOOP. Putting the name of the control variable on ENDS that close repetitive loops can also help locate this kind of error.

The associated subcodes are:

001  END has no corresponding DO, LOOP, or SELECT.

002  Symbol following END ("symbol") must match block specification name ("control_variable") on line line_number or be omitted.

003  END corresponding to block on line symbol must not have a symbol following it because there is no LABEL or control variable; found "line_number".

004  Symbol following END ("symbol") must match LABEL of SELECT specification ("control_variable") on line line_number or be omitted.

005  END must not immediately follow THEN.

006  END must not immediately follow ELSE.

007  END corresponding to SELECT on line symbol must not have a symbol following it because there is no LABEL; found "line_number".

C.1.9. Error 11 - Control stack full.

Explanation:

Your program exceeds the nesting level limit for control structures (for example, DO...END and IF...THEN...ELSE). This could be because of a looping INTERPRET instruction, such as:

```
line='INTERPRET line'
INTERPRET line
```

These lines loop until they exceed the nesting level limit and the language processor issues this message. Similarly, a recursive subroutine or internal function that does not end correctly can loop until it causes this message.

The associated subcodes are:

001  Insufficient control stack space; cannot continue execution.
C.1.10. Error 13 - Invalid character in program.

Explanation:

A character was found outside a literal (quoted) string that is not a whitespace character or one of the valid alphanumeric and special characters.

The associated subcodes are:

- **001**: Incorrect character in program “character” (‘hex_character’X).

900

message.

C.1.11. Error 14 - Incomplete DO/LOOP/SELECT/IF.

Explanation:

At the end of the program or the string for an INTERPRET instruction, a DO, LOOP, or SELECT instruction was found without a matching END or an IF clause that is not followed by a THEN clause. Putting the name of the control variable on each END closing a controlled loop can help locate this kind of error.

The associated subcodes are:

- **001**: DO instruction on line *line_number* requires matching END.
- **002**: SELECT instruction on line *line_number* requires matching END.
- **003**: THEN on line *line_number* must be followed by an instruction.
- **004**: ELSE on line *line_number* must be followed by an instruction.
- **005**: DO or LOOP instruction on line *line_number* requires matching END.
- **901**: OTHERWISE on line *line_number* requires matching END.

C.1.12. Error 15 - Invalid hexadecimal or binary string.

Explanation:

Hexadecimal strings must not have leading or trailing whitespace characters and whitespace can only be embedded at byte boundaries. Only the digits 0-9 and the letters a-f and A-F are allowed. The following are valid hexadecimal strings:

- '13'x
- 'A3C2 1c34'x
Binary strings can have whitespace only at the boundaries of groups of four binary digits. Only the digits 0 and 1 are allowed. These are valid binary strings:

```
'101'b
'11_1101'b
'101101_11010011'b
```

You might have mistyped one of the digits, for example, typing a letter O instead of the number 0. Or you might have used the one-character symbol X or B (the name of the variable X or B, respectively) after a literal string when the string is not intended as a hexadecimal or binary specification. In this case, use the explicit concatenation operator (||) to concatenate the string to the value of the symbol.

The associated subcodes are:

001 Incorrect location of whitespace character in position position in hexadecimal string.

002 Incorrect location of whitespace character in position position in binary string.

003 Only 0-9, a-f, A-F, and whitespace characters are valid in a hexadecimal string; found "character".

004 Only 0, 1, and whitespace characters are valid in a binary string; found "character".

005 Hexadecimal strings must be grouped in units that are multiples of two characters.

006 Binary strings must be grouped in units that are multiples of four characters.

---

**C.1.13. Error 16 - Label not found.**

**Explanation:**

A SIGNAL instruction has been executed or an event for which a trap was set with SIGNAL ON has occurred, and the language processor could not find the label specified. You might have mistyped the label or forgotten to include it.

The associated subcodes are:

001 Label "label_name" not found.

---

**C.1.14. Error 17 - Unexpected PROCEDURE.**

**Explanation:**

A PROCEDURE instruction was encountered at an incorrect position. This could occur because no internal routines are active or because the PROCEDURE instruction was not the first instruction processed after the CALL instruction or function call. One cause for this error is dropping through to an internal routine, rather than calling it with a CALL instruction or a function call.

The associated subcodes are:
C.1.15. Error 18 - THEN expected.

Explanation:
A THEN clause must follow each REXX IF or WHEN clause. The language processor found another clause before it found a THEN clause.

The associated subcodes are:

001
IF instruction on line line_number requires matching THEN clause.

002
WHEN instruction on line line_number requires matching THEN clause.

C.1.16. Error 19 - String or symbol expected.

Explanation:
A symbol or string was expected after the CALL or SIGNAL keywords but none was found. You might have omitted the string or symbol or inserted a special character (such as a parenthesis).

The associated subcodes are:

001
String or symbol expected after ADDRESS keyword.

002
String or symbol expected after CALL keyword.

003
String or symbol expected after NAME keyword.

004
String or symbol expected after SIGNAL keyword.

006
String or symbol expected after TRACE keyword.

007
String or symbol expected after PARSE keyword.

900
message.

901
String or symbol expected after ::CLASS keyword.

902
String or symbol expected after ::METHOD keyword.
Error 19 - String or symbol expected.

903
String or symbol expected after ::ROUTINE keyword.

904
String or symbol expected after ::REQUIRES keyword.

905
String expected after EXTERNAL keyword.

906
String or symbol expected after METACLASS keyword.

907
String or symbol expected after SUBCLASS keyword.

908
String or symbol expected after INHERIT keyword.

909
String or symbol expected after tilde (~).

911
String or symbol expected after superclass colon (:).

912
String or symbol expected after STREAM keyword.

913
String or symbol expected after MIXINCLASS keyword.

914
String or symbol expected as ::ATTRIBUTE directive name.

915
String or symbol expected as ::CONSTANT directive name.

916
String or symbol expected as ::CONSTANT value.

917
String or symbol expected as DIGITS value.

918
String or symbol expected as FUZZ value.

919
String or symbol expected as TRACE value.

920
String or symbol expected as ::RESOURCE directive name.

921
String or symbol expected after ::RESOURCE END keyword.

922
String or symbol expected after keyword keyword.
C.1.17. Error 20 - Symbol expected.

Explanation:

A symbol is expected after CALL ON, CALL OFF, END, ITERATE, LEAVE, NUMERIC, PARSE, SIGNAL ON, or SIGNAL OFF. Also, a list of symbols or variable references is expected after DROP, EXPOSE, and PROCEDURE EXPOSE. Either there was no symbol when one was required or the language processor found another token.

The associated subcodes are:

900  
message.

901  
Symbol expected after DROP keyword.

902  
Symbol expected after EXPOSE keyword.

903  
Symbol expected after PARSE keyword.

904  
Symbol expected after PARSE VAR.

905  
NUMERIC must be followed by one of the keywords DIGITS, FORM, or FUZZ; found "symbol".

906  
Symbol expected after ")" of a variable reference.

907  
Symbol expected after LEAVE keyword.

908  
Symbol expected after ITERATE keyword.

909  
Symbol expected after END keyword.

911  
Symbol expected after ON keyword.

912  
Symbol expected after OFF keyword.
Error 20 - Symbol expected.

913  Symbol expected after USE ARG.

914  Symbol expected after RAISE keyword.

915  Symbol expected after USER keyword.

916  Symbol expected after ::.

917  Symbol expected after superclass colon (:).

918  Symbol expected after LABEL keyword.

919  Symbol expected as ::ANNOTATE attribute name; found “name”.

920  Symbol expected after NAMESPACE keyword.

921  Symbol expected as a class name of qualified class name.

922  Symbol expected as a routine name of qualified CALL instruction.

923  Symbol expected as a name of namespace-qualified symbol.

924  Symbol expected for the ::ANNOTATE type.

925  Symbol expected after the FORM keyword.

926  Symbol expected after the DELEGATE keyword.

927  Symbol expected after USE LOCAL.

928  Symbol expected as an indirect variable name; found “token”.

929  Symbol expected after keyword keyword.

930  Simple variable or stem symbol expected after > or < prefix operator; found token.

931  Simple variable or stem symbol expected after > or < USE ARG reference operator; found token.
932
  Stem symbol expected after STEM keyword.

933
  Symbol expected after WITH keyword.

934
  Symbol expected after COUNTER keyword.

C.1.18. Error 21 - Invalid data on end of clause.

Explanation:

A clause such as SELECT or NOP is followed by a token other than a comment.

The associated subcodes are:

900  

message.

901
  Data must not follow the NOP keyword; found "data".

902
  Data must not follow the SELECT keyword; found "data".

903
  Data must not follow the NAME keyword; found "data".

904
  Data must not follow the condition name; found "data".

905
  Data must not follow the SIGNAL label name; found "data".

906
  Data must not follow the TRACE setting; found "data".

907
  Data must not follow the LEAVE control variable name; found "data".

908
  Data must not follow the ITERATE control variable name; found "data".

909
  Data must not follow the END control variable name; found "data".

911
  Data must not follow the NUMERIC FORM specification; found "data".

912
  Data must not follow the GUARD OFF specification; found "data".

913
  Data must not follow the ::CONSTANT value; found "data".
914
  Data must not follow the ::RESOURCE directive; found "data".


Explanation:

A literal string contains character codes that are not valid. This might be because some characters are not possible, or because the character set is extended and certain character combinations are not allowed.

The associated subcodes are:

  001
    Incorrect character string "character_string" ("hex_string"X).

  900
    message.

  901
    Incorrect double-byte character.

C.1.20. Error 23 - Invalid data string.

Explanation:

A data string (that is, the result of an expression) contains character codes that are not valid. This might be because some characters are not possible, or because the character set is extended and certain character combinations are not allowed.

The associated subcodes are:

  001
    Incorrect data string "string" ("hex_string"X).

  900
    message.

C.1.21. Error 24 - Invalid TRACE request.

Explanation:

This message is issued when:

• The option on a TRACE instruction or the argument to the built-in function does not start with A, C, E, F, I, L, N, O, or R.

• In interactive debugging, you entered a number that is not a whole number.

The associated subcodes are:

  001
    TRACE request letter must be one of "ACEFILNOR"; found "value".

  901
    Numeric TRACE requests are valid only from interactive debugging.
C.1.22. Error 25 - Invalid subkeyword found.

Explanation:

An unexpected token was found at his position of an instruction where a particular subkeyword was expected. For example, in a NUMERIC instruction, the second token must be DIGITS, FUZZ, or FORM.

The associated subcodes are:

001  CALL ON must be followed by one of the keywords ERROR, FAILURE, HALT, NOTREADY, USER, or ANY; found "word".

002  CALL OFF must be followed by one of the keywords ERROR, FAILURE, HALT, NOTREADY, USER, or ANY; found "word".

003  SIGNAL ON must be followed by one of the keywords ERROR, FAILURE, HALT, LOSTDIGITS, NOTREADY, NOMETHOD, NOSTRING, NOVALUE, SYNTAX, USER, or ANY; found "word".

004  SIGNAL OFF must be followed by one of the keywords ERROR, FAILURE, HALT, LOSTDIGITS, NOTREADY, NOMETHOD, NOSTRING, NOVALUE, SYNTAX, USER, or ANY; found "word".

011  NUMERIC FORM must be followed by one of the keywords SCIENTIFIC or ENGINEERING; found "word".

012  PARSE must be followed by optional keywords LOWER or UPPER, by an optional keyword CASELESS, and by one of the keywords ARG, LINEIN, PULL, SOURCE, VALUE, VAR, or VERSION; found "word".

015  NUMERIC must be followed by one of the keywords DIGITS, FORM, or FUZZ; found "word".

017  PROCEDURE must be followed by the keyword EXPOSE or nothing; found "word".

900  message.

901  Unknown keyword on ::CLASS directive; found "word".

902  Unknown keyword on ::METHOD directive; found "word".

903  Unknown keyword on ::ROUTINE directive; found "word".

904  Unknown keyword on ::REQUIRES directive; found "word".

message.
Error 25 - Invalid subkeyword found.

905
USE must be followed by the keyword ARG, LOCAL or STRICT; found "word".

906
RAISE must be followed by one of the keywords ERROR, FAILURE, HALT, LOSTDIGITS, NOMETHOD, NOSTRING, NOTREADY, NOVALUE, PROPAGATE, SYNTAX, or USER; found "word".

907
Unknown keyword on RAISE instruction; found "word".

908
Duplicate DESCRIPTION keyword found.

909
Duplicate ADDITIONAL or ARRAY keyword found.

911
Duplicate RETURN or EXIT keyword found.

912
GUARD ON or GUARD OFF must be followed by the keyword WHEN; found "word".

913
GUARD must be followed by the keyword ON or OFF; found "word".

914
CALL ON condition must be followed by the keyword NAME; found "word".

915
SIGNAL ON condition must be followed by the keyword NAME; found "word".

916
Unknown keyword on FORWARD instruction; found "keyword".

917
Duplicate TO keyword found.

918
Duplicate ARGUMENTS or ARRAY keyword found.

919
Duplicate RETURN or CONTINUE keyword found.

921
Duplicate CLASS keyword found.

922
Duplicate MESSAGE keyword found.

923
SELECT must be followed by the keyword LABEL or CASE; found "word".

924
Unknown keyword on ::OPTIONS directive; found "word".
C.1.23. Error 26 - Invalid whole number.

Explanation:

An expression was found that did not evaluate to a whole number or is greater than the limit (the default is 999,999,999 for 32-bit system and 999,999,999,999,999,999 for 64-bit systems):

- The positional patterns in parsing templates (including variable positional patterns)
- The operand to the right of the power operator
- The values of expr and exprf in the DO and LOOP instructions
- The values given for DIGITS or FUZZ in the NUMERIC instruction
- The number used in the option of the TRACE setting
- This error is also raised if the value is not permitted (for example, a negative repetition count in a DO instruction), or the division performed during an integer divide or remainder operation does not result in a whole number.

The associated subcodes are:

002

Value of repetition count expression in DO or LOOP instruction must be zero or a positive whole number; found "value".
C.1.24. Error 27 - Invalid DO or LOOP syntax.

Explanation:

A syntax error was found in a DO or LOOP instruction. You probably used BY, TO, FOR, WHILE, or UNTIL twice, used a WHILE and an UNTIL, or used BY, TO, or FOR when there is no control variable specified.

The associated subcodes are:

001 Only one WHILE or UNTIL condition can be used on the same loop.

901 Incorrect data following FOREVER keyword on the loop; found "data".
Error 28 - Invalid LEAVE or ITERATE.

DO or LOOP keyword keyword can be specified only once.

No INDEX or ITEM control variable specified on a WITH loop.

OVER keyword expected for a WITH loop.

COUNTER keyword not allowed on a simple DO instruction.

C.1.25. Error 28 - Invalid LEAVE or ITERATE.

Explanation:

A LEAVE or ITERATE instruction was found at an incorrect position. Either no loop was active, or the name specified on the instruction did not match the control variable of any active loop. Note that internal routine calls and the INTERPRET instruction protect loops by making them inactive. Therefore, for example, a LEAVE instruction in a subroutine cannot affect a DO loop in the calling routine. You probably tried to use the SIGNAL instruction to transfer control within or into a loop. Because a SIGNAL instruction ends all active loops, any ITERATE or LEAVE instruction causes this message.

The associated subcodes are:

001

LEAVE is valid only within a repetitive loop or labeled block instruction.

002

ITERATE is valid only within a repetitive loop.

003

Symbol following LEAVE ("symbol") must either match the label of a current loop or block instruction.

004

Symbol following ITERATE ("symbol") must either match the label of a current loop or be omitted.

005

Symbol following ITERATE ("symbol") does not match a repetitive block instruction.


Explanation:

The environment name specified on the ADDRESS instruction is longer than permitted for the system under which the interpreter is running.

The associated subcodes are:

001

Environment name exceeds limit characters; found "environment_name".

C.1.27. Error 30 - Name or symbol too long.

Explanation:
A variable name, label name, or symbol has exceeded the allowed limit of 250 characters.

The associated subcodes are:

001
   Name or symbol exceeds 250 characters: "name".

900
   message.

C.1.28. Error 31 - Name starts with number or ".".

Explanation:

A variable was found whose name begins with a numeric digit or a period. You cannot assign a value to such a variable because you could then redefine numeric constants.

The associated subcodes are:

001
   A value cannot be assigned to a number; found "number".

002
   Variable symbol must not start with a number; found "symbol".

003
   Variable symbol must not start with a "."; found "symbol".

900
   message.

C.1.29. Error 33 - Invalid expression result.

Explanation:

The result of an expression was found not to be valid in the context in which it was used.

The associated subcodes are:

001
   Value of NUMERIC DIGITS ("value") must exceed value of NUMERIC FUZZ ("value").

002
   Value of NUMERIC DIGITS ("value") must not exceed value.

900
   message.

901
   Incorrect expression result following VALUE keyword of ADDRESS instruction.

902
   Incorrect expression result following VALUE keyword of SIGNAL instruction.

903
   Incorrect expression result following VALUE keyword of TRACE instruction.
C.1.30. Error 34 - Logical value not 0 or 1.

Explanation:

An expression was found in an IF, WHEN, WHILE, or UNTIL phrase that did not result in a 0 or 1. Any value operated on by a logical operator must result in a 0 or 1. For example, the phrase "If result then exit rc" fails if result has a value other than 0 or 1.

The associated subcodes are:

001 Value of expression following IF keyword must be exactly "0" or "1"; found "value".

002 Value of expression following WHEN keyword must be exactly "0" or "1"; found "value".

003 Value of expression following WHILE keyword must be exactly "0" or "1"; found "value".

004 Value of expression following UNTIL keyword must be exactly "0" or "1"; found "value".

005 Value of expression to the left of the logical operator "operator" must be exactly "0" or "1"; found "value".

006 Value of logical list expression element must be exactly "0" or "1"; found "value".

message.

Logical value must be exactly "0" or "1"; found "value".

Value of expression following GUARD keyword must be exactly "0" or "1"; found "value".

Authorization return value must be exactly "0" or "1"; found "value".

Property logical value must be exactly "0", "1", "true", or "false"; found "value".

SELECT CASE comparison result must be exactly "0" or "1"; found "value".

Supplier AVAILABLE method did not return exactly "0" or "1"; found "value".

C.1.31. Error 35 - Invalid expression.

Explanation:
Error 35 - Invalid expression.

An expression contains a grammatical error. Possible causes:
• An expression is missing when one is required
• You ended an expression with an operator
• You specified, in an expression, two operators next to one another with nothing in between them
• You did not specify a right parenthesis when one was required
• You used special characters (such as operators) in an intended character expression without enclosing them in quotation marks

The associated subcodes are:

001
Incorrect expression detected at "token".

900
message.

901
Prefix operator "operator" is not followed by an expression term.

902
Missing conditional expression following IF keyword.

903
Missing conditional expression following WHEN keyword.

904
Missing initial expression for DO or LOOP control variable.

905
Missing expression following BY keyword.

906
Missing expression following TO keyword.

907
Missing expression following FOR keyword.

908
Missing expression following WHILE keyword.

909
Missing expression following UNTIL keyword.

911
Missing expression following OVER keyword.

912
Missing expression following INTERPRET keyword.

913
Missing expression following OPTIONS keyword.
Error 35 - Invalid expression.

914  Missing expression following VALUE keyword of an ADDRESS instruction.

915  Missing expression following VALUE keyword of a SIGNAL instruction.

916  Missing expression following VALUE keyword of a TRACE instruction.

917  Missing expression following VALUE keyword of a NUMERIC FORM instruction.

918  Missing expression following assignment instruction.

919  Operator "operator" is not followed by an expression term.

921  Missing expression following GUARD keyword.

922  Missing expression following DESCRIPTION keyword of a RAISE instruction.

923  Missing expression following ADDITIONAL keyword of a RAISE instruction.

924  Missing "(" on expression list of the ARRAY keyword.

925  Missing expression following TO keyword of a FORWARD instruction.

926  Missing expression following ARGUMENTS keyword of a FORWARD instruction.

927  Missing expression following MESSAGE keyword of a FORWARD instruction.

928  Missing expression following CLASS keyword of a FORWARD instruction.

929  Missing expression in logical expression list.

930  Invalid or missing expression following "=" token of a USE ARG instruction.

931  Missing expression following "(" of parse template.

932  Missing expression for calculated CALL name.

933  Missing expression following CASE keyword of a SELECT instruction.
934
  Missing expression in WHEN case expression list.

935
  Missing expression following keyword keyword of a instruction instruction.

936
  Missing expression on ::CONSTANT directive.

C.1.32. Error 36 - Unmatched "(" or "[" in expression.

Explanation:
A matched parenthesis or bracket was found within an expression. There are more left parentheses than right parentheses or more left brackets than right brackets. To include a single parenthesis in a command, enclose it in quotation marks.

The associated subcodes are:

900
  message.

901
  Left parenthesis "(" in position position on line line_number requires a corresponding right parenthesis ")".

902
  Square bracket "[" in position position on line line_number requires a corresponding right square bracket "]".

C.1.33. Error 37 - Unexpected ",", ")", or "]".

Explanation:
Either a comma was found outside a function invocation, or there are too many right parentheses or right square brackets in an expression. To include a comma in a character expression, enclose it in quotation marks. For example, write the instruction:

```plaintext
Say Enter A, B, or C
```

as follows:

```plaintext
Say 'Enter A, B, or C'
```

The associated subcodes are:

001
  Unexpected ",".

002
  Unmatched ")" in expression.

900
  message.

901
  Unexpected "]".
C.1.34. Error 38 - Invalid template or pattern.

Explanation:
A special character that is not allowed within a parsing template (for example, "%") has been found, or
the syntax of a variable pattern is incorrect (that is, no symbol was found after a left parenthesis). This
message is also issued if you omit the WITH subkeyword in a PARSE VALUE instruction.

The associated subcodes are:

001
   Incorrect PARSE template detected at "column_position".

002
   Incorrect PARSE position detected at "column_position".

003
   PARSE VALUE instruction requires WITH keyword.

900
   message.

901
   Missing PARSE relative position.

C.1.35. Error 39 - Evaluation stack overflow.

Explanation:
The expression is too complex to be evaluated by the language processor.

C.1.36. Error 40 - Incorrect call to routine.

Explanation:
An incorrect call to a routine was found. Possible causes:
• You passed incorrect data (arguments) to the built-in or external routine.
• You passed too many arguments to the built-in, external, or internal routine.
• The external routine called was not compatible with the language processor.

