ooRexx
Documentation 5.0.0

Open Object Rexx

Programmer Guide

ooRexx Documentation 5.0.0 Open Object Rexx Programmer Guide Edition 2022.12.22

Author	W. David Ashley
Author	Rony G. Flatscher
Author	Mark Hessling
Author	Rick McGuire
Author	Lee Peedin
Author	Oliver Sims
Author	Erich Steinböck
Author	Jon Wolfers

Preface

This book describes the Open Object Rexx, or ooRexx programming language. In the following, it is called Rexx unless compared to its traditional predecessor.

This book is aimed at developers who want to use Rexx for object-oriented programming, or a mix of traditional and object-oriented programming.

This book assumes you are already familiar with the techniques of traditional structured programming, and uses them as a springboard for quickly understanding Rexx and, in particular, ooRexx. This approach is designed to help experienced programmers get involved quickly with the Rexx language, exploit its virtues, and become productive fast.

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.

```
-- reverse an array
a = .Array~of("one", "two", "three", "four", "five")

-- five, four, three, two, one
aReverse = .CircularQueue~new(a~size)~appendAll(a)~makeArray("lifo")
```

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.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed, like mandatory initialization. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

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.

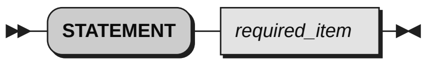
  The ▶▶— symbol indicates the beginning of a statement.

  The —··· symbol indicates that the statement syntax is continued on the next line.

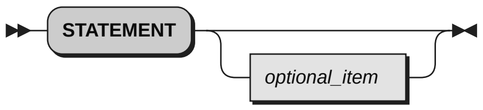
  The ···— symbol indicates that a statement is continued from the previous line.

  The —▶◀ symbol indicates the end of a statement.

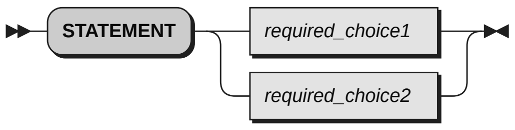
- Required items appear on the horizontal line (the main path).



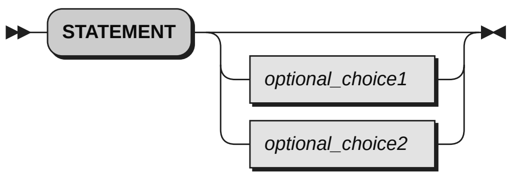
- Optional items appear below the main path.



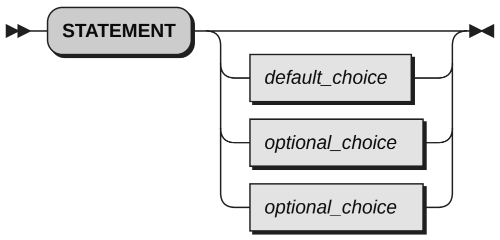
- If you can choose from two or more items, they appear vertically, in a stack. If you must choose one of the items, one item of the stack appears on the main path.



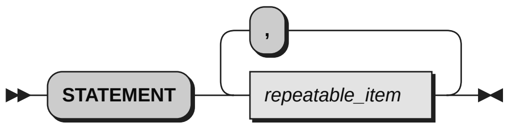
- If choosing one of the items is optional, the entire stack appears below the main path.



- If one of the items is the default, it is usually the topmost item of the stack of items below the main path.

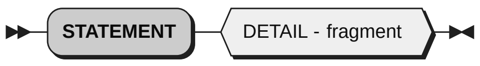


- A path returning to the left above the main line indicates an item that can be repeated.



  A repeat path above a stack indicates that you can repeat the items in the stack.

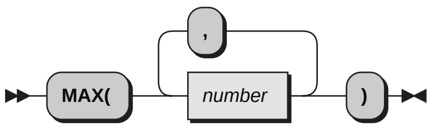
- A pointed rectangle around an item indicates that the item is a fragment, a part of the syntax diagram that appears in greater detail below the main diagram.