If you did not try to call a routine, you might have a symbol or a string adjacent to a "(" when you
meant it to be separated by a blank or other operator. The language processor would treat this as a
function call. For example, write TIME(4+5) as follows: TIME*(4+5)

The associated subcodes are:

001
   External routine "routine" failed.

003
   Not enough arguments in invocation of routine; minimum expected is number.

004
   Too many arguments in invocation of routine; maximum expected is number.

005
   Missing argument in invocation of routine; argument argument_number is required.
Error 40 - Incorrect call to routine.

011  
    function_name argument argument_number must be a number; found "value".

012  
    function_name argument argument_number must be a whole number; found "value".

013  
    function_name argument argument_number must be zero or positive; found "value".

014  
    function_name argument argument_number must be positive; found "value".

019  
    function_name argument 2, "value", is not in the format described by argument 3, "value".

021  
    function_name argument argument_number must not be a null string.

022  
    function_name argument argument_number must be a single character or null; found "value".

023  
    function_name argument argument_number must be a single character; found "value".

024  
    function_name argument argument_number must be a binary string; found "value".

025  
    function_name argument argument_number must be a hexadecimal string; found "value".

026  
    function_name argument argument_number must be a valid symbol; found "value".

027  
    function_name argument 1 must be a valid stream name; found "value".

028  
    function_name argument argument_number must be a character class name or a single character; found "value".

029  
    function_name conversion to format "value" is not allowed.

032  
    RANDOM difference between argument 1 ("value") and argument 2 ("value") must not exceed 999,999,999.

033  
    RANDOM argument 1 ("argument") must be less than or equal to argument 2 ("argument").

034  
    SOURCELINE argument 1 ("argument") must be less than or equal to the number of lines in the program (argument).

035  
    X2D argument 1 cannot be expressed as a whole number; found "value".

674
043  
function_name argument number must be a single non-alphanumeric character or the null string; found "value".

044  
function_name argument number, "value", is a format incompatible with the separator specified in argument number.

900  
message.

901  
Result returned by routine is longer than length: "value".

902  
function_name argument argument_number must not exceed the whole number limit.

903  
function_name argument argument_number must be in the range 0-99; found "value".

904  
function_name argument argument_number must be one of values; found "value".

905  
TRACE setting letter must be one of "ACEFILNOR"; found "value".

912  
function_name argument argument_number must be a single-dimensional array; found "value".

913  
function_name argument argument_number must have a string value; found "value".

914  
Unknown VALUE function variable environment selector; found "value".

915  
function_name cannot be used with QUEUE:.

916  
Cannot read from a write-only property.

917  
Cannot write to a read-only property or typelib element.

918  
Invalid native function signature specification.

919  
Argument argument must have a stem object or stem name value; found "value".

920  
function_name argument name must be one of "option"; found "value".

921  
Argument first option ("first value") must be greater than argument second option ("second value").
922
Stem argument does not have a valid count for element 0.

923
The operation is outside the size of the stem array ("value").

924
Stem element at position "position" is not set.

925
Argument argument must be an array object, a stem object, or a stem name value; found "value".

C.1.37. Error 41 - Bad arithmetic conversion.

Explanation:
A term in an arithmetic expression is not a valid number or has an exponent outside the allowed range of whole number range.
You might have mistyped a variable name, or included an arithmetic operator in a character expression without putting it in quotation marks.
The associated subcodes are:

001
Nonnumeric value ("value") used in arithmetic operation.

003
Nonnumeric value ("value") used with prefix operator.

004
Value of TO expression of DO or LOOP instruction must be numeric; found "value".

005
Value of BY expression of DO or LOOP instruction must be numeric; found "value".

006
Value of control variable expression of DO or LOOP instruction must be numeric; found "value".

007
Exponent exceeds number digits; found "value".

900
message.

901
Value of RAISE instruction SYNTAX expression must be numeric; found "value".

C.1.38. Error 42 - Arithmetic overflow/underflow.

Explanation:
The result of an arithmetic operation requires an exponent that is greater than the platform limit of nine digits.
This error can occur during the evaluation of an expression (often as a result of trying to divide a number by 0) or while stepping a loop control variable.
Error 43 - Routine not found.

The associated subcodes are:

001
   Arithmetic overflow detected at: "value operator value".

002
   Arithmetic underflow detected at: "value operator value".

003
   Arithmetic overflow; divisor must not be zero.

900
   message.

901
   Arithmetic overflow; exponent ("exponent") exceeds number digits.

902
   Arithmetic underflow; exponent ("exponent") exceeds number digits.

903
   Arithmetic underflow; zero raised to a negative power.

C.1.39. Error 43 - Routine not found.

Explanation:

A function has been invoked within an expression or a subroutine has been invoked by a CALL, but it cannot be found. Possible reasons:

• The specified label is not in the program

• It is not the name of a built-in function

• The language processor could not locate it externally

Check if you mistyped the name.

If you did not try to call a routine, you might have put a symbol or string adjacent to a "(" when you meant it to be separated by a blank or another operator. The language processor then treats it as a function call. For example, write the string 3(4+5) as 3*(4+5).

The associated subcodes are:

001
   Could not find routine "routine".

900
   message.

901
   Could not find file "file" for ::REQUIRES.

902
   Routine "routineName" not found in namespace "namespace".

C.1.40. Error 44 - Function or message did not return data.

Explanation:
The language processor called an external routine within an expression. The routine seemed to end without error, but it did not return data for use in the expression.

You might have specified the name of a program that is not intended for use as a REXX function. Call it as a command or subroutine instead.

The associated subcodes are:

001
   No data returned from function “function”.

900
   message.

**C.1.41. Error 45 - No data specified on function RETURN.**

**Explanation:**

A REXX program has been called as a function, but returned without passing back any data.

The associated subcodes are:

001
   Data expected on RETURN instruction because routine “routine” was called as a function.

**C.1.42. Error 46 - Invalid variable reference.**

**Explanation:**

Within an ARG, DROP, EXPOSE, PARSE, PULL, or PROCEDURE instruction, the syntax of a variable reference (a variable whose value is to be used, indicated by its name being enclosed in parentheses) is incorrect. The right parenthesis that must immediately follow the variable name might be missing or the variable name might be misspelled.

The associated subcodes are:

001
   Extra token ("token") found in variable reference list; "") expected.

900
   message.

901
   Missing "") in variable reference.

902
   Extra token ("token") found in USE ARG variable reference; ",," or end of instruction expected.

**C.1.43. Error 47 - Unexpected label.**

**Explanation:**

A label was used in the expression being evaluated for an INTERPRET instruction or in an expression entered during interactive debugging.

The associated subcodes are:

001
   INTERPRET data must not contain labels; found “label.”
C.1.44. Error 48 - Failure in system service.

Explanation:

The language processor stopped processing the program because a system service, such as stream input or output or the manipulation of the external data queue, has failed to work correctly.

The associated subcodes are:

001
  Failure in system service: service.

900
  message.

C.1.45. Error 49 - Interpretation error.

Explanation:

A severe error was detected in the language processor or execution process during internal self-consistency checks.

The associated subcodes are:

001
  Interpretation error: unexpected failure initializing the interpreter.

002
  Interpretation error: unmatched type number.

900
  message.

C.1.46. Error 88 - Invalid argument.

Explanation:

An argument passed to a method, function, or routine was not valid.

The associated subcodes are:

900
  message.

901
  Missing argument; argument argument is required.

902
  The argument argument must be a number; found "value".

903
  The argument argument must be a whole number; found "value".

904
  The argument argument must be zero or a positive whole number; found "value".

905
  Argument argument must be a positive whole number; found "value".
Error 88 - Invalid argument.

906  
Argument argument must not exceed limit; found "value".

907  
Argument argument must be in the range min to max; found "value".

908  
Argument argument must not be a null string.

909  
Argument argument must have a string value.

910  
Argument argument is an invalid pad or character argument; found "value".

911  
Argument argument is an invalid length value; found "value".

912  
Argument argument is an invalid position value; found "value".

913  
Argument argument must be a single-dimensional array.

914  
Argument argument must be an instance of the class class.

915  
Argument argument could not be converted to a type type.

916  
Argument argument must be one of values; found "value".

917  
Argument argument reason.

918  
Argument argument is not in a valid format; found "value".

919  
Argument argument is not in valid pointer format; found "value".

920  
Argument argument must be a stem object or stem name; found "value".

921  
Argument argument must be a valid double value; found "value".

922  
Too many arguments in invocation; number expected.

923  
name argument argument_number must be a single-dimensional array; found "value".

924  
name argument argument_number must be an array with exactly count items.
Error 89 - Variable or message term expected.

The argument argument must be zero or a positive number; found "value".

The argument argument must be a positive number; found "value".

The argument argument must be exactly "0" or "1"; found "value".

The argument argument must be a VariableReference instance; found "value".

The argument argument must be a VariableReference for a Stem variable; found "value".

The argument argument must be a VariableReference for a simple variable; found "value".

Argument argument was omitted. A VariableReference argument is required.

Date template template contains an invalid pattern; found "value".

Unable to parse date "date" with template "template".

C.1.47. Error 89 - Variable or message term expected.

Explanation:

An instruction was expecting either a single Rexx variable symbol or a message term to be used for an assignment.

The associated subcodes are:

001

The USE instruction requires a comma-separated list of variables or assignment message terms.

002

The PARSE instruction was expecting a variable or a message term.

C.1.48. Error 90 - External name not found.

Explanation:

An external method or routine (specified with the EXTERNAL option on an ::ATTRIBUTE, a ::METHOD, or a ::ROUTINE directive) cannot be found.

The associated subcodes are:

900

message.

997

Unable to find external class "class".
C.1.49. Error 91 - No result object.

Explanation:
A message term requires a result object, but the method did not return one.

The associated subcodes are:

- 900 message
- 999 Message "message" did not return a result.

C.1.50. Error 92 - OLE error.

The associated subcodes are:

- 900 message
- 901 An unknown OLE error occurred (HRESULT=hresult).
- 902 Cannot convert OLE VARIANT to REXX object: The conversion of the VARIANT type varianttype into a REXX object failed.
- 903 Cannot convert REXX object to OLE VARIANT: The conversion of rexx_object into a VARIANT failed.
- 904 The number of elements provided to the method or property is different from the number of parameters accepted by it.
- 905 One of the parameters is not a valid VARIANT type.
- 906 OLE exception: exc_name.
- 907 The requested method does not exist, or you tried to set the value of a read-only property.
- 908 One of the parameters could not be coerced to the desired type.
- 909 One or more of the parameters could not be coerced to the desired type. The first parameter with incorrect type is argument index.
910  
A required parameter was omitted.

911  
Could not create OLE instance.

912  
The object invoked has disconnected from its clients.

C.1.51. Error 93 - Incorrect call to method.

Explanation:
The specified method, built-in function, or external routine exists, but you used it incorrectly.

The associated subcodes are:

900  
message.

901  
Not enough arguments for method; number expected.

902  
Too many arguments in invocation of method; number expected.

903  
Missing argument in method; argument argument is required.

904  
Method argument argument must be a number; found "value".

905  
Method argument argument must be a whole number; found "value".

906  
Method argument argument must be zero or a positive whole number; found "value".

907  
Method argument argument must be a positive whole number; found "value".

908  
Method argument argument must not exceed limit; found "value".

909  
Method argument argument must be in the range 0-99; found "value".

911  
Method argument argument must not be null.

912  
Method argument argument must be a hexadecimal string; found "value".

913  
Method argument argument must be a valid symbol; found "value".
914
Method argument argument must be one of arguments; found "value".

915
Method option must be one of "arguments"; found "value".

917
Method method does not exist.

918
Incorrect list index "index".

919
Incorrect array position "position".

921
Argument missing on binary operator.

922
Incorrect pad or character argument specified; found "value".

923
Invalid length argument specified; found "value".

924
Invalid position argument specified; found "value".

925
Not enough subscripts for array; number expected.

926
Too many subscripts for array; number expected.

927
Length must be specified to convert a negative value.

928
D2X value must be a valid whole number; found "value".

929
D2C value must be a valid whole number; found "value".

931
Incorrect location of whitespace character in position position in hexadecimal string.

932
Incorrect location of whitespace character in position position in binary string.

933
Only 0-9, a-f, A-F, and whitespace characters are valid in a hexadecimal string; character found "character".

934
Only 0, 1, and whitespace characters are valid in a binary string; character found "character".
Error 93 - Incorrect call to method.

935
X2D result is not a valid whole number with NUMERIC DIGITS digits.

936
C2D result is not a valid whole number with NUMERIC DIGITS digits.

937
No more supplier items available.

938
Method argument argument must have a string value.

939
Method argument argument must have a single-dimensional array value.

940
method method target must be a whole number; found "value".

941
Exponent of "exponent" is too large for number spaces.

942
Integer part of "integer" is too large for number spaces.

943
method method target must be a number; found "value".

944
Method argument argument must be a message object.

945
Missing argument in message array; argument argument is required.

946
A message array must be a single-dimensional array with 2 elements.

947
Method SECTION can be used only on single-dimensional arrays.

948
Method argument argument must be of the class class.

949
The value and index objects must be the same for PUT to an index-only collection.

951
Incorrect alarm time; found “time”.

952
Method argument argument is an array and does not contain all string values.

953
Method argument argument could not be converted to type type.

954
Method "method" can be used only on a single-dimensional array.
Element *element* of the array must be a string.

Target object "*object*" is not a subclass of the message override scope (*scope*).

Positioning of transient streams is not valid.

An array cannot contain more than *size* elements.

The *argument* argument must be a string or array object.

Invalid Base64 encoded string.

Call to unsupported or unimplemented method.

Application error: *message*.

Method *name* is ABSTRACT and cannot be directly invoked.

Incorrect queue index "*index*".

NEW method is not supported for the *name* class.

Invalid native method signature specification.

Method argument *argument* must have a stem object value; found "*value*".

COPY method is not supported for object *object*.

Method argument *argument* cannot have more than a single dimension.

A message name argument must be a string or an array with 2 elements; found "*value*".

ORDERABLE comparison method "*method*" argument cannot be .nil.

The *argument* argument must be a string, array, or method object.

Invalid *option* value; found "*value*".
C.1.52. Error 97 - Object method not found.

Explanation:
The object does not have a method with the given name. A frequent cause of this error is an uninitialized variable.

The associated subcodes are:

001  Object "object" does not understand message "message".

002  Object "object" cannot accept private message "message" from this context.

003  Object "object" cannot accept package scope message "message" from a different package caller.

004  Constant "name" of object "object" has not been initialized.

005  An unhandled NOMETHOD condition has been raised.

C.1.53. Error 98 - Execution error.

Explanation:
The language processor detected a specific error during execution. The associated error gives the reason for the error.

The associated subcodes are:

900  message.

901  Object "object" was not correctly initialized.

902  Unable to convert object "object" to a double-float value.

903  Unable to load library "name".
Abnormal termination occurred.

Deadlock detected on a guarded method.

Incorrect object reference detected.

Object of type "type" was required.

Metaclass "metaclass" not found.

Class "class" not found.

Cyclic inheritance in program "program".

Unable to convert object "object" to a single-dimensional array value.

Unable to convert object "object" to a string value.

A message object that was sent a START message cannot be sent another START, SEND, or REPLY message.

Message object "object" received an error from message "message".

Incorrect condition object received for RAISE OBJECT; found "value".

No active condition available for PROPAGATE.

Unable to convert object "object" to a method.

Unable to open file "file" for writing; open result was "error".

Address environment "address" does not support input/output redirection.

REPLACE or APPEND cannot be specified with a RexxQueue object USING target.

Address command redirection failed (oserror).
Error 98 - Execution error.

Object "object" is not a valid ADDRESS WITH INPUT source.

REPLY can be issued only once per method invocation.

RETURN cannot return a value after a REPLY.

EXIT cannot return a value after a REPLY.

Message search overrides can be used only from methods of the target object.

Additional information for SYNTAX errors must be a single-dimensional array of values.

Unknown error number specified on RAISE SYNTAX; found "number".

Class "class" must be a MIXINCLASS for INHERIT.

Class "class" is not a subclass of "class" base class "class".

Class "class" cannot inherit from itself, a superclass, or a subclass ("class").

Class "class" has not inherited class "class".

FORWARD arguments must be a single-dimensional array of values.

FORWARD can only be issued in an object method invocation.

Authorization failure: value.

Concurrency not supported.

Circular ::REQUIRES references detected with name.

External command "command" ended with return code rc.

External command "command" failed with return code rc.

Number number has more digits than the current precision.
Error 98 - Execution error.

973  Object "object" does not have a string representation.
974  Stream "object" is not ready.
975  Missing array element at position position.
976  Stem object default value cannot be another stem object.
978  Unable to load method "name" from library "library".
979  Unable to load routine "name" from library "library".
980  Unable to load native routine "name".
981  Target RexxContext is no longer active.
982  Library "name" is not compatible with current interpreter version.
983  Execution thread does not match API thread context.
984  User additions are not allowed to the REXX package.
985  User additions are not allowed to the REXX language classes.
986  Reference to unassigned variable "name".
987  Namespace "namespace" not found in package "package".
988  Class "className" not found in namespace "namespace".
989  Class name is ABSTRACT and cannot be directly created.
990  Class name is a metaclass and cannot be made ABSTRACT.
991  Method name may only be invoked from a method of the same object or one of its classes.
992  The EXPOSE instruction may only be used from method invocations.
993  
The USE LOCAL instruction may only be used from method invocations.

994  
Unable to convert object "object" to a supplier object.

995  
Unable to reference variable "object"; it must be an uninitialized local variable.

996  
Object "object" is not a valid ADDRESS WITH OUTPUT or ERROR target.

997  
REPLACE or APPEND cannot be specified with a Stream object USING target.

998  
Stem "object" does not contain a size count in element 0.

999  
Unable to open file "file" for reading; open result was "error".

**C.1.54. Error 99 - Translation error.**

**Explanation:**

An error was detected in the language syntax. The associated error subcode identifies the syntax error.

The associated subcodes are:

900  
*message.*

901  
Duplicate ::CLASS directive instruction.

902  
Duplicate ::METHOD directive instruction.

903  
Duplicate ::ROUTINE directive instruction.

904  
Duplicate ::REQUIRES directive instruction.

905  
CLASS keyword on ::METHOD directive requires a matching ::CLASS directive.

906  
A ::CONSTANT directive with an expression requires a matching ::CLASS directive.

907  
EXPOSE must be the first instruction executed after a method invocation.

908  
INTERPRET data must not contain EXPOSE.
GUARD must be the first instruction executed after EXPOSE or USE.

USE LOCAL must be the first instruction executed after a method invocation.

GUARD can only be issued in an object method invocation.

INTERPRET data must not contain GUARD.

GUARD instruction did not include references to exposed variables.

INTERPRET data must not contain directive instructions.

INTERPRET data must not contain USE LOCAL.

Unrecognized directive instruction.

Incorrect external name specification "name".

USE ARG requires a "," between variable names; found "token".

REPLY can only be issued in an object method invocation.

Incorrect program line in method source array.

::REQUIRES directives must appear before other directive instructions.

INTERPRET data must not contain FORWARD.

INTERPRET data must not contain REPLY.

An ATTRIBUTE method name must be a valid variable name; found "name".

Incorrect class external; too many parameters.

"classname" is not a valid metaclass.

Incorrect class external; class name missing or invalid.
Incorrect class external; invalid class server "servername".

The "..." argument marker can only appear at the end of the argument list.

Duplicate ::ATTRIBUTE directive instruction.

Duplicate ::CONSTANT directive instruction.

Abstract methods cannot have a method body.

Attribute methods cannot have a method body.

External attributes cannot have a method body.

External methods cannot have a method body.

Attribute methods without a SET or GET designation cannot have a method body.

Constant methods cannot have a method body.

External routines cannot have a code body.

Abstract attributes cannot have a method body.

Method or routine code may not contain directive instructions.

Duplicate ::RESOURCE directive instruction.

Missing ::RESOURCE end marker "endmarker" for resource "resourcename".

The REXX name is reserved for the language-provided namespace.

::ANNOTATE target type "name" not found.

Delegate methods cannot have a method body.

Delegate attributes cannot have a method body.
USE LOCAL cannot process compound variables; found "variable".

"classname" is not a valid class.

A USE ARG default value is not allowed for variable references.

C.2. RXSUBCOM Utility Program
RXSUBCOM issues the following errors:

C.2.1. Error 116 - The RXSUBCOM REGISTER parameters are incorrect.
Explanation:
RXSUBCOM REGISTER requires the following parameters:

RXSUBCOM REGISTER envname dllname procname
  envname is the name of the subcommand handler.
  dllname is the name of the file containing the subcommand handler routine.
  procname is the name of the procedure that REXX calls as a subcommand handler.

C.2.2. Error 117 - The RXSUBCOM DROP parameters are incorrect.
Explanation:
RXSUBCOM DROP requires the environment name be specified.

RXSUBCOM DROP envname [dllname]
  envname is the name of the subcommand handler.
  dllname is the name of the file containing the subcommand handler routine (optional).

C.2.3. Error 118 - The RXSUBCOM LOAD parameters are incorrect.
Explanation:
RXSUBCOM LOAD requires the environment name be specified.

RXSUBCOM LOAD envname [dllname]
  envname is the name of the subcommand handler.
  dllname is the name of the file containing the subcommand handler routine (optional).
C.2.4. Error 125 - The RXSUBCOM QUERY parameters are incorrect.

Explanation:
RXSUBCOM QUERY requires the environment name be specified.

RXSUBCOM QUERY envname [dllname]

envname
  is the name of the subcommand handler.

dllname
  is the name of the file containing the subcommand handler routine (optional).

C.3. RXQUEUE Utility Program
RXQUEUE issues the following errors:

C.3.1. Error 119 - The REXX rxapi queuing system is not available.

C.3.2. Error 120 - The size of the data is incorrect.

Explanation:
The data supplied to the RXQUEUE command is too long. The RXQUEUE program accepts data records containing 0 - 65472 bytes. A record exceeded the allowable limits.

C.3.3. Error 121 - Storage for data queues is exhausted.

Explanation:
The queuing system is out of memory. No more storage is available to store queued data.

C.3.4. Error 122 - The name %1 is not a valid queue name.

Explanation:
The queue name contains an invalid character. Only the following characters can appear in queue names:

  'A' .. 'Z', '0' .. '9', '.', '!', '?', '_'

C.3.5. Error 123 - The queue access mode is not correct.

Explanation:
An internal error occurred in RXQUEUE. The RXQUEUE program tried to access a queue with an incorrect access mode. Correct access modes are LIFO and FIFO.

C.3.6. Error 124 - The queue %1 does not exist.

Explanation:
The command attempted to access a nonexistent queue.
C.3.7. Error 131 - The syntax of the command is incorrect.

C.3.8. Error 132 - System error occurred while processing the command.

C.4. rexxc Utility Program

When rexxc encounters a syntax error in a Rexx program while translating or syntax checking it, it returns the negated ooRexx error code. In addition, rexxc issues the following errors:

C.4.1. Error 127 - The rexxc command parameters are incorrect.

Explanation:

The rexxc utility was invoked with zero or more than two parameters. rexxc accepts the following parameters:

• To check the syntax of a REXX program: rexxc inputfile [-s]

• To translate a REXX program into a sourceless executable file: rexxc inputfile outputfile [-s] [-e]

• The -s option suppresses the copyright banner and the -e option generates the outputfile in base64-encoded format.

• On Windows the -s and -e options can alternatively be specified as /s and /e.

C.4.2. Error 128 - Output file name must be different from input file name.

C.4.3. Error 129 - SYNTAX: rexxc inputfile [outputfile] [-s] [-e]

C.4.4. Error 130 - Without outputfile rexxc only performs a syntax check.
Appendix D. Notices

Any reference to a non-open source product, program, or service is not intended to state or imply that only non-open source product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any Rexx Language Association (RexxLA) intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-open source product, program, or service.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-open source products was obtained from the suppliers of those products, their published announcements or other publicly available sources. RexxLA has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-RexxLA packages. Questions on the capabilities of non-RexxLA packages should be addressed to the suppliers of those products.

All statements regarding RexxLA's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

D.1. Trademarks

Open Object Rexx™ and ooRexx™ are trademarks of the Rexx Language Association.

The following terms are trademarks of the IBM Corporation in the United States, other countries, or both:

1-2-3
AIX
IBM
Lotus
OS/2
S/390
VisualAge

AMD is a trademark of Advance Micro Devices, Inc.

Intel, Intel Inside (logos), MMX and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

D.2. Source Code For This Document

The source code for this document is available under the terms of the Common Public License v1.0 which accompanies this distribution and is available in the appendix Appendix E, Common Public License Version 1.0. The source code is available at https://sourceforge.net/p/oorexx/code-0/HEAD/tree/docs/.

The source code for this document is maintained in DocBook SGML/XML format.

The railroad diagrams were generated with the help of "Railroad Diagram Generator" located at http://bottlecaps.de/rr/ui. Special thanks to Gunther Rademacher for creating and maintaining this tool.
Appendix E. Common Public License
Version 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS COMMON PUBLIC LICENSE (“AGREEMENT”). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT’S ACCEPTANCE OF THIS AGREEMENT.

E.1. Definitions

"Contribution" means:

1. in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and
2. in the case of each subsequent Contributor:
   a. changes to the Program, and
   b. additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution ‘originates’ from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents " mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

E.2. Grant of Rights

1. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.

2. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

3. Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement
of intellectual property rights or otherwise. As a condition to exercising the rights and licenses
granted hereunder, each Recipient hereby assumes sole responsibility to secure any other
intellectual property rights needed, if any. For example, if a third party patent license is required
to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license
before distributing the Program.

4. Each Contributor represents that to its knowledge it has sufficient copyright rights in its
Contribution, if any, to grant the copyright license set forth in this Agreement.

E.3. Requirements
A Contributor may choose to distribute the Program in object code form under its own license
agreement, provided that:

1. it complies with the terms and conditions of this Agreement; and
2. its license agreement:
   a. effectively disclaims on behalf of all Contributors all warranties and conditions, express and
      implied, including warranties or conditions of title and non-infringement, and implied warranties
      or conditions of merchantability and fitness for a particular purpose;
   b. effectively excludes on behalf of all Contributors all liability for damages, including direct,
      indirect, special, incidental and consequential damages, such as lost profits;
   c. states that any provisions which differ from this Agreement are offered by that Contributor
      alone and not by any other party; and
   d. states that source code for the Program is available from such Contributor, and informs
      licensees how to obtain it in a reasonable manner on or through a medium customarily used
      for software exchange.

When the Program is made available in source code form:

1. it must be made available under this Agreement; and
2. a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that
reasonably allows subsequent Recipients to identify the originator of the Contribution.

E.4. Commercial Distribution
Commercial distributors of software may accept certain responsibilities with respect to end users,
business partners and the like. While this license is intended to facilitate the commercial use of the
Program, the Contributor who includes the Program in a commercial product offering should do so in
a manner which does not create potential liability for other Contributors. Therefore, if a Contributor
includes the Program in a commercial product offering, such Contributor ("Commercial Contributor")
hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against
any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal
actions brought by a third party against the Indemnified Contributor to the extent caused by the acts
or omissions of such Commercial Contributor in connection with its distribution of the Program in
a commercial product offering. The obligations in this section do not apply to any claims or Losses
relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified
Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow
the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense
and any related settlement negotiations. The Indemnified Contributor may participate in any such
claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product
X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes
performance claims, or offers warranties related to Product X, those performance claims and
warranties are such Commercial Contributor’s responsibility alone. Under this section, the Commercial
Contributor would have to defend claims against the other Contributors related to those performance
claims and warranties, and if a court requires any other Contributor to pay any damages as a result,
the Commercial Contributor must pay those damages.

E.5. No Warranty
EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON
AN “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS
OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF
TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
Each Recipient is solely responsible for determining the appropriateness of using and distributing the
Program and assumes all risks associated with its exercise of rights under this Agreement, including
but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or
loss of data, programs or equipment, and unavailability or interruption of operations.

E.6. Disclaimer of Liability
EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY
CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION
LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE
OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGES.

E.7. General
If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect
the validity or enforceability of the remainder of the terms of this Agreement, and without further action
by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such
provision valid and enforceable.

If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to
software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by
that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation
is filed. In addition, if Recipient institutes patent litigation against any entity (including a cross-claim or
counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with
other software or hardware) infringes such Recipient’s patent(s), then such Recipient’s rights granted
under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient’s rights under this Agreement shall terminate if it fails to comply with any of the material
terms or conditions of this Agreement and does not cure such failure in a reasonable period of time
after becoming aware of such noncompliance. If all Recipient’s rights under this Agreement terminate,
Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable.
However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. IBM is the initial Agreement Steward. IBM may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.
Appendix F. Revision History

Revision 0-0  Aug 2016
Initial creation for 5.0
Index

Symbols
% (integer division operator), 19, 585
% method, 174, 427
& (AND logical operator) operator, 21
& method, 176
&& (exclusive OR operator), 21
&& method, 176
* (multiplication operator), 19, 585
* method, 174, 427
** method, 174, 427
** (power operator), 585
*-* tracing flag, 84
+ (addition operator), 19, 585
+ method, 174, 346, 426
+++ tracing flag, 85
, (comma)
    as a special character, 15
    as continuation character, 16
    in CALL instruction, 48
    in function calls, 443
    in parsing template list, 46, 579
    separator of arguments, 48, 443
- (subtraction operator), 19, 585
- method, 174, 346, 427
. (period)
    as place holder in parsing, 569
    causing substitution in variable names, 33
    in numbers, 584
.CONTEXT, 440
.DEBUGINPUT object, 439
.ENDOFLINE object, 437
.ENVIRONMENT object, 437
.ERROR object, 439
.FALSE object, 437
.INPUT object, 439
.LINE object, 440
.LOCAL object, 438
.METHODS StringTable, 441
.nil object, 437
.OUTPUT object, 439
.REQUIRES StringTable, 441
.RexxInfo object, 437
.ROUTINES StringTable, 441
.RS (return code)
    not set during interactive debug, 629
.RS object, 442
.TRACEOUTPUT object, 439
.TRUE object, 438
/ (division operator), 19, 585
/ method, 174, 427
// (remainder operator), 585
// method, 174, 427
: (colon)
    as a special character, 15
    in a label, 27
::ANNOTATE directive, 90
::ATTRIBUTE directive, 91
::CLASS directive, 94
::CONSTANT directive, 96
::METHOD directive, 97
::OPTIONS directive, 100
    ALL option, 101
    DIGITS option, 101
    ERROR option, 101
    FAILURE option, 101
    FORM option, 101
    FUZZ option, 101
    LOSTDIGITS option, 101
    NOPROLOG option, 102
    NOSTRING option, 101
    NOTREADY option, 101
    NOVALUE option, 101
    PROLOG option, 102
    TRACE option, 102
::REQUIRES directive, 102
::RESOURCE directive, 103
::ROUTINE directive, 104
; semicolon
    as a special character, 15
< (less than operator), 20
< method, 175, 390
<< (strictly less than operator), 20, 20
<< method, 175, 391
<<= (strictly less than or equal operator), 20
<<= method, 175, 391
<= (less than or equal operator), 20
<= method, 175, 391
<> (less than or greater than operator), 20
<> method
    of Class class, 122
    of Object class, 148
    of Orderable class, 390
    of Pointer class, 392
    of String class, 174
= (equal sign)
    assignment operator, 28
    equal operator, 20
    immediate debug command, 629
    in DO instruction, 50
    in LOOP instruction, 63
in parsing template, 572
= method
  of Class class, 122
  of Object class, 148
  of Orderable class, 390
  of Pointer class, 392
  of String class, 174
== (strictly equal operator), 20, 20, 588
== method
  of Class class, 122
  of Object class, 148
  of Orderable class, 391
  of Pointer class, 392
  of String class, 175
> (greater than operator), 20
> method, 174, 390
>.> tracing flag, 85
>< (greater than or less than operator), 20
>< method
  of Class class, 122
  of Object class, 148
  of Orderable class, 390, 390
  of Pointer class, 392
  of String class, 174, 174
>= (greater than or equal operator), 20
>= method, 175, 391
>= method, 175, 391
>> (strictly greater than operator), 20, 20
>> method, 175, 391
>>> (strictly greater than or equal operator), 20
>>> method, 175, 391
>A> tracing flag, 85
>C> tracing flag, 85
>E> tracing flag, 85
>F> tracing flag, 85
>I> tracing flag, 85
>K> tracing flag, 85
>L> tracing flag, 85
>M> tracing flag, 85
>N> tracing flag, 85
>O> tracing flag, 85
>P> tracing flag, 86
>R> tracing flag, 86
>V> tracing flag, 86
? method
  of String class, 177
? prefix on TRACE option, 83
[] method
  of Array class, 249
  of Bag class, 262
  of Collection class, 238
  of Directory class, 275
  of IdentityTable class, 280
  of List class, 284
  of MutableBuffer class, 373
  of Queue class, 293
  of Relation class, 299
  of Routine class, 168
  of Set class, 305
  of Stem class, 310
  of String class, 177
  of StringTable class, 314
  of Table class, 319
\ (NOT operator), 21
\ method, 176
< (not less than operator), 20
< method, 175, 391
<< (strictly not less than operator), 20
<< method, 175, 391
= (not equal operator), 20
= method
  of Class class, 122
  of Object class, 148
  of Pointer class, 392
  of String class, 175
\= (not strictly equal operator), 20, 20, 588
\= method, 175, 391
\> (not greater than operator), 20
\> method, 175, 391
\>> (strictly not greater than operator), 20
\>> method, 176, 391
| inclusive OR operator, 21
| method, 176
|| concatenation operator, 19
|| method, 148, 177
~ (tilde or twiddle), 4, 24
~~, 24
¬ (NOT operator), 21
¬< (not less than operator), 20
¬<< (strictly not less than operator), 20
\begin{itemize}
\item $\equiv$ (not equal operator), 20
\item $\equiv\equiv$ (not strictly equal operator), 20, 20, 588
\item $\rightarrow$ (not greater than operator), 20
\item $\rightarrow\rightarrow$ (strictly not greater than operator), 20
\end{itemize}

\subsection*{A}

\textbf{ABBREV function}
\begin{itemize}
\item description, 450
\item example, 450
\item using to select a default, 451
\end{itemize}

\textbf{abbrev method}
\begin{itemize}
\item of String class, 178
\end{itemize}

\textbf{abbreviations with ABBREV function}, 450

\textbf{ABS function}
\begin{itemize}
\item description, 451
\item example, 451
\end{itemize}

\textbf{abs method}
\begin{itemize}
\item of String class, 178
\end{itemize}

\textbf{absolute value}
\begin{itemize}
\item finding using the ABS function, 451
\item finding using the abs method, 178
\item used with power, 586
\end{itemize}

\textbf{absoluteFile method}
\begin{itemize}
\item of File class, 362
\end{itemize}

\textbf{absolutePath method}
\begin{itemize}
\item of File class, 362
\end{itemize}

\textbf{abstract class, definition}, 108

\textbf{ABSTRACT subkeyword}
\begin{itemize}
\item in a CLASS directive, 94
\item in a METHOD directive, 97
\item in an ATTRIBUTE directive, 91
\end{itemize}

\textbf{abuttal}, 19

\textbf{acquire method}
\begin{itemize}
\item of MutexSemaphore class, 388
\end{itemize}

\textbf{action taken when a condition is not trapped}, 595

\textbf{action taken when a condition is trapped}, 592, 595

\textbf{activate method}
\begin{itemize}
\item of Class class, 122
\end{itemize}

\textbf{active blocks}, 62

\textbf{activity}, 601

\textbf{add external function}, 483

\textbf{addClass method}
\begin{itemize}
\item of Package class, 158
\end{itemize}

\textbf{addDays method}
\begin{itemize}
\item of DateTime class, 351
\item of TimeSpan class, 429
\end{itemize}

\textbf{addHours method}
\begin{itemize}
\item of DateTime class, 351
\item of TimeSpan class, 430
\end{itemize}

\textbf{addMinutes method}
\begin{itemize}
\item of DateTime class, 351
\item of TimeSpan class, 430
\end{itemize}

\textbf{addPackage method}
\begin{itemize}
\item of Package class, 158
\end{itemize}

\textbf{addPublicClass method}
\begin{itemize}
\item of Package class, 159
\end{itemize}

\textbf{addPublicRoutine method}
\begin{itemize}
\item of Package class, 159
\end{itemize}

\textbf{ADDRESS function}
\begin{itemize}
\item description, 451
\item determining current environment, 451
\item example, 451
\end{itemize}

\textbf{ADDRESS instruction}
\begin{itemize}
\item description, 42
\item example, 45
\item issuing commands to, 42
\item settings saved during subroutine calls, 50
\end{itemize}

\textbf{address setting}, 46, 50

\textbf{addRoutine method}
\begin{itemize}
\item of Package class, 159
\end{itemize}

\textbf{addSeconds method}
\begin{itemize}
\item of DateTime class, 352
\item of TimeSpan class, 430
\end{itemize}

\textbf{add Weeks method}
\begin{itemize}
\item of DateTime class, 351
\item of TimeSpan class, 429
\end{itemize}

\textbf{addYears method}
\begin{itemize}
\item of DateTime class, 350
\end{itemize}

\textbf{Alarm class}, 328

\textbf{AlarmNotification class}, 331

\textbf{algebraic precedence}, 21

\textbf{allAt method}
\begin{itemize}
\item of Bag class, 262
\item of Relation class, 299
\end{itemize}

\textbf{allIndex method}
\begin{itemize}
\item of Bag class, 262
\item of Relation class, 300
\end{itemize}