- Keywords appear in uppercase (for example, **`SIGNAL`**). They must be spelled exactly as shown but you can type them in upper, lower, or mixed case. Variables appear in all lowercase letters (for example, *index*). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.

The following example shows how the syntax is described:



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
: Subscribe to the oorexx-devel mailing list at *https://lists.sourceforge.net/lists/listinfo/oorexx-devel* to discuss ooRexx project development activities and future interpreter enhancements. You can find its archive of past messages at *http://sourceforge.net/mailarchive/forum.php?forum_name=oorexx-devel*.

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 announcments 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

The Rexx Language Association maintains a forum at *http://www.rexxla.org/forum.html*.

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: Reference*

Meet Open Object Rexx (ooRexx)

Rexx is a versatile, free-format language. Its simplicity makes it a good first language for beginners. For more experienced users and computer professionals, Rexx offers powerful functions and the ability to issue commands to several environments.

1.1. The Main Attractions

The following aspects of Rexx round out its versatility and functions.

1.1.1. Object-Oriented Programming

Object-oriented extensions have been added to traditional Rexx, but its existing functions and instructions have not changed. The Open Object Rexx interpreter is an enhanced version of its predecessor with support for:

- Classes, objects, and methods
- Messaging and polymorphism
- Inheritance and multiple inheritance

Rexx supplies the user with a base set of built-in classes providing many useful functions. Open Object Rexx is fully compatible with earlier versions of Rexx that were not object-oriented.

1.1.2. An English-Like Language

To make Rexx easier to learn and use, many of its instructions are meaningful English words. Rexx instructions are common words such as SAY, PULL, IF...THEN...ELSE, DO...END, and EXIT.

1.1.3. Cross-Platform Versatility

Versions of ooRexx are now available for a wide variety of platforms, and the programs you create with ooRexx will run on any of these. There are 32-bit and 64-bit versions that can exploit larger address spaces.

1.1.4. Fewer Rules

Rexx has relatively few rules about format. A single instruction can span many lines, and you can include several instructions on a single line. Instructions need not begin in a particular column and can be typed in uppercase, lowercase, or mixed case. You can skip spaces in a line or entire lines. There is no line numbering.

1.1.5. Interpreted, Not Compiled

Rexx is an interpreted language. When a Rexx program runs, its language processor reads each statement from the source file and runs it, one statement at a time. Languages that are not interpreted must be compiled into object code before they can be run.

1.1.6. Built-In Classes and Functions

Rexx has built-in classes and functions that perform various processing, searching, and comparison operations for text and numbers and provide formatting capabilities and arithmetic calculations.

1.1.7. Typeless Variables

Rexx regards all data as objects of various kinds. Variables can hold any kind of object, so you need not declare variables as strings or numbers.

1.1.8. String Handling

Rexx includes capabilities for manipulating character strings. This allows programs to read and separate characters, numbers, and mixed input. Rexx performs arithmetic operations on any string that represents a valid number, including those in exponential formats.

1.1.9. Clear Error Messages and Powerful Debugging

Rexx displays messages with meaningful explanations when a Rexx program encounters an error. In addition, the TRACE instruction provides a powerful debugging tool.

1.1.10. Impressive Development Tools

The ooRexx places many powerful tools at your disposal. These include a Rexx API to other languages like C/C++ or Cobol, OLE/ActiveX support, a mathematical functions package.

1.2. Rexx and the Operating System

The most important role Rexx plays is as a programming language for Windows and Unix-based systems. A Rexx program can serve as a script for the operating system. Using Rexx, you can reduce long, complex, or repetitious tasks to a single command or program.

1.3. A Classic Language Gets Classier

Object-oriented extensions have been added to traditional Rexx without changing its existing functions and instructions. So you can continue to use Rexx's procedural instructions, and incorporate objects as you become more comfortable with the technology. In general, your current Rexx programs will work without change.

In object-oriented technology, *objects* are used in programs to model the real world. Similar objects are grouped into *classes*, and the classes themselves are arranged in hierarchies.

As an object-oriented programmer, you solve problems by identifying and classifying objects related to the problem. Then you determine what actions or behaviors are required of those objects. Finally, you write the instructions to generate the classes, create the objects, and implement the actions. Your main program consists of instructions that send messages to objects.

A billing application, for example, might have an Invoice class and a Receipt class. These two classes might be members of a Forms class. Individual invoices are *instances* of the Invoice class.

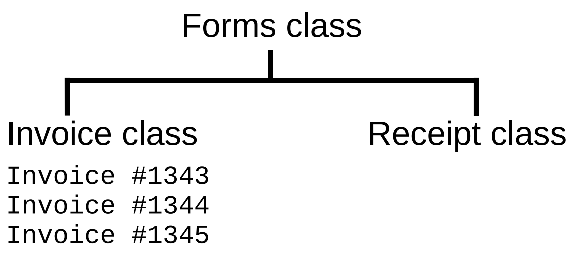


Figure 1.1. Objects in a Billing Application

Each instance contains all the data associated with it (such as customer name or descriptions and prices of items purchased). To get at the data, you write instructions that send messages to the objects. These messages activate coded actions called methods. For an invoice object, you might need CREATE, DISPLAY, PRINT, UPDATE, and ERASE methods.

1.3.1. From Traditional Rexx to Open Object Rexx

In traditional (classic) Rexx, all data was stored as strings. The strings represented character data as well as numeric data. From an object-oriented perspective, traditional Rexx had just one kind of object: the string. In object-oriented terminology, each string variable is an *object* that is a reference to an instance of the String class.

With ooRexx, variables can now reference objects other than strings. In addition to the String class, Rexx includes classes for creating arrays, queues, streams, and many other useful objects. Additionally, you can create your own classes that can interoperate seamlessly with the language built-in classes. Objects in these Rexx classes are manipulated by methods instead of traditional functions. To activate a method, you just send a message to the object.

For example, instead of using the SUBSTR function on a string variable Name, you send a SUBSTR message to the string object. In classic Rexx, you would do the following:

```
s=substr(name,2,3)
```

In Open Object Rexx, the equivalent would be:

```
s=name~substr(2,3)
```

The tilde (**~**) character is the Rexx message send operator, called *twiddle*. The object receiving the message is to the left of the twiddle. The message sent is to the right. In this example, the Name object is sent the SUBSTR message. The numbers in parentheses *(2,3)* are arguments sent as part of the message. The SUBSTR method is run for the Name object, and the result is assigned to the *s* string object. Conceptually, you are "asking" the String object referred to by variable NAME to give you its substring starting at character 2 for 3 characters. Many String operations are available as both class methods and built-in functions, but the String class also provides many method enhancements for which there are no corresponding built-in functions.

For classes other than String (such as Array or Queue), methods are provided, but not equivalent built-in functions. For example, suppose you want to use a Rexx Array object instead of the traditional string-based stem variables (such as *text.1* or *text.2*). To create an array object of five elements, you would send a NEW message to the Array class as follows:

```
myarray=.array~new(5)
```

A new instance of the Array class is created and reference to it is stored in the variable **`myarray`**. A period and the class name identify the class, **`.Array`**. The period tells the interpreter to look in the Rexx environment for a class named "ARRAY". The **`myarray`** Array object has five elements, but the array itself is currently empty. Items can be added using methods defined by the Array class.

```
myarray[1] = "Rick"
myarray[2] = "David"
myarray[3] = "Mark"

say myarray[1] myarray[2] myarray[3]
```

The Array class implements many more methods. See the ooRexx reference for details on what additional methods are provided.

By adding object technology to its repertoire of traditional programming techniques, Rexx has evolved into an object-oriented language, like Smalltalk. Rexx accommodates the programming techniques of traditional Rexx while adding new ones. With ooRexx, you can use the new technology as much or as little as you like, at whatever pace you like. You can mix classic and object techniques. You can ease into the object world gradually, building on the Rexx skills and knowledge you already have.