\textbf{allIndexes method}
\begin{itemize}
\item of Array class, 250
\item of Bag class, 263
\item of Collection class, 238
\item of Directory class, 275
\item of IdentityTable class, 281
\item of List class, 284
\item of Queue class, 293
\item of Relation class, 300
\item of Set class, 305
\item of Stem class, 311
\item of StringTable class, 315
\item of Supplier class, 419
\item of Table class, 319
\end{itemize}
allItems method
  of Array class, 250
  of Bag class, 263
  of Collection class, 239
  of Directory class, 275
  of IdentityTable class, 281
  of List class, 284
  of Queue class, 293
  of Relation class, 300
  of Set class, 306
  of Stem class, 311
  of StringTable class, 315
  of Supplier class, 419
  of Table class, 319
alnum method
  of String class, 170
alpha method
  of String class, 171
alphabetical character word options in TRACE, 82
alphabetics
  checking with dataType, 193
  checking with DATATYPE, 464
  used in symbols, 13
alphanumerics
  checking with dataType, 193
  checking with DATATYPE, 464
altering
  flow within a repetitive loop, 61
  special variables, 38
  TRACE setting, 500
alternating exclusive scope access, 607
AND, logical operator, 21
ANDing character strings, 179, 454
annotation
  creation, 90
annotation method
  of Class class, 122
  of Method class, 143
  of Package class, 159
  of Routine class, 168
annotations method
  of Class class, 123
  of Method class, 143
  of Package class, 159
  of Routine class, 168
ANY subkeyword
  in a CALL instruction, 47, 592
  in a SIGNAL instruction, 79, 592
append method
  of Array class, 250
  of CircularQueue class, 268
  of List class, 284
  of MutableBuffer class, 374
  of OrderedCollection class, 243
  of Queue class, 294
  of String class, 179
APPEND subkeyword
  in an ADDRESS instruction, 42
appendAll method
  of OrderedCollection class, 243
architecture method
  of RexxInfo class, 401
ARG function
  description, 451
  example, 452
ARG instruction
  description, 46
  example, 46
ARG option of PARSE instruction, 67
ARG subkeyword
  in a PARSE instruction, 46, 66, 577
  in a USE instruction, 86
args method
  of RexxContext class, 398
arguments
  checking with ARG function, 451
  of functions, 46, 443
  of programs, 46
  of subroutines, 46
  passing in messages, 24
  passing to functions, 443, 444
  retrieving with ARG function, 451
  retrieving with ARG instruction, 46
  retrieving with PARSE ARG instruction, 67
arguments method
  of Message class, 134
  of StackFrame class, 415
ARGUMENTS subkeyword
  in a FORWARD instruction, 55
arithmetic
  basic operator examples, 586
  comparisons, 588
  errors, 589
  exponential notation, 587
  examples, 587
  numeric comparisons example
    examples, 589
  NUMERIC setting, 64
  operator examples, 586
  operators, 18, 584, 585
  overflow, 590
  precision, 585
  underflow, 590
array
  initialization, 30
  setting up, 33
  Array class, 247
ARRAY subkeyword
  in a FORWARD instruction, 55
  in a RAISE instruction, 73
array term, 25, 26
array method
  of InputStream class, 215
  of Stream class, 219
arrayOut method
  of OutputStream class, 217
  of Stream class, 219
assigning data to variables, 66
assignment
  description, 28, 29
  indicator (=), 28
  of compound variables, 33
  of stems variables, 30
  several assignments, 576
associative storage, 33
attribute
  creation, 91, 96
ATTRIBUTE subkeyword
  in a METHOD directive, 97
  in an ANNOTATE directive, 90
available method
  of StreamSupplier class, 418
  of Supplier class, 420
B
B2X function
  description, 453
  example, 453
b2x method
  of String class, 179
backslash, use of, 15, 21
Bag class, 261
base class for mixins, 108
Base option of DATE function, 466
base64
  decodeBase64 method, 195
  encodeBase64 method, 196
baseClass method
  of Class class, 123
baseDate method
  of DateTime class, 352
basic operator examples, 586
BEEP function
  description, 453
  example, 454
binary
  digits, 12
  strings
    description, 12
    implementation maximum, 13
    nibbles, 12
    to hexadecimal conversion, 179, 453
BITAND function
  description, 454
  example, 454
bitAnd method
  of String class, 179
BITOR function
  description, 455
  example, 455
bitOr method
  of String class, 180
bits checked using DATATYPE function, 464
bits checked using dataType method, 193
BITXOR function
  description, 455
  example, 455
bitXor method
  of String class, 180
blank method
  of String class, 171
blanks, 19
  adjacent to special character, 8
  in parsing, treatment of, 568
  removal with STRIP function, 494
  removal with strip method, 206
boolean operations, 21
bottom of program reached during execution, 54
bounded buffer, 612
Buffer class, 332
built-in functions
  ABBREV, 450
  ABS, 451
  ADDRESS, 451
  ARG, 451
  B2X, 453
  BEEP, 453
  BITAND, 454
  BITOR, 455
BITXOR, 455
C2D, 456
C2X, 456
calling, 48
CENTER, 457
CENTRE, 457
CHANGESTR, 457
CHARIN, 458
CHAROUT, 459
CHARS, 460
COMPARE, 460
CONDITION, 460
COPIES, 462
COUNTSTR, 462
D2C, 463
D2X, 463
DATATYPE, 464
DATE, 465
definition, 48
DELSTR, 469
DELWORD, 470
DIGITS, 470
DIRECTORY, 470
ENDLOCAL, 471
ERRORTEXT, 471
FILESPEC, 472
FORM, 472
FORMAT, 473
FUZZ, 474
INSERT, 474
LASTPOS, 474
LEFT, 475
LENGTH, 475
LINEIN, 475
LINEOUT, 477
LINES, 478
LOWER, 479
MAX, 479
MIN, 480
OVERLAY, 480
POS, 480
QUALIFY, 481
QUEUED, 481
RANDOM, 481
REVERSE, 482
RIGHT, 482
RXFUNCADD, 483
RXFUCNDROP, 483
RXFUNCQUERY, 483
RXQUEUE, 484
SELOCAL, 485
SIGN, 486
SOURCELINE, 486
SPACE, 486
STREAM, 487
STRIP, 494
SUBSTR, 495
SUBWORD, 496
SYMBOL, 496
TIME, 497
TRACE, 500
TRANSLATE, 501
TRUNC, 502
UPPER, 502
USERID, 503
VALUE, 503
VAR, 505
VERIFY, 506
WORD, 506
WORDINDEX, 507
WORDLENGTH, 507
WORDPOS, 507
WORDS, 508
X2B, 508
X2C, 509
X2D, 509
XRANGE, 510
built-in object
 CONTEXT object, 440
 DEBUGINPUT object, 439
 ENDOFLINE object, 437
 ENVIRONMENT object, 437
 ERROR object, 439
 FALSE object, 437
 INPUT object, 439
 LINE object, 440
 LOCAL object, 438
 METHODS object, 441
 nil object, 437
 OUTPUT object, 439
 RESOURCES object, 441
 RexxInfo object, 437
 ROUTINES object, 441
 RS object, 442
 STDERR object, 439
 STDIN object, 439
 STDOUT object, 440
 STDQUE object, 440
 SYSCARGS object, 441
 TRACEOUTPUT object, 439
 TRUE object, 438
BY phrase of DO instruction, 51
BY subkeyword
 in a DO instruction, 50, 641
 in a LOOP instruction, 63, 641
C
C2D function
strings, exclusive-ORing, 180, 455
strings, ORing, 180, 455
to decimal conversion, 181, 456
to hexadecimal conversion, 181, 456
class using UPPER function, 502
class character input and output, 619, 630
class character input streams, 620
class character output streams, 620
CHARIN function
description, 458
element, 458
class charIn method
do of InputStream class, 216
do of OutputStream class, 217
do of Stream class, 219
class CHARIN method
class role in input and output, 620
CHAROUT function
description, 459
element, 459
class charOut method
do of InputStream class, 216
do of OutputStream class, 217
do of Stream class, 220
class CHAROUT method
class role in input and output, 620
CHARS function
description, 460
element, 460
class chars method
do of InputStream class, 216
do of OutputStream class, 217
do of Stream class, 220
class CHARSM method
class role in input and output, 620
checking arguments with ARG function, 451
CircularQueue class, 267
civilTime method
do of DateTime class, 354
class class
Alarm class, 328
AlarmNotification class, 331
Array class, 247
Bag class, 261
Buffer class, 332
CaselessColumnComparator class, 335
CaselessComparator class, 334
CaselessDescendingComparator class, 337
CircularQueue class, 267
Class class, 121
ColumnComparator class, 334
Comparable class, 332
Comparator class, 333
DateTime class, 339
definition, 5
DescendingComparator class, 336
Directory class, 273
EventSemaphore class, 357
File class, 360
IdentityTable class, 279
InputOutputStream class, 215
InputStream class, 215
InvertingComparator class, 337
List class, 283
Message class, 132
MessageNotification class, 371
Method class, 142
Monitor class, 371
MutableBuffer class, 373
MutexSemaphore class, 387
NumericComparator class, 338
Object class, 147
Orderable class, 390
OutputStream class, 216
Package class, 157
Pointer class, 391
Properties class, 288
Queue class, 292
RegularExpression class, 392
Relation class, 298
RexxContext class, 397
RexxInfo class, 400
RexxQueue class, 409
Routine class, 166
Set class, 304
Singleton class, 413
StackTrace class, 415
Stem class, 308
Stream class, 218
StreamSupplier class, 417
String class, 169
StringTable class, 313
Supplier class, 419
Table class, 318
Ticker class, 421
TimeSpan class, 424
types
abstract, 108
metaclass, 108
mixin, 108
object, 107
Validate class, 431
VariableReference class, 434
WeakReference class, 435
Class class, 121
class method
of Object class, 148
class methods, 107
CLASS subkeyword
  in a FORWARD instruction, 55
  in a METHOD directive, 97
  in an ANNOTATE directive, 90
  in an ATTRIBUTE directive, 91
classes method
  of Package class, 159
classType method
  of Validate class, 431
clauses
  assignment, 27, 28
  commands, 28
  continuation of, 16
  description, 8, 28, 29
  directives, 27
  extended assignment, 29
  instructions, 27
  keyword instructions, 28
  labels, 27
  message instructions, 28
  null, 26
close method
  of InputStream class, 216
  of OutputStream class, 217
  of Stream class, 220
CMD command environment, 45
cntrl method
  of String class, 171
code page, 9
codes, error, 651
collating sequence using XRANGE, 510
Collection class, 238
  organization, 237
Collection classes, 236
COLLECTOR example program, 626
colon
  as a special character, 15
  as label terminators, 27
  in a label, 27
ColumnComparator class, 334
combining string and positional patterns, 579
comma
  as a special character, 15
  as continuation character, 16
  in CALL instruction, 48
  in function calls, 443
  in parsing template list, 46, 579
  separator of arguments, 48, 443
command
  alternative destinations, 38
  clause, 28
  destination of, 42
  errors, trapping, 592
  issuing to host, 38
command method
  of Stream class, 220
COMMAND method, 619
comments, 9
  line comment, 9
  standard comment, 9
Common Public License, 699
Comparable class, 332
Comparator class, 333
COMPARE function
  description, 460
  example, 460
compare method
  of CaselessColumnComparator class, 336
  of CaselessComparator class, 334
  of CaselessDescendingComparator class, 337
  of ColumnComparator class, 335
  of Comparator class, 333
  of DescendingComparator class, 336
  of InvertingComparator class, 338
  of NumericComparator class, 339
  of String class, 190
compareTo method
  of Comparable class, 333
  of DateTime class, 347
  of File class, 363
  of Orderable class, 390
  of String class, 190
  of TimeSpan class, 427
comparisons
  description, 20
  numeric, example, 589
  of numbers, 20, 588
  of strings, 20, 183, 190, 190, 460
completed method
  of Message class, 134
compound
  symbols, 33
  variable
    description, 33
    setting new value, 31
concatenation
  abuttal, 19
  as concatenation operator, 19
  blank, 19
  of strings, 18
  operator
    ||, 15, 19
conceptual overview of parsing, 567
concurrency
  alternating exclusive scope access, 607
  conditional, 607
D
D2C function
  description, 463
  example, 463
d2c method
  of String class, 192
D2X function
  description, 463
  example, 463
d2x method
  of String class, 193
data
  abstraction, 5
  encapsulation, 4
  modularization, 1
  objects, 17
  terms, 17
DATATYPE function
  description, 464
  example, 465
dataType method
  of String class, 193
date and version of the language processor, 69
DATE function
  description, 465
  example, 467
date method
  of DateTime class, 356
  of RexxInfo class, 402
DateTime class, 339
day method
  of DateTime class, 349
dayMicroseconds method
  of DateTime class, 350
dayMinutes method
  of DateTime class, 350
dayName method
  of DateTime class, 353
days method
  of TimeSpan class, 427
Days option of DATE function, 466
daySeconds method
  of DateTime class, 350
daysInMonth method
  of DateTime class, 357
daysInYear method
  of DateTime class, 357
debug input object, 439
debug interactive, 81
debug method
  of RexxInfo class, 402
decimal
  integer, 584
to character conversion, 192, 463
to hexadecimal conversion, 193, 463
decodeBase64 method
  of String class, 195
default
  character streams, 619
  concurrency, 603
  environment, 37
  search order for methods, 113
  selecting with ABBREV function, 450
  selecting with abbrev method, 178
  selecting with caselessAbbrev method, 182
defaultName method
  of Class class, 123
  of Object class, 149
define method
  of Class class, 123
definedMethods method
  of Package class, 160
defineMethods method
  of Class class, 124
delayed state
  description, 592
  of NOTREADY condition, 627
DELEGATE subkeyword
  in a METHOD directive, 97
  in an ATTRIBUTE directive, 91
delete class method
  of RexxQueue class, 410
delete method
  of Array class, 251
  of Class class, 125
  of File class, 363
  of List class, 285
  of MutableBuffer class, 379
  of OrderedCollection class, 244
  of Queue class, 294
  of RexxQueue class, 411
deleting
  part of a string, 195, 469
  words from a mutablebuffer, 379
  words from a string, 196, 470
DELSTR function
  description, 469
  example, 469
delStr method
  of MutableBuffer class, 379
  of String class, 195
DELWORD function
  description, 470
  example, 470
delWord method
  of MutableBuffer class, 379
  of String class, 196
derived names of variables, 33
DescendingComparator class, 336
description method
  of Stream class, 226
DESCRIPTION subkeyword
  in a RAISE instruction, 73
destination method
  of Monitor class, 372
difference method
  of Bag class, 263
  of Collection class, 239
  of IdentityTable class, 281
  of OrderedCollection class, 244
  of Relation class, 300
  of Set class, 306
  of Table class, 319
digit method
  of String class, 171
DIGITS function
description, 470
example, 470
digits method
  of Package class, 160
  of RexxContext class, 398
  of RexxInfo class, 402
DIGITS option of NUMERIC instruction, 584
DIGITS subkeyword
  in a NUMERIC instruction, 64, 585
dimension method
  of Array class, 251
dimensions method
  of Array class, 252
directives
  ::ANNOTATE, 90
  ::ATTRIBUTE, 91
  ::CLASS, 94
  ::CONSTANT, 96
  ::METHOD, 97
  ::OPTIONS, 100
  ::REQUIRES, 102
  ::RESOURCE, 103
  ::ROUTINE, 104
Directory class, 273
DIRECTORY function
description, 470
example, 471
directorySeparator method
  of RexxInfo class, 403
disjoint method
  of Collection class, 239
division operator, 19
DO instruction
description, 50
example, 641
drop external function, 483
DROP instruction
description, 52
example, 53
DROP keyword
  in a RXSUBCOM command, 636
duration method
  of TimeSpan class, 427
dyadic operators, 18
E
  early reply, 75, 601
  elapsed method
    of DateTime class, 356
  elapsed-time clock
    measuring intervals with, 497
    saved during subroutine calls, 50
ELSE
  as free standing clause, 42
ELSE subkeyword
  in an IF instruction, 58
empty method
  of Array class, 252
  of Bag class, 264
  of Directory class, 275
  of List class, 285
  of Queue class, 294
  of Relation class, 300
  of RexxQueue class, 411
  of Stem class, 311
  of StringTable class, 315
  of String class, 316
  of StringTable class, 315
   encapsulation of data, 4
   encodeBase64 method
     of String class, 196
END
  as free standing clause, 42
END clause
  specifying control variable, 641
END subkeyword
  in a DO instruction, 50
  in a LOOP instruction, 63
  in a SELECT instruction, 77
ENDLOCAL function
description, 471
example, 471
endofline method
  of RexxInfo class, 403
endsWith method
  of MutableBuffer class, 380
  of String class, 196
engineering notation, 588
ENGINEERING subkeyword
  in a NUMERIC instruction, 64
enhanced method
of Class class, 125
entry method
  of Directory class, 276
  of StringTable class, 315
environment, 42
  addressing of, 43
default, 46
determining current using ADDRESS function, 451
equal
  operator, 20
  sign
    in parsing templates, 571, 572
to indicate assignment, 15, 28
equality, testing of, 20
equals method
  of String class, 196
equivalent method
  of Collection class, 239
error
  definition, 38
during execution of functions, 448
during stream input and output, 626
from commands, 38
messages
  list, 651
  retrieving with ERRORTEXT, 651
syntax, 651
traceback after, 82
trapping, 592
error codes, 651
error messages and codes, 651
ERROR subkeyword
  in a CALL instruction, 47, 592, 597
  in a RAISE instruction, 73
  in a SIGNAL instruction, 79, 592, 597
  in an ADDRESS instruction, 42
errorCondition method
  of Message class, 134
ERRORTEXT function
  description, 471
  example, 471
European option of DATE function, 466
europeanDate method
  of DateTime class, 353
evaluation of expressions, 18
EventSemaphore class, 357
examples
  ::ANNOTATE directive, 90
  ::ATTRIBUTE directive
    EXTERNAL option, 93
  ::CLASS directive, 95
  ::METHOD directive
    EXTERNAL option, 99
  ::RESOURCE directive, 103
  ::ROUTINE directive, 104
    EXTERNAL option, 105
    ABBREV function, 450
    abbrev method, 178, 182
    ABS function, 451
    abs method, 178
    acquire method, 388
    ADDRESS function, 451
    ADDRESS instruction, 45
    allIndexes method, 250
    allItems method, 250
    append method, 251
    ARG function, 452
    ARG instruction, 46
    arithmetic methods of DateTime class, 347, 427
    arithmetic methods of String class, 174
    at method, 251
    attachment method, 421
    B2X function, 453
    b2x method, 179
    basic operator examples, 586
    BEEP function, 454
    BITAND function, 454
    bitAnd method, 180
    BITOR function, 455
    bitOr method, 180
    BITXOR function, 455
    bitXor method, 180
    C2D function, 456
    c2d method, 181
    C2X function, 457
    c2x method, 182
    CALL instruction, 49
    cancel method, 422
    canceled/cancelled method, 422
    caselessChangeStr method, 183
    CaselessColumnComparator class, 336
    CaselessComparator class, 334
    caselessCompareTo method, 184
    CaselessDescendingComparator class, 337
    caselessEquals method, 185
    caselessLastPos method, 186
    caselessMatch method, 186
    caselessMatchChar method, 187
    caselessPos method, 187
    ceiling method, 188
    CENTER function, 457
    center method, 189
    CENTRE function, 457
    centre method, 189
    CHANGESTR function, 457
    changeStr method, 189
CHARIN function, 458
CHAROUT function, 459
CHARS function, 460
COLLECTOR program, 626
ColumnComparator class, 335
combining positional pattern and parsing into words, 575
combining string and positional patterns, 580
combining string pattern and parsing into words, 574
command method
  OPEN option, 223
  QUERY DATETIME option, 224
  QUERY EXISTS option, 225
  QUERY HANDLE option, 225
SEEK option, 224
Comparator class, 334
COMPARE function, 460
cmpare method, 183, 190
compareTo method, 190
comparison methods of String class, 175
concatenation methods of String class, 177
CONDITION function, 462
continuation, 17
COPIES function, 462
copies method, 192
copy method, 148
COUNTSTR function, 462
countStr method, 185, 192
create method, 410
D2C function, 463
d2c method, 192
D2X function, 463
d2x method, 193
DATATYPE function, 465
dataType method, 194
DATE function, 467
decodeBase64 method, 195
defaultName method, 123
define method, 124
delete method, 125, 251, 294
DELSTR function, 469
delStr method, 195
DELWORD function, 470
delWord method, 196, 196
DescendingComparator class, 337
DIGITS function, 470
dimension method, 251
DIRECTORY function, 471
DO instruction, 641
DROP instruction, 53
empty method, 252
encodeBase64 method, 196
ENDLOCAL function, 471
enhanced method, 125
equals method, 197
ERRORTEXT function, 471
executable method, 403
exists method, 411
EXIT instruction, 53
exponential notation, 587
EXPOSE instruction, 54
expressions, 22
extension method, 364
FILECOPY program, 625
FILESPEC function, 472
fill method, 252
first method, 252
floor method, 197
FORM function, 472
FORMAT function, 473
format method, 198
FORWARD instruction, 56
fromOrdinalDate method, 342
fromWeekNumberDate method, 344
FUZZ function, 474
GUARD instruction, 57
halt method, 134
hasIndex method, 253
hasItem method, 253
id method, 126
IF instruction, 58
index method, 254
inherit method, 126
init method, 423
INSERT function, 474
insert method
  of Array class, 254
  of List class, 286
  of Queue class, 295
  of String class, 199
instanceMethods method, 151
internalMaxNumber method, 405
internalMinNumber method, 405
INTERPRET instruction, 60, 60
interval method, 423
InvertingComparator class, 338
isEmpty method, 255
isNil method, 152
isPosted method, 358
items method, 255
ITERATE instruction, 61
last method, 255
lastAccessed attribute, 366
LASTPOS function, 475
lastPos method, 199, 380
LEAVE instruction, 62
LEFT function, 475
STRIP function, 495
strip method, 207
subclass method, 131
SUBSTR function, 495
subStr method, 207
SUBWORD function, 496
subWord method, 208
subWords method, 208
superClass method, 131
superClasses method, 132
supplier method, 259
SYMBOL function, 496
SysCArgs Array, 440
SysCurPos, 519
SysDriveInfo, 520
SysDriveMap, 521
SysDumpVariables, 522
SysFileCopy, 522
SysFileDelete, 523
SysFileMove, 524
SysFileSearch, 525
SysFileSystemType, 526
SysFileTree, 529
SysFormatMessage, 531
SysGetErrorText, 533
SysGetFileDateTime, 534
SysGetLongPathName, 535
SysGetMessage, 535
SysGetMessageX, 536
SysGetShortPathName, 536
SysIni, 538
SysMkDir, 545
SysRmDir, 548
SysSearchPath, 549
SysSetDefaultPrinter, 565
SysSetFileDateTime, 550
SysSleep, 552
SysStemCopy, 554
SysStemDelete, 554
SysStemSort, 556
SysTemporaryFileName, 558
SysTextScreenRead, 558
SysTextScreenSize, 560
SysWinGetDefaultPrinter, 565
SysWinGetPrinters, 565
templates containing positional patterns, 571
templates containing string patterns, 570
temporaryPath method, 362
TIME function, 499, 499
toString method, 260
TRACE function, 501
TRACE instruction, 84
TRANSLATE function, 501
translate method, 209
triggered method, 141
TRUNC function, 502
trunc method, 209
type method, 417
uninherit method, 132
upper function, 503
upper method, 210
USE instruction, 87
USE LOCAL instruction, 89
using a variable as a string pattern, 576
using an expression as a positional pattern, 576
VALUE function, 503, 505
value method, 435
VAR function, 505
VERIFY function, 506
verify method, 211, 386
wait method, 359
weekNumber method, 347
weekNumberDate method, 348
weekNumberYear method, 348
weeksInYear method, 348
WORD function, 507
word method, 211
WORDINDEX function, 507
wordIndex method, 211
WORLDLENGTH function, 507
wordLength method, 212
WORDPOS function, 508
wordPos method, 188, 212
WORDS function, 508
words method, 212
X2B function, 508
x2b method, 213
X2C function, 509
x2c method, 213
X2D function, 509, 510
x2d method, 214
XRANGE function, 511
yearDay method, 353
exception conditions saved during subroutine
calls, 50
exclusive OR operator, 21
exclusive-ORing character strings together, 180,
455
executable method
of RexxContext class, 398
of RexxInfo class, 403
of StackFrame class, 415
execution
by language processor, 1
of data, 59
exists method
of File class, 364
of RexxQueue class, 411
EXIT instruction
description, 53
example, 53
EXIT subkeyword
in a RAISE instruction, 73
exponential notation
description, 587
example, 14, 587
exponentiation
description, 587
operator, 19
EXPOSE instruction
description, 54
example, 54
EXPOSE option of PROCEDURE instruction, 69
EXPOSE subkeyword
in a PROCEDURE instruction, 69
exposed variable, 69
expressions
evaluation, 18
examples, 22
parsing of, 68
results of, 18
tracing results of, 83
extended assignments, 29
extension method
of File class, 364
external character streams, 619
external data queue
counting lines in, 481
creating and deleting queues, 484
description, 621
naming and querying queues, 484
reading from with PULL, 71
RXQUEUE function, 484
writing to with PUSH, 72
writing to with QUEUE, 72
external functions
description, 444
functions
description, 443
search order, 444
external routine, 48
EXTERNAL subkeyword
in a METHOD directive, 97
in a ROUTINE directive, 104
LIBRARY routine, 104
REGISTERED routine, 104
in an ATTRIBUTE directive, 91
external subroutines, 444
external variables
access with VALUE function, 503
extracting
substring, 207, 495
word from a mutable buffer, 386
word from a string, 211, 506
words from a mutable buffer, 387
words from a string, 212, 508
extracting words with subWord, 207, 384
extracting words with subWords, 208, 384

F
FAILURE subkeyword
in a CALL instruction, 47, 593, 597
in a RAISE instruction, 73
in a SIGNAL instruction, 79, 593, 597
failure, definition, 38
FIFO (first-in/first-out) stacking, 72
File class, 360
file name, extension, path of program, 68
FILECOPY example program, 625
files, 619
FILESPEC function
description, 472
example, 472
fill method
of Array class, 252
findClass method
of Package class, 160
finding
mismatch using caselessCompare, 183
mismatch using compare, 190
mismatch using COMPARE, 460
mismatch using compareTo, 190
string in a MutableBuffer, 382
string in another string, 203, 480
string length, 200, 475
word length, 212, 387, 507
findNamespace method
of Package class, 160
findProgram method
of Package class, 161
findPublicClass method
of Package class, 161
findPublicRoutine method
of Package class, 161
findFirst method
of Package class, 161
first method
of Array class, 252
of List class, 285
of OrderedCollection class, 244
of Queue class, 294
firstItem method
of Array class, 253
of List class, 285
of OrderedCollection class, 244
of Queue class, 294
flag, tracing
*:*, 84
+++8, 85
>><, 85
>=>, 85
>>>, 85
>A>, 85
>C>, 85
>E>, 85
>F>, 85
>I>, 85
>K>, 85
>L>, 85
>M>, 85
>N>, 85
>O>, 85
>P>, 86
>R>, 86
>V>, 86
floor method
of String class, 197
flow of control
unusual, with CALL, 591
unusual, with SIGNAL, 591
with CALL and RETURN construct, 47
with DO construct, 50
with IF construct, 58
with LOOP construct, 63
with SELECT construct, 77
flush method
of Stream class, 227
FOR phrase of DO instruction, 51
FOR subkeyword
in a DO instruction, 50
in a LOOP instruction, 63
FOREVER phrase of DO instruction, 50
FOREVER phrase of LOOP instruction, 63
FOREVER repetitor on DO instruction, 52
FOREVER subkeyword
in a DO instruction, 50, 641, 644
in a LOOP instruction, 63, 641, 644
FORM function
description, 472
example, 472
form method
of Package class, 162
of RexxContext class, 398
of RexxInfo class, 403
FORM option of NUMERIC instruction, 65
FORM subkeyword
in a NUMERIC instruction, 64, 588
FORMAT function
description, 473
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>of DateTime class</td>
<td>344</td>
</tr>
<tr>
<td>fromUsaDate method</td>
<td>343</td>
</tr>
<tr>
<td>fromUtcIsoDate method</td>
<td>345</td>
</tr>
<tr>
<td>fromWeekNumberDate method</td>
<td>343</td>
</tr>
<tr>
<td>Full option of DATE function</td>
<td>466</td>
</tr>
<tr>
<td>Full option of Time function</td>
<td>497</td>
</tr>
<tr>
<td>fullDate method</td>
<td>355</td>
</tr>
<tr>
<td>functions</td>
<td>443</td>
</tr>
<tr>
<td>ABBREV, 450</td>
<td></td>
</tr>
<tr>
<td>ABS, 451</td>
<td></td>
</tr>
<tr>
<td>ADDRESS, 451</td>
<td></td>
</tr>
<tr>
<td>ARG, 451</td>
<td></td>
</tr>
<tr>
<td>B2X, 453</td>
<td></td>
</tr>
<tr>
<td>BEEP, 453</td>
<td></td>
</tr>
<tr>
<td>BITAND, 454</td>
<td></td>
</tr>
<tr>
<td>BITOR, 455</td>
<td></td>
</tr>
<tr>
<td>BITXOR, 455</td>
<td></td>
</tr>
<tr>
<td>built-in, 449</td>
<td></td>
</tr>
<tr>
<td>built-in, description, 450</td>
<td></td>
</tr>
<tr>
<td>C2D, 456</td>
<td></td>
</tr>
<tr>
<td>C2X, 456</td>
<td></td>
</tr>
<tr>
<td>call, definition, 443</td>
<td></td>
</tr>
<tr>
<td>calling, 443</td>
<td></td>
</tr>
<tr>
<td>CENTER, 457</td>
<td></td>
</tr>
<tr>
<td>CENTRE, 457</td>
<td></td>
</tr>
<tr>
<td>CHANGESTR, 457</td>
<td></td>
</tr>
<tr>
<td>CHARIN, 458</td>
<td></td>
</tr>
<tr>
<td>CHAROUT, 459</td>
<td></td>
</tr>
<tr>
<td>CHRAS, 460</td>
<td></td>
</tr>
<tr>
<td>COMPARE, 460</td>
<td></td>
</tr>
<tr>
<td>CONDITION, 460</td>
<td></td>
</tr>
<tr>
<td>COPIES, 462</td>
<td></td>
</tr>
<tr>
<td>COUNTSTR, 462</td>
<td></td>
</tr>
<tr>
<td>D2C, 463</td>
<td></td>
</tr>
<tr>
<td>D2X, 463</td>
<td></td>
</tr>
<tr>
<td>DATATYPE, 464</td>
<td></td>
</tr>
<tr>
<td>DATE, 465</td>
<td></td>
</tr>
<tr>
<td>definition, 443</td>
<td></td>
</tr>
<tr>
<td>DELSTR, 469</td>
<td></td>
</tr>
<tr>
<td>DELWORD, 470</td>
<td></td>
</tr>
<tr>
<td>description, 443</td>
<td></td>
</tr>
<tr>
<td>DIGITS, 470</td>
<td></td>
</tr>
<tr>
<td>DIRECTORY, 470</td>
<td></td>
</tr>
<tr>
<td>ENDLOCAL, 471</td>
<td></td>
</tr>
<tr>
<td>ERRORTEXT, 471</td>
<td></td>
</tr>
<tr>
<td>external, 444</td>
<td></td>
</tr>
<tr>
<td>FILESPEC, 472</td>
<td></td>
</tr>
<tr>
<td>forcing built-in or external reference, 444</td>
<td></td>
</tr>
<tr>
<td>FORM, 472</td>
<td></td>
</tr>
<tr>
<td>FORMAT, 473</td>
<td></td>
</tr>
<tr>
<td>FUZZ, 474</td>
<td></td>
</tr>
<tr>
<td>INSERT, 474</td>
<td></td>
</tr>
<tr>
<td>internal, 443</td>
<td></td>
</tr>
<tr>
<td>LASTPOS, 474</td>
<td></td>
</tr>
<tr>
<td>LEFT, 475</td>
<td></td>
</tr>
<tr>
<td>LENGTH, 475</td>
<td></td>
</tr>
<tr>
<td>LINEIN, 475</td>
<td></td>
</tr>
<tr>
<td>LINEOUT, 477</td>
<td></td>
</tr>
<tr>
<td>LINES, 478</td>
<td></td>
</tr>
<tr>
<td>logical bit operations, 454, 455, 455</td>
<td></td>
</tr>
<tr>
<td>LOWER, 479</td>
<td></td>
</tr>
<tr>
<td>MAX, 479</td>
<td></td>
</tr>
<tr>
<td>MIN, 480</td>
<td></td>
</tr>
<tr>
<td>numeric arguments of, 589</td>
<td></td>
</tr>
<tr>
<td>OVERLAY, 480</td>
<td></td>
</tr>
<tr>
<td>POS, 480</td>
<td></td>
</tr>
<tr>
<td>QUALIFY, 481</td>
<td></td>
</tr>
<tr>
<td>QUEUED, 481</td>
<td></td>
</tr>
<tr>
<td>RANDOM, 481</td>
<td></td>
</tr>
<tr>
<td>return from, 76</td>
<td></td>
</tr>
<tr>
<td>REVERSE, 482</td>
<td></td>
</tr>
<tr>
<td>RIGHT, 482</td>
<td></td>
</tr>
<tr>
<td>RXFUNCADD, 483</td>
<td></td>
</tr>
<tr>
<td>RXFUNCDROP, 483</td>
<td></td>
</tr>
<tr>
<td>RXFUNQUERY, 483</td>
<td></td>
</tr>
<tr>
<td>RXQUEUE, 484</td>
<td></td>
</tr>
<tr>
<td>SETLOCAL, 485</td>
<td></td>
</tr>
<tr>
<td>SIGN, 486</td>
<td></td>
</tr>
<tr>
<td>SOURCeline, 486</td>
<td></td>
</tr>
<tr>
<td>SPACE, 486</td>
<td></td>
</tr>
<tr>
<td>STREAM, 487</td>
<td></td>
</tr>
<tr>
<td>STRIP, 494</td>
<td></td>
</tr>
<tr>
<td>SUBSTR, 495</td>
<td></td>
</tr>
<tr>
<td>SUBWORD, 496</td>
<td></td>
</tr>
<tr>
<td>SYMBOL, 496</td>
<td></td>
</tr>
<tr>
<td>TIME, 497</td>
<td></td>
</tr>
<tr>
<td>TRACE, 500</td>
<td></td>
</tr>
<tr>
<td>TRANSLATE, 501</td>
<td></td>
</tr>
<tr>
<td>TRUNC, 502</td>
<td></td>
</tr>
<tr>
<td>UPPER, 502</td>
<td></td>
</tr>
<tr>
<td>USERID, 503</td>
<td></td>
</tr>
<tr>
<td>VALUE, 503</td>
<td></td>
</tr>
<tr>
<td>VAR, 505</td>
<td></td>
</tr>
<tr>
<td>variables in, 69</td>
<td></td>
</tr>
<tr>
<td>VERIFY, 506</td>
<td></td>
</tr>
<tr>
<td>WORD, 506</td>
<td></td>
</tr>
<tr>
<td>WORDINDEX, 507</td>
<td></td>
</tr>
<tr>
<td>WORDLENGTH, 507</td>
<td></td>
</tr>
<tr>
<td>WORDPOS, 507</td>
<td></td>
</tr>
<tr>
<td>WORDS, 508</td>
<td></td>
</tr>
<tr>
<td>X2B, 508</td>
<td></td>
</tr>
<tr>
<td>X2C, 509</td>
<td></td>
</tr>
<tr>
<td>X2D, 509</td>
<td></td>
</tr>
<tr>
<td>XRANGE, 510</td>
<td></td>
</tr>
</tbody>
</table>
FUZZ
  controlling numeric comparison, 589
  instruction, 589
FUZZ function
  description, 474
  example, 474
fuzz method
  of Package class, 162
  of RexxContext class, 399
  of RexxInfo class, 404
FUZZ option of NUMERIC instruction, 65
FUZZ subkeyword
  in a NUMERIC instruction, 64, 589

G
  general concepts, 1, 42
get method
  of RexxQueue class, 411
GET subkeyword
  in an ATTRIBUTE directive, 91
getBufferSize method
  of MutableBuffer class, 380
getLogical method
  of Properties class, 290
getProperty method
  of Properties class, 290
getting value with VALUE, 503
getWhole method
  of Properties class, 290
global variables
  access with VALUE function, 504
GOTO, unusual, 591
graph method
  of String class, 171
greater than operator, 20
greater than operator (>), 20
greater than or equal operator, 20
greater than or equal to operator (>=), 20
greater than or less than operator, 20
greater than or less than operator (><), 20
group, DO, 641
grouping instructions to run repetitively, 50, 63
GUARD instruction
  description, 57
  example, 57
guarded methods, 607
GUARDED subkeyword
  in a METHOD directive, 97
  in an ATTRIBUTE directive, 91

H
  halt method
    of Message class, 134
HALT subkeyword
  in a CALL instruction, 47, 593, 597
  in a SIGNAL instruction, 79, 593, 597
halt, trapping, 593
hasEntry method
  of Directory class, 276
  of StringTable class, 315
hasError method
  of Message class, 135
hashbang, 40
hashCode method
  of DateTime class, 350
  of File class, 364
  of Object class, 149
  of String class, 198
  of TimeSpan class, 429
hasIndex method
  of Array class, 253
  of Bag class, 264
  of Collection class, 239
  of Directory class, 276
  of IdentityTable class, 281
  of List class, 285
  of Queue class, 295
  of Relation class, 301
  of Set class, 306
  of Stem class, 311
  of StringTable class, 315
  of Table class, 319
hasItem method
  of Array class, 253
  of Bag class, 264
  of Collection class, 240
  of Directory class, 276
  of IdentityTable class, 281
  of List class, 286
  of Queue class, 295
  of Relation class, 301
  of Set class, 306
  of Stem class, 311
  of StringTable class, 316
  of Table class, 320
hasMethod method
  of Object class, 149
hasResult method
  of Message class, 135
hexadecimal
  checking with dataType, 193
  checking with DATATYPE, 464
digits, 12
strings
  description, 12
  implementation maximum, 12
to binary, converting with X2B, 213, 508
to character, converting with X2C, 213, 509
to decimal, converting with X2D, 214, 509
host commands
  issuing commands to underlying operating
  system, 38
hours calculated from midnight, 498
hours method
  of DateTime class, 349
  of TimeSpan class, 428
id method
  of Class class, 126
identityHash method
  of Object class, 150
IdentityTable class, 279
IF instruction
  description, 58
  example, 58
implementation maximum
  binary strings, 13
  hexadecimal strings, 12
  literal strings, 12
  numbers, 14
  TIME function, 500
implied semicolons, 16
importedClasses method
  of Package class, 162
importedPackages method
  of Package class, 162
importedRoutines method
  of Package class, 162
imprecise numeric comparison, 588
indentation during tracing, 84
index method
  of Array class, 254
  of Bag class, 264
  of Collection class, 240
  of Directory class, 276
  of IdentityTable class, 281
  of List class, 286
  of Queue class, 295
  of Relation class, 301
  of Set class, 306
  of Stem class, 311
  of StreamSupplier class, 418
  of StringTable class, 316
  of Supplier class, 420
  of Table class, 320
INDEX phrase of DO instruction, 50, 51
INDEX phrase of LOOP instruction, 63
INDEX subkeyword
  in a DO instruction, 50, 644
  in a LOOP instruction, 63, 644
indirect evaluation of data, 59
inequality, testing of, 20
infinite loops, 50, 63, 641
information hiding, 4
inherit method
  of Class class, 126
INHERIT subkeyword
  in a CLASS directive, 94
inheritance, 7
init method
  of Alarm class, 330
  of CaselessColumnComparator class, 336
  of CircularQueue class, 268
  of ColumnComparator class, 335
  of DateTime class, 345
  of File class, 364
  of InvertingComparator class, 338
  of Monitor class, 372
  of NumericComparator class, 339
  of Object class, 150
  of RexxQueue class, 411
  of Stream class, 227
  of StreamSupplier class, 418
  of Ticker class, 422
  of TimeSpan class, 426
initialization
  of arrays, 31
  of compound variables, 31
input and output
  functions
    CHARIN, 458
    CHAROUT, 459
    CHARS, 460
    LINEIN, 475
    LINEOUT, 477
    LINES, 478
    STREAM, 487
    model, 619
    streams, 619
  input from the user, 619
  input object, 439
  input streams, 620
  INPUT subkeyword
    in an ADDRESS instruction, 42
  input to PULL from STDIN, 71
  input to PULL from the keyboard, 71
  input, errors during, 626
  InputOutputStream class, 215
  InputStream class, 215
INSERT function
  description, 474
  example, 474
insert method
of Array class, 254
of CircularQueue class, 268
of List class, 286
of MutableBuffer class, 380
of OrderedCollection class, 244
of Queue class, 295
of String class, 198
inserting a string into another, 198, 474
instance methods, 107
instanceMethod method
of Object class, 150
instanceMethods method
of Object class, 150
instances
definition, 5
instructions
ADDRESS, 42
ARG, 46
CALL, 47
definition, 27
DO, 50
DROP, 52
EXIT, 53
EXPOSE, 54
FORWARD, 55
GUARD, 57, 607
IF, 58
INTERPRET, 59
ITERATE, 61
keyword, 28
description, 42
LEAVE, 62
LOOP, 63
message, 28, 36
NOP, 64
NUMERIC, 64
OPTIONS, 66
PARSE, 66
parsing, summary, 577
PROCEDURE, 69
PULL, 71
PUSH, 72
QUEUE, 72
RAISE, 73
REPLY, 75
RETURN, 76
SAY, 76
SELECT, 77
SIGNAL, 79
TRACE, 81
USE, 86
integer
arithmetic, 584
division
description>, 584, 586
integer division operator, 19
interactive debug, 81
internal
functions
return from, 76
variables in, 69
routine, 47
internalDigits method
of RexxInfo class, 404
internalMaxNumber method
of RexxInfo class, 404
internalMinNumber method
of RexxInfo class, 405
INTERPRET instruction
description, 59
example, 60, 60
interpretive execution of data, 59
intersection method
of Bag class, 264
of CircularQueue class, 269
of Collection class, 240
of OrderedCollection class, 245
of Relation class, 301
of Set class, 306
interval method
of Ticker class, 423
InvertingComparator class, 337
invoking
built-in functions, 47
routines, 47
isA method
of Object class, 151
isAbstract method
of Class class, 127
of Method class, 143
isAttribute method
of Method class, 144
isCaseSensitive method
of File class, 360, 365
isConstant method
of Method class, 144
isDirectory method
of File class, 365
isEmpty method
of Array class, 254
of Bag class, 264
of Directory class, 276
of IdentityTable class, 282
of List class, 287
of Queue class, 296
of Relation class, 301
of Set class, 307
of Stem class, 312
of StringTable class, 316
of Table class, 320
isFile method
  of File class, 365
isGuarded method
  of Method class, 144
isHidden method
  of File class, 365
isInstanceOf method
  of Object class, 151
isLeapYear method
  of DateTime class, 356
isMetaclass method
  of Class class, 127
isNil method
  of Object class, 152
isNull method
  of Pointer class, 392
ISO option of DATE function, 466
isoDate method
  of DateTime class, 352
isPackage method
  of Method class, 144
isPosted method
  of EventSemaphore class, 358
isPrivate method
  of Method class, 145
isProtected method
  of Method class, 145
isSubclassOf method
  of Class class, 127
item method
  of StreamSupplier class, 418
  of Supplier class, 420
ITEM phrase of DO instruction, 50, 51
ITEM phrase of LOOP instruction, 63
ITEM subkeyword
  in a DO instruction, 50, 644
  in a LOOP instruction, 63, 644
items method
  of Array class, 255
  of Bag class, 264
  of Collection class, 240
  of Directory class, 276
  of IdentityTable class, 282
  of List class, 287
  of Queue class, 296
  of Relation class, 301
  of Set class, 307
  of Stem class, 312
  of StringTable class, 316
  of Table class, 320
ITERATE instruction
  description, 61
example, 61

J
justification, text right, RIGHT function, 482
justification, text right, RIGHT method, 204

K
keyword
  conflict with commands, 631
  description, 42
  mixed case, 42
  reservation of, 631

L
label
  as target of CALL, 47
  as target of SIGNAL, 79
  description, 27
  duplicate, 81
  in INTERPRET instruction, 59
  search algorithm, 79
LABEL subkeyword
  in a DO instruction, 50, 645
  in a LOOP instruction, 63, 645
language
  processor date and version, 69
  processor execution, 1
  structure and syntax, 8
Language (local) option of DATE function, 466
languageDate method
  of DateTime class, 353
languageLevel method
  of RexxInfo class, 405
last method
  of Array class, 255
  of List class, 287
  of OrderedCollection class, 245
  of Queue class, 296
lastAccessed attribute
  of File class, 366
lastItem method
  of Array class, 255
  of List class, 287
  of OrderedCollection class, 245
  of Queue class, 296
lastModified attribute
  of File class, 366
LASTPOS function
  description, 474
  example, 475
lastPos method
  of MutableBuffer class, 380
  of String class, 199
leading
whitespace removal with STRIP function, 494
whitespace removal with strip method, 206
zeros
adding with RIGHT function, 482
adding with right method, 204
removing with STRIP function, 494
removing with strip method, 206
LEAVE instruction
description, 62
element, 62
leaving your program, 53, 53
LEFT function
description, 475
example, 475
left method
of String class, 199
LENGTH function
description, 475
example, 475
length method
of File class, 367
of MutableBuffer class, 381
of String class, 200
of Validate class, 431
length positional pattern
positional patterns
length, 573
less than operator (<), 20
less than or equal to operator (<=), 20
LIBRARY subkeyword
in a REQUIRES directive, 102
in a RESOURCE directive, 103
libraryPath method
of RexxInfo class, 405
License, Common Public, 699
License, Open Object Rexx, 699
LIFO (last-in, first-out) stacking, 72
line input and output, 619
line method
of RexxContext class, 399
of StackFrame class, 415
LINEIN function
description, 475
example, 476
lineIn method
of InputStream class, 216
of OutputStream class, 217
of RexxQueue class, 412
of Stream class, 227
LINEIN method
role in input and output, 620
LINEIN option of PARSE instruction, 67
LINEIN subkeyword
in a PARSE instruction, 66, 577
LINEOUT function
description, 477
example, 477
lineOut method
of InputStream class, 216
of OutputStream class, 217
of RexxQueue class, 412
of Stream class, 227
LINEOUT method
role in input and output, 621
lines
from a program retrieved with SOURCELINe, 486
from stream, 67
LINES function
description, 478
example, 478
from stream, 475
remaining in stream, 478
lines method
of InputStream class, 216
of OutputStream class, 217
of Stream class, 227
LINES method
role in input and output, 620
List class, 283
list method
of File class, 367
listFiles method
of File class, 367
listRoots method
of File class, 361
literal
description, 11
implementation maximum, 12
patterns, 570
LOAD keyword
in a RXSUBCOM command, 637
load method
of Properties class, 289, 291
loadExternalMethod method
of Method class, 143
loadExternalRoutine method
of Routine class, 167
loadLibrary method
of Package class, 162
loadPackage method
of Package class, 163
local method
of Package class, 163
LOCAL subkeyword
in a USE instruction, 88
locating
  string in a MutableBuffer, 382
  string in another string, 203, 480
  word in another mutable buffer, 386
  word in another string, 211, 506
logical
  operations, 21
logical bit operations
    BITAND, 454
    BITOR, 455
    BITXOR, 455
logical method
  of Validate class, 431
logical NOT character, 15
logical OR operator, 15
longTime method
  of DateTime class, 355
LOOP instruction
  description, 63
  example, 641
loops
  active, 62
  execution model, 647, 648
  modification of, 61
  over collections, 643
  over suppliers, 644
  repetitive, 641
  termination of, 62
LOSTDIGITS subkeyword
  in a CALL instruction, 597
  in a SIGNAL instruction, 79, 593, 597
LOWER function
  description, 479
lower method
  of MutableBuffer class, 381
  of String class, 172, 200
LOWER subkeyword
  in a PARSE instruction, 66, 576
lowercase translation
  with PARSE LOWER, 67
M
majorVersion method
  of RexxInfo class, 406
makeArray method
  of Array class, 255
  of Bag class, 265
  of CircularQueue class, 269
  of Collection class, 240
  of Directory class, 277
  of IdentityTable class, 282
  of List class, 287
  of MapCollection class, 242
  of MutableBuffer class, 381
  of Queue class, 296
  of Relation class, 302
  of RexxQueue class, 412
  of Set class, 307
  of Stem class, 312
  of Stream class, 228
  of String class, 200
  of StringTable class, 316
  of Table class, 320
makeDir method
  of File class, 368
makeDirs method
  of File class, 368
makeString method
  of Array class, 256
  of CircularQueue class, 269
  of DateTime class, 350
  of File class, 369
  of MutableBuffer class, 381
  of StackFrame class, 416
  of String class, 201
  of TimeSpan class, 428
Map Collection classes
  Directory class, 273
  IdentityTable class, 279
  Properties class, 288
  Relation class, 298
  Stem class, 308
  StringTable class, 313
  Table class, 318
MapCollection class, 241
match method
  of MutableBuffer class, 381
  of RegularExpression class, 394
  of String class, 201
matchChar method
  of MutableBuffer class, 382
  of String class, 202
MAX function
  description, 479
  example, 479
max method
  of String class, 202
maxArraySize method
  of RexxInfo class, 406
maxDate method
  of DateTime class, 341
maxExponent method
  of RexxInfo class, 406
maxPathLength method
  of RexxInfo class, 406
Message class, 132
  message instructions, 28, 36
message sequence instructions, 36
MESSAGE subkeyword
in a FORWARD instruction, 55
message-send operator (~), 4
messageComplete method
of Message class, 135
of MessageNotification class, 371
messageName method
of Message class, 136
MessageNotification class, 371
messages, 4
messages to objects
operator as message, 18
~, using, 24
~~, using, 24
messages, error, 651
metaclass
Class class, 121
Singleton class, 413
metaClass method
of Class class, 127
METACLASS subkeyword
in a CLASS directive, 94
metaclasses, 108
method
%, 174, 427
&, 176
&&, 176
*, 174, 427
**, 174
+, 174, 346, 426
-, 174, 346, 427
/, 174, 427
\, 174, 427
<, 175, 390
<=, 175, 391
<>>, 175, 391
<==, 175, 391
<>

of Class class, 122
of Object class, 148
of Pointer class, 390
of String class, 174
>

of Class class, 122
of String class, 175

of Object class, 148
of Orderable class, 390
of Pointer class, 392
of String class, 174

of Class class, 122
of Orderable class, 390
of Pointer class, 392
of String class, 174

of Class class, 122
of Orderable class, 390
of Pointer class, 392
of String class, 174

of Class class, 122
of Orderable class, 390
of Pointer class, 392
of String class, 174

of Class class, 122
of Orderable class, 390
of Pointer class, 392
of String class, 174