1.4. The Object Advantage

If you are unsure about whether to employ Rexx's object-oriented features, here are some tips to help you decide.

Object-oriented technology reinforces good programming practices, such as hiding your data from code that does not use it (encapsulation and polymorphism), partitioning your program in small, manageable units (classification and data abstraction), reusing code wherever possible and changing it in one place (inheritance and functional decomposition).

Other advantages often associated with object technology are:

- Simplified design through modeling with objects
- Greater code reuse
- Rapid prototyping
- The higher quality of proven components
- Easier and reduced maintenance
- Cost-savings potential
- Increased adaptability and scalability

With ooRexx, you get the user-friendliness of Rexx in an object-oriented environment.

ooRexx provides a Sockets API for Rexx. So you can script Rexx clients and servers, and run them in the Internet.

ooRexx also provides access to FTP commands by means of its RxFtp package, and the use of mathematical functions by means of its RxMath utility package. The Sockets, FTP, and mathematical functions packages are each supplied with separate, full documentation.

1.5. The Next Step

If you already know traditional Rexx and want to go straight to the basic concepts of object-oriented programming, continue with *Chapter 3, Into the Object World*.

If you are unfamiliar with traditional Rexx, continue to read *Chapter 2, A Quick Tour of Traditional Rexx*.

A Quick Tour of Traditional Rexx

Because this book is for Windows and Unix programmers, it is assumed that you are familiar with at least one other language. This chapter gives an overview of the basic Rexx rules and shows you in which respects Rexx is similar to, or different from, other languages you may already know.

2.1. What Is a Rexx Program?

A Rexx program is a text file, typically created using a text editor or a word processor that contains a list of instructions for your computer. Rexx programs are interpreted, which means the program is, like a batch file, processed line by line. Consequently, you do not have to compile and link Rexx programs. To run a Rexx program, all you need is Windows or Unix/Linux, the ooRexx interpreter, and the ASCII text file containing the program.

Rexx is similar to programming languages such as C, Pascal, or Basic. An important difference is that Rexx variables have no data type and are not declared. Instead, Rexx determines from context whether the variable is, for example, a string or a number. Moreover, a variable that was treated as a number in one instruction can be treated as a string in the next. Rexx keeps track of the variables for you. It allocates and deallocates memory as necessary.

Another important difference is that you can execute Windows, Unix/Linux commands and other applications from a Rexx program. This is similar to what you can do with a Windows Batch facility program or a Unix shell script. However, in addition to executing the command, you can also receive a return code from the command and use any displayed output in your Rexx program. For example, the output displayed by a DIR command can be intercepted by a Rexx program and used in subsequent processing.

Rexx can also direct commands to environments other than Windows. Some applications provide an environment to which Rexx can direct subcommands of the application. Or they also provide functions that can be called from a Rexx program. In these situations, Rexx acts as a scripting language for the application.

2.2. Running a Rexx Program

Rexx programs should have a file extension of .rex (the default searched for by the ooRexx interpreter). Here is a typical Rexx program named **`greeting.rex`**. It prompts the user to type in a name and then displays a personalized greeting:

Example 2.1. greeting.rex

```
/* greeting.rex - a Rexx program to display a greeting.  */
say "Please enter your name."    /* Display a message    */
pull name                        /* Read response        */
say "Hello" name                 /* Display greeting     */
exit 0                   /* Exit with a return code of 0 */
```

SAY is a Rexx instruction that displays a message (like PRINT in Basic or printf in C). The message to be displayed follows the SAY keyword. In this case, the message is the literal string "Please enter your name.". The data between the quotes is a constant and will appear exactly as typed. You can use either single (') or double quote (") delimiters for literal strings.

The PULL instruction reads a line of text from the standard input (the keyboard), and returns the text in the variable specified with the instruction. In our example, the text is returned in the variable name.

The next SAY instruction provides a glimpse of what can be done with Rexx strings. It displays the word **`Hello`** followed by the name of the user, which is stored in variable name. Rexx substitutes the value of name and displays the resulting string. You do not need a separate format string as you do with C or Basic.

The final instruction, EXIT, ends the Rexx program. Control returns to the operation system command prompt. EXIT can also return a value. In our example, **`0`** is returned. The EXIT instruction is optional. Running off the end of the program is equivalent to coding "EXIT **`0`**".

You can terminate a running Rexx program by pressing the Ctrl+Break key combination. Rexx stops running the program and control returns to the command prompt.

Rexx programs are often run from the command line, although, on the Windows operating systems there are several other options. These options are discussed several paragraphs later. The ooRexx interpreter is invoked by the command, **`rexx`**. With no arguments, the command produces a simple syntax reminder:

```
C:\rexx

Syntax is "rexx filename [arguments]"
or        "rexx -e program_string [arguments]"
or        "rexx -v".

C:\>
```

To run the program **`greeting.rex`**, for example, use the command

```
rexx greeting.rex
```

or

```
rexx greeting
```

If not provided, an extension of ".rex" is assumed.

The **`-v`** option produces the version and copyright information. For example:

```
Open Object Rexx Version 5.0.0 r11996
Build date: Mar  9 2020
Addressing mode: 32
Copyright (c) 1995, 2004 IBM Corporation. All rights reserved.
Copyright (c) 2005-2020 Rexx Language Association. All rights reserved.
This program and the accompanying materials are made available under the terms
of the Common Public License v1.0 which accompanies this distribution or at
http://www.oorexx.org/license.html
```

The **`-e`** accepts a complete Rexx program in the form of a single string and executes it immediately. Enclose the string in double quotes and separate lines of the program with semi-colons. Arguments can follow the string:

```
C:\>rexx -e "use arg name; say 'Hello to you' name" Mark
Hello to you Mark

C:\>
```

```
C:\>rexx -e "parse arg a b; say a '*' b 'is' a*b" 12 3
12 * 3 is 36

C:\>rexx -e "parse arg a b; say a '*' b 'is' a*b" 22 -1
22 * -1 is -22

C:\>rexx -e "parse arg a b; say a '*' b 'is' a*b" 126 456
126 * 456 is 57456

C:\>
```

On *Windows only* there are these additional ways to run your Rexx programs:

- The installation program on Windows sets up a file association for the **`.rex`** file extension. This association allows the ooRexx programs to be run from Windows Explorer by double-clicking on the icon of the program file. In addition, the program can be run from a command prompt in a console window by simply typing the file name. The **`.rex`** extension is not needed. For example, simply type **`greeting`** to execute the **`greeting.rex`** program:

  ```
  C:\>greeting
  Please enter your name.
  Mark
  Hello MARK

  C:\>
  ```

- A Rexx program can be run in *silent mode* by using **`rexxhide`**. This executes the program without creating a console window. This is most useful when creating a program shortcut. For the shortcut target, enter rexxhide.exe followed by the program name and the program arguments. Double-clicking on the shortcut then runs the program without creating a console window. **Note** that *silent* means there is no output from the Rexx program. When your program is run by **`rexxhide`**, either by double clicking on its icon, or from within a console window, there is no output displayed. Therefore **`rexxhide`** would not normally be used for programs like greeting.rex. This is what the greeting.rex program would look like when executed through rexxhide:

  ```
  C:\>rexxhide greeting.rex

  C:\>
  ```

- As a compliment to **`rexxhide`** is the **`rexxpaws`** program. When a Rexx program is executed through **`rexxpaws`**, at completion of the Rexx program, there will be a pause waiting for the user to hit the enter key. For example, using the greeting.rex program, **`rexxpaws`** would produce the following:

  ```
  C:\>rexxpaws greeting.rex
  Please enter your name.
  Mark
  Hello MARK

  Press ENTER key to exit...

  C:\>
  ```

  `rexxpaws` is useful for running a Rexx program from a shortcut, where the program *does* produce output. On Windows, when double-clicking on the program file icon, a console window opens, the program is run, and then the console window immediately closes. rexxpaws prevents the console window from closing until the user hits the enter key. This allows the user to see what output the program produced.