addYears method
  of DateTime class, 350
allAt method
  of Bag class, 262
  of Relation class, 299
allIndex method
  of Bag class, 262
  of Relation class, 300
allIndexes method
  of Array class, 250
  of Bag class, 263
  of Collection class, 238
  of Directory class, 275
  of IdentityTable class, 281
  of List class, 284
  of Queue class, 293
  of Relation class, 300
  of Set class, 305
  of Stem class, 311
  of StringTable class, 315
  of Supplier class, 419
  of Table class, 319
allItems method
  of Array class, 250
  of Bag class, 263
  of Collection class, 239
  of Directory class, 275
  of IdentityTable class, 281
  of List class, 284
  of Queue class, 293
  of Relation class, 300
  of Set class, 306
  of Stem class, 311
  of StringTable class, 315
  of Supplier class, 419
  of Table class, 319
alnum method
  of String class, 170
alpha method
  of String class, 171
annotation method
  of Class class, 122
  of Method class, 143
  of Package class, 159
  of Routine class, 168
annotations method
  of Class class, 123
  of Method class, 143
  of Package class, 159
  of Routine class, 168
append method
  of Array class, 250
  of CircularQueue class, 268
  of List class, 284
  of MutableBuffer class, 374
  of OrderedCollection class, 243
  of Queue class, 294
  of String class, 179
appendAll method
  of OrderedCollection class, 243
architecture method
  of RexxInfo class, 401
args method
  of RexxContext class, 398
arguments method
  of Message class, 134
  of StackFrame class, 415
arithmetic methods
  of DateTime class, 346
  of String class, 173
  of TimeSpan class, 426
arrayIn method
  of InputStream class, 215
  of Stream class, 219
arrayOut method
  of OutputStream class, 217
  of Stream class, 219
at method
  of Array class, 251
  of Bag class, 263
  of Collection class, 239
  of Directory class, 275
  of IdentityTable class, 281
  of List class, 285
  of Queue class, 294
  of Relation class, 300
  of Set class, 306
  of Stem class, 311
  of StringTable class, 315
  of Table class, 319
attachment method
  of Alarm class, 329
  of Ticker class, 421
available method
  of StreamSupplier class, 418
  of Supplier class, 420
b2x method
  of String class, 179
baseClass method
  of Class class, 123
baseDate method
  of DateTime class, 352
bitAnd method
  of String class, 179
bitOr method
  of String class, 180
bitXor method
  of String class, 180
blank method of String class, 171
c2d method of String class, 181
c2x method of String class, 181
call method of Routine class, 168
callWith method of Routine class, 168
cancel method of Alarm class, 329
 of AlarmNotification class, 331
 of Ticker class, 421
canceled/cancelled method of Alarm class, 329
 of Ticker class, 422
canRead method of File class, 363
canWrite method of File class, 363
caselessAbbrev method of String class, 182
caselessChangeStr method of MutableBuffer class, 374
 of String class, 183
caselessCompareTo method of String class, 183, 183
caselessContains method of MutableBuffer class, 375
 of String class, 184
caselessContainsWord method of MutableBuffer class, 375
 of String class, 184
caselessCountStr method of MutableBuffer class, 376, 376
 of String class, 185, 185
caselessEndsWith method of MutableBuffer class, 376
 of String class, 185
caselessEquals method of String class, 185, 185
caselessLastPos method of MutableBuffer class, 376, 376
 of String class, 186, 186
caselessMatch method of MutableBuffer class, 376, 376
 of String class, 186, 186
caselessMatchChar method of MutableBuffer class, 377, 377
 of String class, 187, 187
caselessPos method of MutableBuffer class, 377, 377
 of String class, 187, 187
caselessStartsWith method of MutableBuffer class, 377
 of String class, 187
caseSensitiveFiles method of RexxInfo class, 401
ceiling method of String class, 188
center method of String class, 189
centre method of String class, 189
changeStr method of MutableBuffer class, 378
 of String class, 189
charIn method of InputStream class, 216
 of OutputStream class, 217
 of Stream class, 219
charOut method of InputStream class, 216
 of OutputStream class, 217
 of Stream class, 220
chars method of InputStream class, 216
 of OutputStream class, 217
 of Stream class, 220
civilTime method of DateTime class, 354
class method of Object class, 148
classes method of Package class, 159
classType method of Validate class, 431
close method of InputStream class, 216
 of OutputStream class, 217
 of Stream class, 220
cntrl method of String class, 171
command method of Stream class, 220
compare method of CaselessColumnComparator class, 336
 of CaselessComparator class, 334
 of CaselessDescendingComparator class, 337
 of ColumnComparator class, 335
 of Comparator class, 333
 of DescendingComparator class, 336
 of InvertingComparator class, 338
 of NumericComparator class, 339
 of String class, 183, 190
compareTo method
of Comparable class, 333
of DateTime class, 347
of File class, 363
of Orderable class, 390
of String class, 190
of TimeSpan class, 427
comparison methods
  of Class class, 122
  of Object class, 147
  of Orderable class, 390
  of String class, 174
completed method
  of Message class, 134
concatenation methods
  of Object class, 148
  of String class, 176
condition method
  of RexxContext class, 398
contains method
  of MutableBuffer class, 378
  of String class, 191
containsWord method
  of MutableBuffer class, 378
  of String class, 191
copies method
  of String class, 192
copy method
  of Object class, 148
countStr method
  of MutableBuffer class, 379
  of String class, 192
cr method
  of String class, 171
create method
  of RexxQueue class, 410
creation, 97
current method
  of Monitor class, 372
d2c method
  of String class, 192
d2x method
  of String class, 193
dataType method
  of String class, 193
date method
  of DateTime class, 356
  of RexxInfo class, 402
day method
  of DateTime class, 349
dayMicroseconds method
  of DateTime class, 350
dayMinutes method
  of DateTime class, 350
dayName method
  of DateTime class, 353
days method
  of TimeSpan class, 427
daySeconds method
  of DateTime class, 350
daysInMonth method
  of DateTime class, 357
daysInYear method
  of DateTime class, 357
debug method
  of RexxInfo class, 402
decodeBase64 method
  of String class, 195
defaultName method
  of Class class, 123
  of Object class, 149
define method
  of Class class, 123
definedMethods method
  of Package class, 160
defineMethods method
  of Class class, 124
definition, 4
delete class method
  of RexxQueue class, 410
delete method
  of Array class, 251
  of Class class, 125
  of File class, 363
  of List class, 285
  of MutableBuffer class, 379
  of OrderedCollection class, 244
  of Queue class, 294
  of RexxQueue class, 411
delStr method
  of MutableBuffer class, 379
  of String class, 195
delWord method
  of MutableBuffer class, 379
  of String class, 196
description method
  of Stream class, 226
destination method
  of Monitor class, 372
difference method
  of Bag class, 263
  of Collection class, 239
  of OrderedCollection class, 244
  of Relation class, 300
digit method
  of String class, 171
digits method
  of Package class, 160
  of RexxContext class, 398
of TimeSpan class, 425
fromNormalDate method
  of DateTime class, 341
fromNormalTime method
  of DateTime class, 344
  of TimeSpan class, 425
fromOrderedDate method
  of DateTime class, 342
fromOrdinalDate method
  of DateTime class, 342
fromSeconds method
  of TimeSpan class, 425
fromStandardDate method
  of DateTime class, 343
fromStringFormat method
  of TimeSpan class, 426
fromTicks method
  of DateTime class, 344
fromUsaDate method
  of DateTime class, 343
fromUtcIsoDate method
  of DateTime class, 345
fromWeekNumberDate method
  of DateTime class, 343
fullDate method
  of DateTime class, 355
fuzz method
  of Package class, 162
  of RexxContext class, 399
  of RexxInfo class, 404
get method
  of RexxQueue class, 411
getBufferSize method
  of MutableBuffer class, 380
getLogical method
  of Properties class, 290
getProperty method
  of Properties class, 290
getWhole method
  of Properties class, 290
graph method
  of String class, 171
halt method
  of Message class, 134
hasEntry method
  of Directory class, 276
  of StringTable class, 315
hasError method
  of Message class, 135
hashCode method
  of DateTime class, 350
  of File class, 364
  of Object class, 149
  of String class, 198
  of TimeSpan class, 429
hasIndex method
  of Array class, 253
  of Bag class, 264
  of Collection class, 239
  of Directory class, 276
  of IdentityTable class, 281
  of List class, 285
  of Queue class, 295
  of Relation class, 301
  of Set class, 306
  of Stem class, 311
  of StringTable class, 315
  of Table class, 319
hasItem method
  of Array class, 253
  of Bag class, 264
  of Collection class, 240
  of Directory class, 276
  of IdentityTable class, 281
  of List class, 286
  of Queue class, 295
  of Relation class, 301
  of Set class, 306
  of Stem class, 311
  of StringTable class, 316
  of Table class, 320
hasMethod method
  of Object class, 149
hasResult method
  of Message class, 135
hours method
  of DateTime class, 349
  of TimeSpan class, 428
id method
  of Class class, 126
identityHash method
  of Object class, 150
importedClasses method
  of Package class, 162
importedPackages method
  of Package class, 162
importedRoutines method
  of Package class, 162
index method
  of Array class, 254
  of Bag class, 264
  of Collection class, 240
  of Directory class, 276
  of IdentityTable class, 281
  of List class, 286
  of Queue class, 295
  of Relation class, 301
  of Set class, 306
of Stem class, 311
of StreamSupplier class, 418
of Supplier class, 420
of Table class, 320

inherit method
of Class class, 126

init method
of Alarm class, 330
of CaselessColumnComparator class, 336
of CircularQueue class, 268
of ColumnComparator class, 335
of DateTime class, 345
of File class, 364
of InvertingComparator class, 338
of Monitor class, 372
of NumericComparator class, 339
of Object class, 150
of RexxQueue class, 411
of Stream class, 227
of StreamSupplier class, 418
of Ticker class, 422
of TimeSpan class, 426

insert method
of Array class, 254
of CircularQueue class, 268
of List class, 286
of MutableBuffer class, 380
of OrderedCollection class, 244
of Queue class, 295
of String class, 198

instanceMethod method
of Object class, 150

instanceMethods method
of Object class, 150

internalDigits method
of RexxInfo class, 404

internalMaxNumber method
of RexxInfo class, 404

internalMinNumber method
of RexxInfo class, 405

intersection method
of Bag class, 264
of CircularQueue class, 269
of Collection class, 240
of OrderedCollection class, 245
of Relation class, 301
of Set class, 306

interval method
of Ticker class, 423

isA method
of Object class, 151

isAbstract method
of Class class, 127

of Method class, 143

isAttribute method
of Method class, 144

isCaseSensitive method
of File class, 360, 365

isConstant method
of Method class, 144

isDirectory method
of File class, 365

isEmpty method
of Array class, 254
of Bag class, 264
of Directory class, 276
of IdentityTable class, 282
of List class, 287
of Queue class, 296
of Relation class, 301
of Set class, 307
of Stem class, 312
of StringTable class, 316
of Table class, 320

isFile method
of File class, 365

isGuarded method
of Method class, 144

isHidden method
of File class, 365

isInstanceOf method
of Object class, 151

isLeapYear method
of DateTime class, 356

isMetaClass method
of Class class, 127

isNil method
of Object class, 152

isNull method
of Pointer class, 392

isoDate method
of DateTime class, 352

isPackage method
of Method class, 144

isPosted method
of EventSemaphore class, 358

isPrivate method
of Method class, 145

isProtected method
of Method class, 145

isSubclassOf method
of Class class, 127

item method
of StreamSupplier class, 418
of Supplier class, 420

items method
of Array class, 255
of Bag class, 264
of Collection class, 240
of Directory class, 276
of IdentityTable class, 282
of List class, 287
of Queue class, 296
of Relation class, 301
of Set class, 307
of Stem class, 312
of StringTable class, 316
of Table class, 320
languageDate method
of DateTime class, 353
languageLevel method
of RexxInfo class, 405
last method
of Array class, 255
of List class, 287
of OrderedCollection class, 245
of Queue class, 296
lastAccessed attribute
of File class, 366
lastItem method
of Array class, 255
of List class, 287
of OrderedCollection class, 245
of Queue class, 296
lastModified attribute
of File class, 366
lastPos method
of MutableBuffer class, 380
of String class, 199
left method
of String class, 199
length method
of File class, 367
of MutableBuffer class, 381
of String class, 200
of Validate class, 431
libraryPath method
of RexxInfo class, 405
line method
of RexxContext class, 399
of StackFrame class, 415
lineIn method
of InputStream class, 216
of OutputStream class, 217
of Stream class, 227
list method
of File class, 367
listFiles method
of File class, 367
listRoots method
of File class, 361
load method
of Properties class, 289, 291
loadExternalMethod method
of Method class, 143
loadExternalRoutine method
of Routine class, 167
loadLibrary method
of Package class, 162
loadPackage method
of Package class, 163
local method
of Package class, 163
logical method
of Validate class, 431
logical methods
of String class, 176
longTime method
of DateTime class, 355
lower method
of MutableBuffer class, 381
of String class, 172, 200
majorVersion method
of RexxInfo class, 406
makeArray method
of Array class, 255
of Bag class, 265
of CircularQueue class, 269
of Collection class, 240
of Directory class, 277
of IdentityTable class, 282
of List class, 287
of MapCollection class, 242
of Package class, 245
of RexxQueue class, 412
of Queue class, 296
of Relation class, 302
of RexxQueue class, 412
of Set class, 307
of Stem class, 312
of Stream class, 228
of String class, 200
of StringTable class, 316
of Table class, 320
makeDir method
of File class, 368
makeDirs method
  of File class, 368
makeString method
  of Array class, 256
  of CircularQueue class, 269
  of DateTime class, 350
  of File class, 369
  of MutableBuffer class, 381
  of StackFrame class, 416
  of String class, 201
  of TimeSpan class, 428
match method
  of MutableBuffer class, 381
  of RegularExpression class, 394
  of String class, 201
matchChar method
  of MutableBuffer class, 382
  of String class, 202
max method
  of String class, 202
maxArraySize method
  of RexxInfo class, 406
maxDate method
  of DateTime class, 341
maxExponent method
  of RexxInfo class, 406
maxPathLength method
  of RexxInfo class, 406
messageComplete method
  of Message class, 135
  of MessageNotification class, 371
messageName method
  of Message class, 136
metaClass method
  of Class class, 127
method method
  of Class class, 128
methods method
  of Class class, 128
microseconds method
  of DateTime class, 350
  of TimeSpan class, 428
min method
  of String class, 202
minDate method
  of DateTime class, 340
minExponent method
  of RexxInfo class, 407
minutes method
  of DateTime class, 349
  of TimeSpan class, 428
mixinClass method
  of Class class, 129
modification method
  of RexxInfo class, 407
modulo method
  of String class, 203
month method
  of DateTime class, 349
monthName method
  of DateTime class, 353
name method
  of File class, 369
  of Package class, 163
  of RexxContext class, 399
  of RexxInfo class, 407
  of StackFrame class, 416
  of VariableReference class, 434
namespaces method
  of Package class, 163
new method
  of Array class, 249
  of Bag class, 262
  of Buffer class, 332
  of Class class, 129
  of Directory class, 274
  of EventSemaphore class, 357
  of IdentityTable class, 280
  of List class, 283
  of Message class, 133
  of Method class, 142
  of MutableBuffer class, 373
  of MutexSemaphore class, 388
  of Object class, 147
  of Package class, 158
  of Pointer class, 391
  of Properties class, 290
  of Queue class, 293
  of RegularExpression class, 394
  of Relation class, 299
  of RexxQueue class, 410
  of Routine class, 167
  of Set class, 305
  of Singleton class, 414
  of Stem class, 310
  of Stream class, 219
  of String class, 170
  of StringTable class, 314
  of Supplier class, 419
  of Table class, 318
  of WeakReference class, 436
newFile method
  of Method class, 142
  of Routine class, 167
next method
  of Array class, 256
  of List class, 287
  of OrderedCollection class, 245
of Queue class, 296
of StreamSupplier class, 418
of Supplier class, 420
nl method
  of String class, 172
nonNegativeNumber method
  of Validate class, 432
nonNegativeWholeNumber method
  of Validate class, 432
normalDate method
  of DateTime class, 354
normalTime method
  of DateTime class, 355
notify method
  of Message class, 136
null method
  of String class, 172
number method
  of Validate class, 432
numberRange method
  of Validate class, 432
objectName method
  of Object class, 152
objectName= method
  of Object class, 152
of method
  of Array class, 249
  of Bag class, 262
  of CircularQueue class, 268
  of List class, 284
  of MapCollection class, 242
  of Queue class, 293
  of Set class, 305
offset method
  of DateTime class, 356
open method
  of InputStream class, 216
  of OutputStream class, 218
  of RexxQueue class, 411
  of Stream class, 228
orderedDate method
  of DateTime class, 354
ordinalDate method
  of DateTime class, 349
overlay method
  of MutableBuffer class, 382
  of String class, 203
package method
  of Class class, 130
  of Method class, 145
  of RexxContext class, 399
  of RexxInfo class, 408
  of Routine class, 169
package-scope, 115
parent method
  of File class, 369
parentFile method
  of File class, 369
parse method
  of RegularExpression class, 395
path method
  of File class, 370
pathSeparator method
  of File class, 361, 370
  of RexxInfo class, 408
peek method
  of Queue class, 296
platform method
  of RexxInfo class, 408
pos method
  of MutableBuffer class, 382
  of RegularExpression class, 397
  of String class, 203
position method
  of InputStream class, 216
  of OutputStream class, 218
  of RegularExpression class, 397
  of Stream class, 230
  of Validate class, 432
positiveNumber method
  of Validate class, 433
positiveWholeNumber method
  of Validate class, 433
post method
  of EventSemaphore class, 358
prefix +, 174, 346, 427
prefix -, 174, 346, 427
previous method
  of Array class, 256
  of List class, 287
  of OrderedCollection class, 245
  of Queue class, 297
print method
  of String class, 172
private, 115
prolog method
  of Package class, 164
public, 115
publicClasses method
  of Package class, 164
publicRoutines method
  of Package class, 164
pull method
  of Queue class, 297
  of RexxQueue class, 412
punct method
  of String class, 172
push method
of CircularQueue class, 270
of Queue class, 297
of RexxQueue class, 412
put method
of Array class, 257
of Bag class, 265
of Collection class, 240
of Directory class, 277
of IdentityTable class, 282
of List class, 288
of Properties class, 291
of Queue class, 297
of Relation class, 302
of Set class, 307
of Stem class, 312
of StringTable class, 316
of Table class, 320
putAll method
of Bag class, 265
of MapCollection class, 242
of Set class, 307
qualify method
of Stream class, 230
query
of Stream class, 230
queryMixinClass method
of Class class, 130
queue method
of CircularQueue class, 270
of Queue class, 297
of RexxQueue class, 413
queued method
of RexxQueue class, 413
release method
of MutexSemaphore class, 389
of RexxInfo class, 408
remove method
of Array class, 257
of Bag class, 265
of Directory class, 277
of IdentityTable class, 282
of List class, 288
of Queue class, 297
of Relation class, 302
of Set class, 307
of Stem class, 312
of StringTable class, 316
of Table class, 320
removeAll method
of Bag class, 265
of Relation class, 302
removeEntry method
of Directory class, 277
of StringTable class, 317
removeItem method
of Array class, 258
of Bag class, 266
of Directory class, 277
of IdentityTable class, 282
of List class, 288
of Queue class, 297
of Relation class, 302
of Set class, 308
of Stem class, 312
of StringTable class, 317
of Table class, 321
renameTo method
of File class, 370
replaceAt method
of MutableBuffer class, 383
of String class, 204
reply method
of Message class, 137
replyWith method
of Message class, 137
request method
of Object class, 152
of Stem class, 312
of VariableReference class, 435
requestClassType method
of Validate class, 433
reset method
of EventSemaphore class, 358
resize method
of CircularQueue class, 270
resource method
of Package class, 165
resources method
of Package class, 165
result method
of Message class, 138
reverse method
of String class, 204
revision method
of RexxInfo class, 409
right method
of String class, 204
round method
of String class, 205
routines method
of Package class, 165
rs method
of RexxContext class, 399
run method
of Object class, 153
save method
of Properties class, 291
say method
of RexxQueue class, 413
of Stream class, 233
scheduledTime method
  of Alarm class, 331
scope, 113
scope method
  of Method class, 145
search order
  changing, 113
searchPath method
  of File class, 361
seconds method
  of DateTime class, 350
  of TimeSpan class, 428
section method
  of Array class, 258
  of CircularQueue class, 271
  of List class, 288
  of OrderedCollection class, 245
  of Queue class, 298
seek method
  of Stream class, 233
selection
  search order, 113
send method
  of Message class, 138
  of Object class, 153
sendWith method
  of Message class, 139
  of Object class, 154
separator method
  of File class, 361, 370
set method
  of RexxQueue class, 413
setBufferSize method
  of MutableBuffer class, 383
setEntry method
  of Directory class, 277
  of StringTable class, 317
setGuarded method
  of Method class, 146
setLogical method, 291
setMethod method
  of Directory class, 278
  of Object class, 155
setPrivate method
  of Method class, 146
setProperty method
  of Properties class, 291
setProtected method
  of Method class, 146
setReadOnly method
  of File class, 370
setSecurityManager method
  of Method class, 146
  of Package class, 166
  of Routine class, 169
setText method
  of MutableBuffer class, 383
setUnguarded method
  of Method class, 146
setWhole method
  of Properties class, 291
setWritable method
  of File class, 370
sign method
  of String class, 205
  of TimeSpan class, 430
size method
  of Array class, 258
  of CircularQueue class, 271
  of Queue class, 298
sort method
  of Array class, 259
  of OrderedCollection class, 245
sortWith method
  of Array class, 259
  of OrderedCollection class, 246
source method
  of Method class, 146
  of Package class, 166
  of Routine class, 169
sourceLine method
  of Package class, 166
sourceSize method
  of Package class, 166
space method
  of MutableBuffer class, 383
  of String class, 173, 206
stableSort method
  of Array class, 259
  of OrderedCollection class, 246
stableSortWith method, 259
  of OrderedCollection class, 246
stackFrames method
  of RexxContext class, 399
standardDate method
  of DateTime class, 354
start method
  of Message class, 139
  of Object class, 155
startsWith method
  of Message class, 140
  of Object class, 156
state method
of Stream class, 235

string method
  of CircularQueue class, 271
  of DateTime class, 357
  of File class, 371
  of Object class, 156
  of StackFrame class, 416
  of Stream class, 235
  of TimeSpan class, 430

strip method
  of String class, 206

subchar method
  of String, 207

subChar method
  of MutableBuffer class, 384

subclass method
  of Class class, 130

subclasses method
  of Class class, 131

subset method
  of Bag class, 266
  of Collection class, 241
  of OrderedCollection class, 246
  of Relation class, 302
  of Set class, 308

substr method
  of MutableBuffer class, 384
  of String class, 207

subWord method
  of MutableBuffer class, 384
  of String class, 207

subWords method
  of MutableBuffer class, 384
  of String class, 208

superClass method
  of Class class, 131

superClasses method
  of Class class, 131

supplier method
  of Array class, 259
  of Bag class, 266
  of CircularQueue class, 272
  of Collection class, 241
  of Directory class, 278
  of IdentityTable class, 282
  of List class, 288
  of Queue class, 298
  of Relation class, 303
  of Set class, 308
  of Stem class, 313
  of Stream class, 235
  of StringTable class, 317
  of Supplier class, 421

of Table class, 321

tab method
  of String class, 173

target method
  of Message class, 141
  of StackFrame class, 416

temporaryPath method
  of File class, 362

ticks method
  of DateTime class, 356

timeOfDay method
  of DateTime class, 356

today method
  of DateTime class, 341

toDirectory method
  of Stem class, 313

toLocalTime method
  of DateTime class, 355

toString method
  of Array class, 259

totalDays method
  of TimeSpan class, 428

totalHours method
  of TimeSpan class, 428

totalMicroseconds method
  of TimeSpan class, 429

totalMinutes method
  of TimeSpan class, 429

totalSeconds method
  of TimeSpan class, 429

toTimezone method
  of DateTime class, 355

toUtcTime method
  of DateTime class, 355

trace method
  of Package class, 166

traceLine method
  of StackFrame class, 416

translate method
  of MutableBuffer class, 385
  of String class, 208

triggered method
  of Alarm class, 331
  of AlarmNotification class, 332
  of Message class, 141

trunc method
  of String class, 209

type method
  of StackFrame class, 416

uninherit method
  of Class class, 132

uninit method
  of EventSemaphore class, 358
  of MutexSemaphore class, 389
of Stream class, 236
union method
  of Bag class, 266
  of CircularQueue class, 272
  of Collection class, 241
  of OrderedCollection class, 246
  of Relation class, 303
  of Set class, 308
uniqueIndexes method
  of Bag class, 266
  of Relation class, 303
unknown method
  of Directory class, 278
  of Monitor class, 372
  of Stem class, 313
  of StringTable class, 317
  of VariableReference class, 435
unsetMethod method
  of Directory class, 278
  of Object class, 157
upper method
  of MutableBuffer class, 385
  of String class, 173, 210
usaDate method
  of DateTime class, 354
utcDate method
  of DateTime class, 355
utcIsoDate method
  of DateTime class, 352
value method
  of VariableReference class, 435
  of WeakReference class, 436
variables method
  of RexxContext class, 399
verify method
  of MutableBuffer class, 385
  of String class, 210
version method
  of RexxInfo class, 409
wait method
  of Message class, 142
Wait method
  of EventSemaphore class, 359
weekDay method
  of DateTime class, 353
weekNumber method
  of DateTime class, 347
weekNumberOfDate method
  of DateTime class, 348
weekNumberOfYear method
  of DateTime class, 348
weeksInYear method
  of DateTime class, 348
of Validate class, 433
wholeNumberRange method
  of Validate class, 434
word method
  of MutableBuffer class, 386
  of String class, 211
wordIndex method
  of MutableBuffer class, 386
  of String class, 211
wordLength method
  of MutableBuffer class, 387
  of String class, 212
wordPos method
  of MutableBuffer class, 377, 387
  of String class, 188, 212
words method
  of MutableBuffer class, 387
  of String class, 212
x2b method
  of String class, 213
x2c method
  of String class, 213
x2d method
  of String class, 214
xDigit method
  of String class, 173
xor method
  of Bag class, 266
  of CircularQueue class, 272
  of Collection class, 241
  of OrderedCollection class, 247
  of Relation class, 303
  of Set class, 308
year method
  of DateTime class, 349
yearDay method
  of DateTime class, 352
[] method
  of Array class, 249
  of Bag class, 262
  of Collection class, 238
  of Directory class, 275
  of IdentityTable class, 280
  of List class, 284
  of MutableBuffer class, 373
  of Queue class, 293
  of Relation class, 299
  of Routine class, 168
  of Set class, 305
  of Stem class, 310
  of String class, 177
  of StringTable class, 314
  of Table class, 319
[]= method
of Array class, 250
of Bag class, 263
of Collection class, 238
of Directory class, 275
of IdentityTable class, 280
of List class, 284
of MutableBuffer class, 374
of Properties class, 290
of Queue class, 293
of Relation class, 299
of Set class, 305
of Stem class, 310
of StringTable class, 314
of Table class, 319
\, 176
\<, 175, 391
\<<, 175, 391
\= of Class class, 122
of Object class, 148
of Orderable class, 390
of Pointer class, 392
of String class, 174
\==
of Class class, 122
of Object class, 148
of Orderable class, 391
of Pointer class, 392
of String class, 175
\>, 175, 391
\>>, 176, 391
|, 176
||, 148, 177
Method class, 142
method method
of Class class, 128
METHOD subkeyword
in an ANNOTATE directive, 90
methods method
of Class class, 128
microseconds method
of DateTime class, 350
of TimeSpan class, 428
MIN function
description, 480
example, 480
min method
of String class, 202
minDate method
of DateTime class, 340
minExponent method
of RexxInfo class, 407
minutes calculated from midnight, 498
minutes method
of DateTime class, 349
of TimeSpan class, 428
mixin classes, 108
mixinClass method
of Class class, 129
MIXINCLASS subkeyword
in a CLASS directive, 94
model of input and output, 619
modification method
of RexxInfo class, 407
modularizing data, 1
modulo method
of String class, 203
monitor, 612
Monitor class, 371
month method
of DateTime class, 349
Month option of DATE function, 467
monthName method
of DateTime class, 353
multiple inheritance, 7
multiplication operator, 19
MutableBuffer class, 373
MutexSemaphore class, 387

N
name method
of File class, 369
of Package class, 163
of RexxContext class, 399
of RexxInfo class, 407
of StackFrame class, 416
of VariableReference class, 434
NAME subkeyword
in a CALL instruction, 47
in a SIGNAL instruction, 79
name, definition, 42
names
of functions, 443
of programs, 68
of subroutines, 47
of variables, 13
namespace, 35
namespaces method
of Package class, 163
negation
of logical values, 21
new method
of Array class, 249
of Bag class, 262
of Buffer class, 332
of Class class, 129
of Directory class, 274
of EventSemaphore class, 357
of IdentityTable class, 280
of List class, 283
of Message class, 133
of Method class, 142
of MutableBuffer class, 373
of MutexSemaphore class, 388
of Object class, 147
of Package class, 158
of Pointer class, 391
of Properties class, 290
of Queue class, 293
of RegularExpression class, 394
of Relation class, 299
of RexxQueue class, 410
of Routine class, 167
of Set class, 305
of Singleton class, 414
of Stem class, 310
of Stream class, 219
of String class, 170
of StringTable class, 314
of Supplier class, 419
of Table class, 318
of WeakReference class, 436
newFile method
of Method class, 142
of Routine class, 167
next method
of Array class, 256
of List class, 287
of OrderedCollection class, 245
of Queue class, 296
of StreamSupplier class, 418
of Supplier class, 420
nibbles, 12
nl method
of String class, 172
NOMETHOD subkeyword
in a SIGNAL instruction, 79, 593, 597
nonNegativeNumber method
of Validate class, 432
nonNegativeWholeNumber method
of Validate class, 432
NOP instruction
description, 64
example, 64
Normal option of DATE function, 467
NORMAL subkeyword
in an ADDRESS instruction, 42
normalDate method
of DateTime class, 354
normalTime method
of DateTime class, 355
NOSTRING subkeyword
in a SIGNAL instruction, 79, 593, 597
not equal operator, 20
not greater than operator, 20
not less than operator, 20
NOT operator, 15, 21
notation
engineering, 587
exponential, example, 587
scientific, 587
Notices, 697
notify method
of Message class, 136
NOTREADY condition
condition trapping, 626
raised by stream errors, 626
NOTREADY subkeyword
in a CALL instruction, 47, 597
in a SIGNAL instruction, 79, 593, 597
NOVALUE condition
not raised by VALUE function, 503
use of, 631
NOVALUE subkeyword
in a SIGNAL instruction, 79, 594, 597
null
clauses, 26
strings, 11
null method
of String class, 172
number method
of Validate class, 432
numberRange method
of Validate class, 432
numbers
arithmetic on, 19, 584, 585
checking with dataType, 193
checking with DATATYPE, 464
comparison of, 20, 588
description, 14, 584
formatting for display, 197, 473
implementation maximum, 14
in DO instruction, 51
truncating, 209, 502
use in the language, 589
numbers for display, 197, 473
numeric
comparisons, example, 589
options in TRACE, 84
NUMERIC instruction
description, 64, 64
DIGITS option, 65
FORM option, 65, 588
FUZZ option, 65
settings saved during subroutine calls, 49
NumericComparator class, 338
O
object, 17
  as data value, 18
definition, 3
  kinds of, 3
Object class, 147
object classes, 7, 107
object method, 107
object variable pool, 54, 603
object-based concurrency, 601
object-oriented programming, 1
objectName method
  of Object class, 152
objectName= method
  of Object class, 152
of method
  of Array class, 249
  of Bag class, 262
  of CircularQueue class, 268
  of List class, 284
  of MapCollection class, 242
  of Queue class, 293
  of Set class, 305
OFF subkeyword
  in a CALL instruction, 47
  in a GUARD instruction, 57
  in a SIGNAL instruction, 79
offset method
  of DateTime class, 356
ON subkeyword
  in a CALL instruction, 47
  in a GUARD instruction, 57
  in a SIGNAL instruction, 79
ooRexx License, 699
open method
  of InputStream class, 216
  of OutputStream class, 218
  of RexxQueue class, 411
  of Stream class, 228
Open Object Rexx License, 699
operations
  tracing results, 81
operator
  arithmetic
    description, 18, 584, 585
    list, 19
  as message, 18
  as special characters, 15
  characters, 15
  comparison, 19, 588
  concatenation, 18
  examples, 587
  logical, 21
  precedence (priorities) of, 21
options
  alphabetical character word options, 82
  numeric in TRACE, 84
OPTIONS instruction
  description, 66
OR, logical, 21
Orderable class, 390
Ordered collection classes
  Array class, 247
  CircularQueue class, 267
  List class, 283
  Queue class, 292
Ordered option of DATE function, 467
OrderedCollection class, 243
orderedDate method
  of DateTime class, 354
ordinalDate method
  of DateTime class, 349
ORing character together, 180, 455
OTHERWISE
  as free standing clause, 42
  OTHERWISE subkeyword
    in a SELECT instruction, 77
output
  errors during, 626
  object, 439
to the user, 619
OUTPUT subkeyword
  in an ADDRESS instruction, 42
OutputStream class, 216
OVER phrase of DO instruction, 50
OVER phrase of LOOP instruction, 63
OVER subkeyword
  in a DO instruction, 50, 643, 644
  in a LOOP instruction, 63, 643, 644
overflow, arithmetic, 590
OVERLAY function
  description, 480
  example, 480
overlay method
  of MutableBuffer class, 382
  of String class, 203
overlaying a string onto another, 203, 480
overview of parsing, 581
P
Package class, 157
package method
  of Class class, 130
  of Method class, 145
  of RexxContext class, 399
  of RexxInfo class, 408
  of Routine class, 169
PACKAGE subkeyword
in a METHOD directive, 97
in an ANNOTATE directive, 90
in an ATTRIBUTE directive, 91
package-scope method, 115
packing a string with X2C, 213, 509
pad character, definition, 450
page, code, 9
parent method
of File class, 369
parentFile method
of File class, 369
parentheses
adjacent to whitespace, 15
in expressions, 21
in function calls, 443
in parsing templates, 576
PARSE instruction
description, 66
example, 68
PARSE LINEIN method
role in input and output, 620
parse method
of RegularExpression class, 395
PARSE PULL method
role in input and output, 620
parsing, 572, 573
advanced topics, 579
combining patterns and parsing into words
string, 574
combining string and positional patterns, 579
conceptual overview, 580
description, 567, 581
equal sign, 572
examples
combining positional patterns with parsing into words, 575
combining string and positional patterns, 580
combining string pattern and parsing into words, 574
parsing instructions, 577
parsing multiple strings in a subroutine, 579
period as a placeholder, 569
simple template, 567
templates containing positional patterns, 571
templates containing string patterns, 570
using a variable as a string pattern, 576
using an expression as a positional pattern, 576
into words, 567
patterns
positional, 567, 571
string, 567, 569
word parsing, conceptual overview, 583
period as placeholder, 569
positional patterns, 567
absolute, 571
variable, 576
selecting words, 567
several assignments, 576
several strings, 579
source string, 567
special case, 579
steps, 581
string patterns, 567
literal string patterns, 569
variable string patterns, 576
summary of instructions, 577
templates
in ARG instruction, 46
in PARSE instruction, 67
in PULL instruction, 71
treatment of blanks, 568
treatment of whitespace, 568
UPPER, use of, 577
variable patterns
string, 576
word parsing
conceptual overview, 583
description and examples, 567
path method
of File class, 370
pathSeparator method
of File class, 361, 370
of RexxInfo class, 408
patterns in parsing
combined with parsing into words, 574
conceptual overview, 581, 582, 583
positional, 567, 571
string, 567, 569
peek method
of Queue class, 296
period
as placeholder in parsing, 569
causing substitution in variable names, 33
in numbers, 584
permanent command destination change, 42
persistent input and output, 619
platform method
of RexxInfo class, 408
Pointer class, 391
polymorphism, 7
POS function
description, 480
example, 481
pos method
of MutableBuffer class, 382
of RegularExpression class, 397
of String class, 203
position
last occurrence of a string, 199, 380, 474
position method
of InputStream class, 216
of OutputStream class, 218
of RegularExpression class, 397
of Stream class, 230
of Validate class, 432
positional patterns
absolute, 569
description, 567
length, 573
relative, 572
variable, 576
positiveNumber method
of Validate class, 433
positiveWholeNumber method
of Validate class, 433
post method
of EventSemaphore class, 358
power operator, 19
powers of ten in numbers, 14
precedence of operators, 21
prefix + method, 177
prefix + operator, 19
prefix - method, 177
prefix - operator, 19
prefix \ operator, 20, 21
presumed command destinations, 42
previous method
of Array class, 256
of List class, 287
of OrderedCollection class, 245
of Queue class, 297
print method
of String class, 172
private method, 115
PRIVATE subkeyword
in a CLASS directive, 94
in a METHOD directive, 97
in a ROUTINE directive, 104
in an ATTRIBUTE directive, 91
PROCEDURE instruction
description, 69
element, 69
programming restrictions, 1
programs
arguments to, 46
examples, 626
retrieving lines with SOURCELINE, 486
retrieving name of, 68
programs without source, 639
prolog method
of Package class, 164
PROPAGATE subkeyword
in a RAISE instruction, 73
Properties class, 288
PROTECTED subkeyword
in a METHOD directive, 97
in an ATTRIBUTE directive, 91
protecting variables, 69
pseudo random number RANDOM function, 481
public method, 115
public object