2.3. Elements of Rexx

Rexx programs are made up of clauses. Each clause is a complete Rexx instruction.

Rexx instructions include the obligatory program control verbs (IF, SELECT, DO, CALL, RETURN) as well as verbs that are unique to Rexx (such as PARSE, GUARD, and EXPOSE). In all, there are about 30 instructions. Many Rexx programs use only a small subset of the instructions.

A wide variety of built-in functions complements the instruction set. Many functions manipulate strings (such as SUBSTR, WORDS, POS, and SUBWORD). Other functions perform stream I/Os (such as CHARIN, CHAROUT, LINEIN, and LINEOUT). Still other functions perform data conversion (such as X2B, X2C, D2X, and C2D). A quick glance through the functions section of the *Open Object Rexx: Reference* gives you an idea of the scope of capabilities available to you.

The built-in functions are also available in Rexx implementations on other operating systems. In addition to these system-independent functions, Rexx includes a set of functions for working with Windows itself. These functions, known as the Rexx Utilities, let you work with resources managed by Windows or Linux, such as the display, the desktop, and the file system.

Instructions and functions are the building blocks of traditional Rexx programs. To convert Rexx into an object-oriented language, two more elements are needed: classes and methods. Classes and methods are covered in later chapters. This chapter continues with traditional building blocks of Rexx.

2.4. Writing Your Program

You can create Rexx programs using any editor that can create simple ASCII files without hidden format controls. Windows Notepad or Gnome gedit (on Linux) are a couple widely available editors.

Rexx is a free-format programming language. You can indent lines and insert blank lines for readability if you wish. But even free-format languages have some rules about how language elements are used. Rexx's rules center around its basic language element: the clause.

Usually, there is one clause on each line of the program, but you can put several and separate each clause with a semicolon (;):

```
say "Hello"; say "Goodbye"  /* Two clauses on one line */
```

To continue a clause on a second line, put a comma (,) or hyphen (-) at the end of the line:

```
say -             /* Continuation */
"It isn't so"
```

or

```
say ,            /* Continuation */
"It isn't so"
```

If you need to continue a literal string, do it like this:

```
say -              /* Continuation of literal strings */
"This is a long string that we want to continue" -
"on another line."
```

Rexx automatically adds a blank after **`continue`**. If you need to split a string, but do not want to have a blank inserted when Rexx puts the string back together, use the Rexx concatenation operator (||):

```
say "I do not want Rexx to in" || -   /* Continuation with concatenation */
"sert a blank!"
```

2.5. Testing Your Program

When writing your program, you can test statements as you go along using the rexxtry command from the Windows command prompt. rexxtry is a kind of Rexx mini-interpreter that checks Rexx statements one at a time. If you run rexxtry with no parameter, or with a question mark as a parameter, rexxtry also briefly describes itself.

From your command prompt type:

```
rexx rexxtry  /* on windows the case of the REXX is insignificant */
```

rexxtry describes itself and asks you for a Rexx statement to test. Enter your statement; rexxtry then runs it and returns any information available, or displays an error message if a problem is encountered. rexxtry remembers any previous statements you have entered during the session. To continue, just type the next line in your program and rexxtry will check it for you.

Enter an equal sign (=) to repeat your previous statement.

When you are done, type:

```
exit
```

and press Enter to leave rexxtry.

You can also enter a Rexx statement directly on the command line for immediate processing and exit:

```
rexx rexxtry call show
```

In this case, entering CALL SHOW displays the user variables provided by rexxtry.

2.6. Variables, Constants, and Literal Strings

Comprehensive rules for variables, constants, and literal strings are contained in the *Open Object Rexx: Reference*.