CONTEXT object, 440
.DEBUGINPUT object, 439
.ENDOFLINE object, 437
.ERROR object, 439
.FALSE object, 437
.INPUT object, 439
.LINE object, 440
.LOCAL object, 438
.METHODS object, 441
.nil object, 437
.OUTPUT object, 439
.RESOURCES object, 441
.RexxInfo object, 437
.ROUTINES object, 441
.RS object, 442
.STDERR object, 439
.STDIN object, 439
.STDOUT object, 440
.STDQUE object, 440
.SYSCARGS object, 440
.TRACEOUTPUT object, 439
.TRUE object, 438
PUBLIC subkeyword
in a CLASS directive, 94
in a METHOD directive, 97
in a ROUTINE directive, 104
in an ATTRIBUTE directive, 91
publicClasses method
of Package class, 164
publicRoutines method
of Package class, 164
PULL instruction
description, 71
element, 72
pull method
of Queue class, 297
of RexxQueue class, 412
PULL method
role in input and output, 620
PULL option of PARSE instruction, 67
PULL subkeyword
in a PARSE instruction, 66, 71, 577
punct method
    of String class, 172
PUSH instruction
description, 72
    example, 72
push method
    of CircularQueue class, 270
    of Queue class, 297
    of RexxQueue class, 412
put method
    of Array class, 257
    of Bag class, 265
    of Collection class, 240
    of Directory class, 277
    of IdentityTable class, 282
    of List class, 288
    of Properties class, 291
    of Queue class, 297
    of Relation class, 302
    of Set class, 307
    of Stem class, 312
    of StringTable class, 316
    of Table class, 320
putAll method
    of Bag class, 265
    of MapCollection class, 242
    of Set class, 307, 308
R
QUALIFY function
description, 481
qualify method
    of Stream class, 230
query external function, 483
QUERY keyword
    in a RXSUBCOM command, 636
query method
    of Stream class, 230
queryMixinClass method
    of Class class, 130
queue
    creating and deleting queues, 484
    named, 621
    naming and querying, 484
    RXQUEUE function, 484
    session, 621
    unnamed, 621
Queue class, 292
QUEUE instruction
description, 72
    example, 72
    role in input and output, 621
Queue interface from Rexx programs, 484
queue method
    of CircularQueue class, 270
    of Queue class, 297
    of RexxQueue class, 413
QUEUED function
description, 481
    example, 481
    role in input and output, 622
queued method
    of RexxQueue class, 413
RC (return code)
    not set during interactive debug, 629
    set by commands, 38
    special variable, 599, 632
RC special variable
description, 632
read position in a stream, 620
recursive call, 49, 444
register external functions, 483
REGISTER keyword
    in a RXSUBCOM command, 635
RegularExpression class, 392
Relation class, 298
relative positional pattern
    positional patterns
    relative, 572
release method
    of MutexSemaphore class, 389
    of RexxInfo class, 408
remainder
description>, 586
remainder operator, 19
remove method
    of Array class, 257
    of Bag class, 265
    of Directory class, 277
    of IdentityTable class, 282
    of List class, 288
    of Queue class, 297
    of Relation class, 302
    of Set class, 307
    of Stem class, 312
    of StringTable class, 316
    of Table class, 320
removeAll method
of Bag class, 265
of Relation class, 302
removeEntry method
of Directory class, 277
of StringTable class, 317
removeItem method
of Array class, 258
of Bag class, 266
of Directory class, 277
of IdentityTable class, 282
of List class, 288
of Queue class, 297
of Relation class, 302
of Set class, 308
of Stem class, 312
of StringTable class, 317
of Table class, 321
renameTo method
of File class, 370
reordering data, 208, 385, 501
repeating a string with COPIES, 462
repeating s string with copies, 192
repetitive loops
altering flow, 62
controlled repetitive loops, 641
exiting, 62
simple DO group, 641
REPLACE subkeyword
in an ADDRESS instruction, 42
replaceAt method
of MutableBuffer class, 383
of String class, 204
replacing characters within a string, 204, 383
REPLY instruction
description, 75
element, 75
reply method
of Message class, 137
replyWith method
of Message class, 137
request method
of Object class, 152
of Stem class, 312
of VariableReference class, 435
requestClassType method
of Validate class, 433
reservation of keywords, 631
reset method
of EventSemaphore class, 358
resize method
of CircularQueue class, 270
resource method
of Package class, 165
resources method
of Package class, 165
restrictions
embedded in numbers, 14
first character of variable name, 28
in programming, 1
result method
of Message class, 138
RESULT special variable
description, 632
return value from a routine, 448
set by RETURN instruction, 49, 76
retrieving
argument strings with ARG, 46
arguments with ARG function, 451
lines with SOURCELINE, 486
return
code
as set by commands, 38
setting on exit, 53
string
setting on exit, 53
RETURN instruction
description, 76
RETURN subkeyword
in a RAISE instruction, 73
returning control from Rexx program, 76
REVERSE function
description, 482
element, 482
reverse method
of String class, 204
revision method
of RexxInfo class, 409
RexxContext class, 397
RexxInfo class, 400
RexxQueue class, 409
rexxutil functions, 512
RxMessageBox, 514
element, 516
RxWinExec, 516
SysAddRexxMacro, 518
SysBootDrive, 518
SysClearRexxMacroSpace, 518
SysCls, 518
SysCreatePipe, 519
element, 519
SysCurPos, 519
SysCurState, 519
SysDriveInfo, 520
element, 520
SysDriveMap, 520
element, 521
SysDropRexxMacro, 521
SysDumpVariables, 521
example, 522
SysFileCopy, 522
   example, 522
SysFileDelete, 522
   example, 523
SysFileExists, 523
SysFileMove, 524
   example, 524
SysFileSearch, 524
   example, 525
SysFileSystemType, 526
   example, 526
SysFileTree, 526
   example, 529
SysFork, 529
SysFormatMessage, 531
   example, 531
SysGetErrorCode, 531
SysGetErrorText, 533
   example, 533
SysGetFileDateTime, 533
   example, 534
SysGetKey, 534
SysGetLongPathName, 534
   example, 535
SysGetMessage, 535
   example, 535
SysGetMessageX, 535
   example, 536
SysGetShortPathName, 536
   example, 536
SysIni, 537
   example, 538
SysIsFile, 539
SysIsFileCompressed, 539
SysIsFileDirectory, 540
SysIsFileEncrypted, 540
SysIsFileLink, 541
SysIsFileNotContentIndexed, 541
SysIsFileOffline, 542
SysIsFileSparse, 542
SysIsFileTemporary, 543
SysLinVer, 543
SysLoadRexxMacroSpace, 543
SysMkDir, 544
   example, 545
SysQueryProcess, 545
SysQueryRexxMacro, 546
SysReorderRexxMacro, 547
SysRmDir, 547
   example, 548
SysSaveRexxMacroSpace, 548
SysSearchPath, 548
   example, 549
SysSetFileDateTime, 549
   example, 550
SysSetPriority, 550
SysShutdownSystem, 551
SysSleep, 552
   example, 552
SysStemCopy, 553
   example, 554
SysStemDelete, 554
   example, 554
SysStemInsert, 555
SysStemSort, 555
   example, 556
SysSwitchSession, 556
SysSystemDirectory, 557
SysTempFileName, 557
   example, 558
SysTextScreenRead, 558
   example, 558
SysTextScreenSize, 559
   example, 560
SysToUnicode, 560
SysUtilVersion, 561
SysVersion, 562
SysVolumeLabel, 562
SysWait, 562
SysWaitNamedPipe, 563
SysWinDecryptFile, 563
SysWinEncryptFile, 564
SysWinGetDefaultPrinter, 564
   example, 565
SysWinGetPrinters, 564
   example, 565
SysWinSetDefaultPrinter, 565
   example, 565
SysWinVer, 566
RIGHT function
description, 482
example, 482
right method
   of String class, 204
round method
   of String class, 205
rounding
   using a character string as a number, 14
Routine class, 166
ROUTINE subkeyword
   in an ANNOTATE directive, 90
routines method
   of Package class, 165
rs method
   of RexxContext class, 399
run method
   of Object class, 153
running off the end of a program, 76
RXFUNCADD function
description, 483
example, 483
RXFUNCDROP function
description, 483
example, 483
RXFUNCQUERY function
description, 483
example, 483
RXFUNCQUEUE function
element, 483
RxMessageBox, 514
element, 516
RXQUEUE filter, 637
RXQUEUE function
description, 484
RXSUBCOM command, 635
RXTRACE environment variable, 630
RxWinExec, 516
S
save method
of Properties class, 291
SAY instruction
description, 76
displaying data, 76
element, 77
role in output, 620
say method
of RexxQueue class, 413
of Stream class, 233
scheduledTime method
of Alarm class, 331
scientific notation, 587
SCIENTIFIC subkeyword
in a NUMERIC instruction, 64
scope
alternating exclusive access, 607
description, 113
scope method
of Method class, 145
search order
external functions, 444
for functions, 444
for methods
changing, 114
default, 113
for subroutines, 48
searchPath method
of File class, 361
seconds calculated from midnight, 498
seconds method
of DateTime class, 350
of TimeSpan class, 428
section method
of Array class, 258
of CircularQueue class, 271
of List class, 288
of OrderedCollection class, 245
of Queue class, 298
Security Manager, 614
calls to, 614
seek method
of Stream class, 233
SELECT instruction
description, 77
element, 78
selecting a default with ABBREV function, 450
selecting a default with abbrev method, 178
selecting a default with caselessAbbrev method, 182
SELF special variable
description, 632
semaphore, 608
semicolon
implied, 16
omission of, 42
within a clause, 8
send method
of Message class, 138
of Object class, 153
sendWith method
of Message class, 139
of Object class, 154
separator method
of File class, 361, 370
sequence, collating using XRANGE, 510
serial input and output, 619
Set class, 304
Set Collection classes
Bag class, 261
Set class, 304
set method
of RexxQueue class, 413
SET subkeyword
in an ATTRIBUTE directive, 91
set-operator methods, 324
setBufferSize method
of MutableBuffer class, 383
SetCollection class, 247
setEntry method
of Directory class, 277
of StringTable class, 317
setGuards method
of Method class, 146
SETLOCAL function
description, 485
example, 485
setLogical method
  of Properties class, 291
setMethod method
  of Directory class, 278
  of Object class, 155
setPrivate method
  of Method class, 146
setProperty method
  of Properties class, 291
setProtected method
  of Method class, 146
setReadOnly method
  of File class, 370
setSecurityManager method
  of Method class, 146
  of Package class, 166
  of Routine class, 169
setText method
  of MutableBuffer class, 383
setUnguarded method
  of Method class, 146
SETUNGUARDED method, 606
size method
  of Array class, 258
  of CircularQueue class, 271
  of Queue class, 298
sort method
  of Array class, 259
  of OrderedCollection class, 245
sortBy method
  of Array class, 259
  of OrderedCollection class, 246
source
  of program and retrieval of information, 68
  string, 567
source method
  of Method class, 146
  of Package class, 166
  of Routine class, 169
SOURCE option of PARSE instruction, 68
sourceSize method
  of Package class, 166
sourceSize method
  of Package class, 166
SPACE function
  description, 486
  example, 487
space method
  of MutableBuffer class, 383
  of String class, 173, 206
Spain, formatting, SPACE function, 486
space method
  of Package class, 166
special
  characters and example, 15
  parsing case, 579
  variable
    RC, 632
    RESULT, 49, 76, 448, 632
    SELF, 632
    SIGL, 49, 632
    SUPER, 632
  variables
    RC, 38, 598, 632
    RESULT, 76, 448, 632
    SELF, 632
    SIGL, 598, 632
    SUPER, 632
stableSort method
  of Array class, 259
  of OrderedCollection class, 246
stableSortWith method
  of Array class, 259
  of OrderedCollection class, 246
StackFrame class, 415
stackFrames method
  of RexxContext class, 399
standard input and output, 623
Standard option of DATE function, 467
standardDate method
  of DateTime class, 354
start method
  of Message class, 139
  of Object class, 155
startsWith method
  of MutableBuffer class, 384
  of String class, 206
startWith method
  of Message class, 140
  of Object class, 156
state method
  of Stream class, 235
State method, 627
Stem class, 308
stem of a variable
  assignment to, 31
  description, 33
  used in DROP instruction, 53
  used in PROCEDURE instruction, 70
STEM subkeyword
  in an ADDRESS instruction, 42
steps in parsing, 580
stream, 619
  character positioning, 624
  function overview, 625
  line positioning, 624
Stream class, 218
stream errors, 626
STREAM function
  command options, 488
  command strings, 488
  description, 487
  example, 491, 492
  options, 489
  query options, 492
STREAM subkeyword
  in an ADDRESS instruction, 42
StreamSupplier class, 417
strict comparison, 20, 20
STRICK subkeyword
  in a USE instruction, 86
strictly equal operator, 20, 20
strictly greater than operator, 20, 20
strictly greater than or equal operator, 20
strictly less than operator, 20, 20
strictly not equal operator, 20, 20
strictly not greater than operator, 20
strictly not less than operator, 20
string
  as literal constant, 11
  as name of function, 11
  as name of subroutine, 47
  binary specification of, 12
  centering using center function, 189
  centering using CENTER method, 457
  centering using centre function, 189
  centering using CENTRE method, 457
  comparison of, 20
  concatenation of, 18
  copying using copies, 192
  copying using COPIES, 462
  decodeBase64 method, 195
delating part, DELSTR function, 469
deleting part, delStr method, 195
description, 11
cencodeBase64 method, 196
eextracting using SUBSTR function, 495
extracting using substr method, 207
eextracting words with SUBWORD, 496
from stream, 458
hexadecimal specification of, 12
interpretation of, 59
lowercasing using LOWER function, 479
null, 11
patterns
  description, 567
  literal, 570
  variable, 576
quotations marks in, 11
repeating using copies, 192
repeating using COPIES, 462
uppercasing using UPPER function, 502
verifying contents of, 210, 385, 506
String class, 169
string method
  of CircularQueue class, 271
  of daysInMonth class, 357, 430
  of File class, 371
  of MutableBuffer class, 384
  of Object class, 156
  of StackFrame class, 416
  of Stream class, 235
StringTable class, 313
STRIP function
  description, 494
  example, 495
strip method
  of String class, 206
structure and syntax, 8
subchar method
  of String class, 207
subChar method
  of MutableBuffer class, 384
subclass method
  of Class class, 130
SUBCLASS subkeyword
  in a CLASS directive, 94
subclasses, 7
subclasses method
  of Class class, 131
subexpression, 18
subkeyword, 28
subroutines
  calling of, 47
  definition, 443
  forcing built-in or external reference, 48
  naming of, 47
  passing back values from, 76
  return from, 76
  use of labels, 47
  variables in, 69
subset method
  of Bag class, 266
  of Collection class, 241
  of OrderedCollection class, 246
  of Relation class, 302
subsidiary list, 52, 54, 69
substitution
  in variable names, 33
SUBSTR function
  description, 495
  example, 495
substr method
  of MutableBuffer class, 384
  of String class, 207
subtraction operator, 19
SUBWORD function
  description, 496
  example, 496
subWord method
  of MutableBuffer class, 384
  of String class, 207
subWords method
  of MutableBuffer class, 384
  of String class, 208
summary
  parsing instructions, 577
SUPER special variable
  description, 632
superClass method
  of Class class, 131
superclasses, 7
superClasses method
  of Class class, 131
Supplier class, 419
supplier method
  of Array class, 259
  of Bag class, 266
  of CircularQueue class, 272
  of Collection class, 241
  of Directory class, 278
  of IdentityTable class, 282
  of List class, 288
  of Queue class, 298
  of Relation class, 303
  of Set class, 308
  of Stem class, 313
  of Stream class, 235
  of StringTable class, 317
  of Supplier class, 421
  of Table class, 321
symbol
  assigning values to, 28, 29
  classifying, 29
  compound, 33
  constant, 29
  description, 13
  simple, 28
  uppercase translation, 13
  use of, 28, 29
  valid names, 13
SYMBOL function
  description, 496
  example, 496
symbols
  environment, 34
  namespace, 35
syntax
  error
    traceback after, 86
    trapping with SIGNAL instruction, 592
  general, 8
SYNTAX subkeyword
  in a RAISE instruction, 73
  in a SIGNAL instruction, 79, 594, 598
SysAddBootDrive, 518
SysAddRexxMacro, 518
SysClearRexxMacroSpace, 518
SysCls, 518
SysCreatePipe, 519
SysCurPos, 519
  example, 519
SysCurState, 519
SysDriveInfo, 520
  example, 520
SysDriveMap, 520
  example, 521
SysDropRexxMacro, 521
SysDumpVariables, 521
  example, 522
SysFileCopy, 522
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SysFileDelete</td>
<td>522</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysFileExists</td>
<td>523</td>
</tr>
<tr>
<td>SysFileMove</td>
<td>524</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysFileSearch</td>
<td>524</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysFileSystemType</td>
<td>526</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysFileTree</td>
<td>526</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysFork</td>
<td>529</td>
</tr>
<tr>
<td>SysFormatMessage</td>
<td>531</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysFromUnicode</td>
<td>531</td>
</tr>
<tr>
<td>SysGetErrorText</td>
<td>533</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysGetFileDateTime</td>
<td>533</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysGetKey</td>
<td>534</td>
</tr>
<tr>
<td>SysGetLongPathName</td>
<td>535</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysGetLongPathNamey</td>
<td>534</td>
</tr>
<tr>
<td>SysGetMessage</td>
<td>535</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysGetMessageX</td>
<td>535</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysGetShortPathName</td>
<td>536</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysGetShortPathNamey</td>
<td>536</td>
</tr>
<tr>
<td>SysIni</td>
<td>537</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysIsFile</td>
<td>539</td>
</tr>
<tr>
<td>SysIsFileCompressed</td>
<td>539</td>
</tr>
<tr>
<td>SysIsFileDirectory</td>
<td>540</td>
</tr>
<tr>
<td>SysIsFileEncrypted</td>
<td>540</td>
</tr>
<tr>
<td>SysIsFileLink</td>
<td>541</td>
</tr>
<tr>
<td>SysIsFileNotContentIndexed</td>
<td>541</td>
</tr>
<tr>
<td>SysIsFileOffline</td>
<td>542</td>
</tr>
<tr>
<td>SysIsFileSparse</td>
<td>542</td>
</tr>
<tr>
<td>SysIsFileTemporary</td>
<td>543</td>
</tr>
<tr>
<td>SysLinVer</td>
<td>543</td>
</tr>
<tr>
<td>SysLoadRexxMacroSpace</td>
<td>543</td>
</tr>
<tr>
<td>SysMdKDir</td>
<td>544</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysQueryProcess</td>
<td>545</td>
</tr>
<tr>
<td>SysQueryRexxMacro</td>
<td>546</td>
</tr>
<tr>
<td>SysReorderRexxmacro</td>
<td>547</td>
</tr>
<tr>
<td>SysRmDir</td>
<td>547</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysSaveRexxMacroSpace</td>
<td>548</td>
</tr>
<tr>
<td>SysSearchPath</td>
<td>548</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysSetFileDateTime</td>
<td>549</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysSetPriority</td>
<td>550</td>
</tr>
<tr>
<td>SysShutdownSystem</td>
<td>551</td>
</tr>
<tr>
<td>SysSleep</td>
<td>552</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysStemCopy</td>
<td>553</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysStemDelete</td>
<td>554</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysStemInsert</td>
<td>555</td>
</tr>
<tr>
<td>SysStemSort</td>
<td>555</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysSwitchSession</td>
<td>556</td>
</tr>
<tr>
<td>SysSystemDirectory</td>
<td>557</td>
</tr>
<tr>
<td>SysTempFileName</td>
<td>557</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysTextScreenRead</td>
<td>558</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysTextScreenSize</td>
<td>559</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysTextScreenSize</td>
<td>559</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysToUnicode</td>
<td>560</td>
</tr>
<tr>
<td>SysUtilVersion</td>
<td>561</td>
</tr>
<tr>
<td>SysVersion</td>
<td>562</td>
</tr>
<tr>
<td>SysVolumeLabel</td>
<td>562</td>
</tr>
<tr>
<td>SysWait</td>
<td>562</td>
</tr>
<tr>
<td>SysWaitNamedPipe</td>
<td>563</td>
</tr>
<tr>
<td>SysWinDecryptFile</td>
<td>563</td>
</tr>
<tr>
<td>SysWinEncryptFile</td>
<td>564</td>
</tr>
<tr>
<td>SysWinGetDefaultPrinter</td>
<td>564</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysWinGetPrinters</td>
<td>564</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysWinSetDefaultPrinter</td>
<td>565</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>SysWinVer</td>
<td>566</td>
</tr>
</tbody>
</table>

**T**

- tab method
  - of `String` class, 173
- Table class, 318
- tail, 33
- target method
  - of `Message` class, 141
  - of `StackFrame` class, 416
- template
  - definition, 567
  - list
    - ARG instruction, 46
    - PARSE instruction, 66
    - PULL instruction, 71
- temporary change of, 42
- temporary command destination change, 42
- temporaryPath method
of File class, 362
ten, powers of, 587

terminal
    reading from with PULL, 71
    writing to with SAY, 76
terms and data, 17
testing, 450, 496
    abbreviations with abbrev method, 178
    abbreviations with caselessAbbrev method, 182
THEN
    as free standing clause, 42
    following IF clause, 58
    following WHEN clause, 77
THEN subkeyword
    in a SELECT instruction, 77
    in an IF instruction, 58
thread, 532, 545, 560, 601
Ticker class, 421
ticks method
    of DateTime class, 356
Ticks option of DATE function, 467
Ticks option of Time function, 498
tilde (~), 4
TIME function
    description, 497
    example, 499, 499
    implementation maximum, 500
timeOfDay method
    of DateTime class, 356
TimeSpan class, 424
tips, tracing, 84
TO phrase of DO instruction, 51
TO subkeyword
    in a DO instruction, 50, 641
    in a FORWARD instruction, 55
    in a LOOP instruction, 63, 641
today method
    of DateTime class, 341
toDirectory method
    of Stem class, 313
tokens
    binary strings, 12
    description, 11
    hexadecimal strings, 12
    literal strings, 11
    numbers, 14
    operator characters, 15
    special characters, 15
    symbols, 13
toLocalTime method
    of DateTime class, 355
toString method
    of Array class, 259

    totalDays method
    of TimeSpan class, 428
    totalHours method
    of TimeSpan class, 428
    totalMicroseconds method
    of TimeSpan class, 429
    totalMinutes method
    of TimeSpan class, 429
    totalSeconds method
    of TimeSpan class, 429
toTimezone method
    of DateTime class, 355
toUtcTime method
    of DateTime class, 355
TRACE function
    description, 500
    example, 501
TRACE instruction
    alphabetical character word options, 82
    description, 81
    example, 84
trace method
    of Package class, 166
TRACE setting
    altering with TRACE function, 500
    altering with TRACE instruction, 81
    querying, 500
traceback, on syntax error, 86
traceLine method
    of StackFrame class, 416
tracing
    action saved during subroutine calls, 50
    by interactive debug, 629
    data identifiers, 84
    execution of programs, 81
    tips, 84
tracing flag
    -*-*, 84
    ++++, 85
    =>>>, 85
    >>>, 85
    >>>>>, 85
    >>A>, 85
    >>C>, 85
    >>E>, 85
    >>F>, 85
    >>I>, 85
    >>K>, 85
    >>L>, 85
    >>M>, 85
    >>N>, 85
    >>O>, 85
    >>P>, 86
    >>R>, 86
>V>, 86

trailing
whitespace removed using STRIP function, 494
whitespace removed using strip method, 206
transient input and output, 619
TRANSLATE function
description, 501
example, 501
translate method
of MutableBuffer class, 385
of String class, 208
translation
with TRANSLATE function, 501
with translate method, 208, 385
trap conditions
explanation, 591
how to, 591
information about trapped conditions, 460
using CONDITION function, 460
trapname, 595
triggered method
of Alarm class, 331
of AlarmNotification class, 332
of Message class, 141
TRUNC function
description, 502
example, 502
trunc method
of String class, 209
truncating numbers, 209, 502
twiddle (~), 4
type method
of StackFrame class, 416
type of data, checking with DataType, 193
type of data, checking with DATATYPE, 464
typewriter input and output, 619

U
unassigning variables, 52
unconditionally leaving your program, 53
underflow, arithmetic, 590
UNGUARDED option of ::ATTRIBUTE, 94
UNGUARDED option of ::METHOD, 99, 606
UNGUARDED subkeyword
in a METHOD directive, 97
in an ATTRIBUTE directive, 91
uninher method
of Class class, 132
uninit method
of EventSemaphore class, 358
of MutexSemaphore class, 389
of Stream class, 236
uninitialized variable, 29
union method
of Bag class, 266
of CircularQueue class, 272
of Collection class, 241
of OrderedCollection class, 246
of Relation class, 303
of Set class, 308
uniqueIndexes method
of Bag class, 266
of Relation class, 303
unknown method
of Directory class, 278
of Monitor class, 372
of Stem class, 313
of StringTable class, 317
of VariableReference class, 435
unpacking a string
with b2x, 179
with B2X, 453
with c2x, 181
with C2X, 456
UNPROTECTED subkeyword
in a METHOD directive, 97
in an ATTRIBUTE directive, 91
unsetMethod method
of Directory class, 278
of Object class, 157
UNTIL phrase of DO instruction, 51
UNTIL phrase of LOOP instruction, 63
UNTIL subkeyword
in a DO instruction, 50, 644
in a LOOP instruction, 63, 644
unusual change in flow of control, 591
UPPER function
description, 502
upper method
of MutableBuffer class, 385
of String class, 173, 210
UPPER subkeyword
in a PARSE instruction, 46, 66, 71, 576
uppercase translation
during ARG instruction, 46
during PULL instruction, 71
of symbols, 13
with PARSE UPPER, 67
with TRANSLATE function, 501
with translate method, 208, 385
Usa option of DATE function, 467
usaDate method
of DateTime class, 354
USE instruction
description, 86
example, 87
USE LOCAL instruction
example, 89
user input and output, 619, 629
USER subkeyword
  in a CALL instruction, 47, 595, 598
  in a RAISE instruction, 73
  in a SIGNAL instruction, 79, 595, 598
USERID function
  description, 503
USING subkeyword
  in an ADDRESS instruction, 42
utcDate method
    of DateTime class, 355
utcIsoDate method
    of DateTime class, 352
V
Validate class, 431
value, 17
VALUE function
  description, 503
  example, 503, 505
value method
  of VariableReference class, 435
  of WeakReference class, 436
VALUE option of PARSE instruction, 68
VALUE subkeyword
  in a NUMERIC instruction, 64
  in a PARSE instruction, 66, 577
  in a SIGNAL instruction, 79
  in a TRACE instruction, 81
  in an ADDRESS instruction, 42
VAR function
  description, 505
  example, 505
VAR option of PARSE instruction, 68
VAR subkeyword
  in a PARSE instruction, 66, 577
variable
  access with VALUE function, 503
  checking name, 505
  compound, 33
  controlling loops, 641
description, 28, 29
dropping of, 52
exposing to caller, 69
external collections, 504
global, 504
in internal functions, 69
in subroutines, 69
names, 13
new level of, 69
 parsing of, 68
patterns, parsing with
  string, 576
patterns, parsing with positional, 576
positional patterns, 576
reference, 576
resetting of, 52
setting a new value, 28, 29
SIGL, 598
simple, 30
special
  RC, 38
  SIGL, 49, 598
string patterns, 576
testing for initialization, 496
valid names, 28
variable initialization, 496
VariableReference class, 434
variables
  acquiring, 7, 8
  in objects, 4
variables method
  of RexxContext class, 399
VERIFY function
  description, 506
  example, 506
verify method
  of MutableBuffer class, 385
  of String class, 210
verifying contents of a buffer, 385
verifying contents of a string, 210, 506
version method
  of RexxInfo class, 409
VERSION option of PARSE instruction, 69
VERSION subkeyword
  in a PARSE instruction, 66, 577
W
wait method
  of Message class, 142
Wait method
  of EventSemaphore class, 359
WeakReference class, 435
weekDay method
  of DateTime class, 353
Weekday option of DATE function, 467
weekNumber method
  of DateTime class, 347
weekNumberDate method
  of DateTime class, 348
weekNumberYear method
  of DateTime class, 348
weeksInYear method
  of DateTime class, 348
WHEN
  as free standing clause, 42
WHEN subkeyword
in a GUARD instruction, 57
in a SELECT instruction, 77
WHILE phrase of DO instruction, 51
WHILE phrase of LOOP instruction, 63
WHILE subkeyword
in a DO instruction, 50, 644
in a LOOP instruction, 63, 644
whitespace, 9
adjacent to special character, 8
in parsing, treatment of, 568
removal with STRIP function, 494
removal with strip method, 206
whole numbers
checking with dataType, 193
checking with DATATYPE, 464
description, 14
wholeNumber method
of Validate class, 433
wholeNumberRange method
of Validate class, 434
Windows System Error Codes, 512
WITH phrase of DO instruction, 50
WITH phrase of LOOP instruction, 63
WITH subkeyword
in a DO instruction, 50, 644
in a LOOP instruction, 63, 644
in an ADDRESS instruction, 42
word
alphabetical character options in TRACE, 82
counting in a mutable buffer, 387
counting in a string, 212, 508
deleting from a mutablebuffer, 379
deleting from a string, 196, 470
extracting from a mutable buffer, 387
extracting from a string, 207, 208, 212, 384, 384, 496, 508
finding length of, 200, 475
in parsing, 567
locating in a mutable buffer, 387
locating in a string, 188, 212, 377, 507
parsing
conceptual view, 583
examples, 567
WORD function
description, 506
example, 507
word method
of MutableBuffer class, 386
of String class, 211
WORLDLENGTH function
description, 507
example, 507
wordLength method
of MutableBuffer class, 387
of String class, 212
WORDPOS function
description, 507
example, 508
wordPos method
of MutableBuffer class, 387
of String class, 212
WORDS function
description, 508
example, 508
words method
of MutableBuffer class, 387
of String class, 212
write position in a stream, 620
writing to external data queue
with PUSH, 72
with QUEUE, 72
X
X2B function
description, 508
example, 508
x2b method
of String class, 213
X2C function
description, 509
example, 509
x2c method
of String class, 213
X2D function
description, 509
example, 509, 510
x2d method
of String class, 214
xdigit method
of String class, 173
xor method
of Bag class, 266
of CircularQueue class, 272
of Collection class, 241
of OrderedCollection class, 247
of Relation class, 303
of Set class, 308
XOR, logical, 21
XORing character strings together, 180, 455
XRANGE function
description, 510
example, 511

Y
year method
  of DateTime class, 349
yearDay method
  of DateTime class, 352

Z
zeros
  added on left with RIGHT function, 482
  added on left with right method, 204
  removal with STRIP function, 494
  removal with strip method, 206