Rexx imposes few rules on variable names. A variable name can be up to 250 characters long, with the following restrictions:

- The first character must be **A-Z**, **a-z**, **!**, **?**, or _ .
- The rest of the characters may be **A-Z**, **a-z**, **!**, **?**, or _, **.**, or **0-9**.
- The period (**.**) has a special meaning for Rexx variables. Do not use it in a variable name until you understand the rules for forming compound symbols.

The variable name can be typed and queried in uppercase, mixed-case, or lowercase characters. A variable name in uppercase characters, for example, can also be queried in lowercase or mixed-case characters. Rexx translates lowercase letters in variables to uppercase before using them. Thus the variables names "*abc*", "*Abc*", and "*ABC*" all refer to the single variable "*ABC*". If you reference a variable name that has not yet been set, the name, in uppercase, is returned.

Literal strings in Rexx are delimited by quotation marks (either **'** or **"**). Examples of literal strings are:

```
'Hello'
"Final result:"
```

If you need to use quotation marks within a literal string, use quotation marks of the other type to delimit the string. For example:

```
"Don't panic"
'He said, "Bother"'
```

There is another way to do this. Within a literal string, a pair of quotation marks of the same type that starts the string is interpreted as a single character of that type. For example:

```
'Don''t panic'                (same as "Don't panic"      )
"He said, ""Bother"""         (same as 'He said, "Bother"')
```

2.7. Assignments

Assignments in Rexx usually take this form:

```
name = expression
```

For *name*, specify any valid variable name. For *expression*, specify the information to be stored, such as a number, a string, or a calculation to be performed. Here are some examples:

Example 2.2. Arithmetic

```
a=1+2
b=a*1.5
c="This is a string assignment. No memory allocation needed!"
```

The PARSE instruction and its variants PULL and ARG also assign values to variables. PARSE assigns data from various sources to one or more variables according to the rules of parsing. PARSE PULL, for example, is often used to read data from the keyboard:

Example 2.3. PARSE PULL

```
/* Using PARSE PULL to read the keyboard                              */
say "Enter your first name and last name"   /* prompt user            */
parse pull response       /* read keyboard and put result in RESPONSE */
say response              /* possibly displays "John Smith"           */
```

Other operands of PARSE indicate the source of the data. PARSE ARG, for example, retrieves command line arguments. PARSE VERSION retrieves the information about the version of the Rexx interpreter being used.

The most powerful feature of PARSE, however, is its ability to break up data using a template. The various pieces of data are assigned to variables that are part of the template. The following example prompts the user for a date, and assigns the month, day, and year to different variables. (In a real application, you would want to add instructions to verify the input.)

Example 2.4. PARSE PULL

```
/* PARSE example using a template */
say "Enter a date in the form MM/DD/YY"
parse pull month "/" day "/" year
say month
say day
say year
```

The template in this example contains two literal strings (`"/"`). The PARSE instruction uses these literals to determine how to split the data.

The PULL and ARG instructions are short forms of the PARSE instruction. See the *Open Object Rexx: Reference* for more information on Rexx parsing.

2.8. Using Functions

Rexx functions can be used in any expression. In the following example, the built-in function WORD is used to return the third blank-delimited word in a string:

Example 2.5. Rexx function use

```
/* Example of function use                                    */
myname="John Q. Public"   /* assign a literal string to MYNAME */
surname=word(myname,3)    /* assign WORD result to SURNAME     */
say surname               /* display Public                    */
```

Literal strings can be supplied as arguments to functions, so the previous program can be rewritten as follows:

Example 2.6. Rexx function use

```
/* Example of function use                                           */
surname=word("John Q. Public",3) /* assign WORD result to SURNAME    */
say surname                      /* display Public                   */
```

Because an expression can be used with the SAY instruction, you can further reduce the program to:

Example 2.7. Rexx expressions

```
/* Example of function use                                           */
say word("John Q. Public",3)
```

Functions can be nested. Suppose you want to display only the first two letters of the third word, Public. The LEFT function can return the first two letters, but you need to give it the third word. LEFT expects the input string as its first argument and the number of characters to return as its second argument:

Example 2.8. Rexx function use

```
/* Example of function use */

/* Here is how to do it without nesting */
thirdword=word("John Q. Public",3)
say left(thirdword,2)

/* And here is how to do it with nesting */
say left(word("John Q. Public",3),2)
```

2.9. Program Control

Rexx has instructions such as DO, LOOP, IF, and SELECT to control your program. Here is a typical Rexx IF instruction:

Example 2.9. IF instruction

```
if a>1 & b<0 then do
say "Whoops, A is greater than 1 while B is less than 0!"
say "I'm ending with a return code of 99."
exit 99
end
```

The Rexx relational operator & for a logical AND is different from the operator in C, which is &&. Other relational operators differ as well, so you may want to review the appropriate section in the *Open Object Rexx: Reference*.

Here is a list of some Rexx comparison operators and operations:

=	True if the terms are equal (numerically, when padded, and so on)
\=, ¬=	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 equal to
<=	Less than or equal to
==	True if terms are strictly equal (identical)
\==, ¬==	True if the terms are NOT strictly equal (inverse of ==)

Note

Throughout the language, the NOT character, **¬**, is synonymous with the backslash (****). You can use both characters. The backslash can appear in the **** (prefix not), **\=**, and **\==** operators.

A character string has the value >*false* if it is **0**, and *true* if it is **1**. (These values are sometimes called *logical* or *boolean* values and can be optionally expressed in programs with the ooRexx environment symbols **.false** which returns **0** and **.true** which returns **1**. Environment symbols in ooRexx start with a dot.)

A logical operator can take at least two values and return **0** or **1** as appropriate:

&	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**.

Note

On ASCII systems, Rexx recognizes the ASCII character encoding *124* as the logical OR character. Depending on the code page or keyboard you are using for your particular country, the logical OR character is shown as a solid vertical bar (**|**) or a split vertical bar (**¦**). The appearance of the character on your 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, make sure this character is ASCII character encoding *124*.

Using the wrong relational or comparison operator is a common mistake when switching between C and Rexx. The familiar C language braces { } are not used in Rexx for blocks of instructions. Instead, Rexx uses DO/END pairs. The THEN keyword is always required.

Here is an IF instruction with an ELSE:

Example 2.10. IF and ELSE instructions

```
if a>1 & b<0 then do
    say "Whoops, A is greater than 1 while B is less than 0!"
    say "I'm ending with a return code of 99."
    exit 99
end
else do
    say "A and B are okay."
    say "On with the rest of the program."
end  /* if */
```

You can omit the DO/END pairs if only one clause follows the THEN or ELSE keyword:

Example 2.11. IF and ELSE instructions

```
if words(myvar) > 5 then
    say "Variable MYVAR has more than five words."
else
    say "Variable MYVAR has fewer than six words."
```

Rexx also supports an ELSE IF construction:

Example 2.12. ELSE IF instruction

```
count=words(myvar)
if count > 5 then
    say "Variable MYVAR has more than five words."
else if count >3 then
    say "Variable MYVAR has more than three, but fewer than six words."
else
    say "Variable MYVAR has fewer than four words."
```

The SELECT instruction in Rexx is similar to the SELECT CASE statement in Basic and the switch statement in C. SELECT executes a block of statements based on the value of an expression. Rexx's SELECT differs from the equivalent statements in Basic and C in that the SELECT keyword is not followed by an expression. Instead, expressions are placed in WHEN clauses:

Example 2.13. SELECT instruction

```
select
when name="Bob" then
    say "It's Bob!"
when name="Mary" then
    say "Hello, Mary."
otherwise
end /* select */
```

WHEN clauses are evaluated sequentially. When one of the expressions is true, the statement, or block of statements, is executed. All the other blocks are skipped, even if their WHEN clauses would have evaluated to true. Notice that statements like the break statement in C are not needed.

The OTHERWISE keyword is used without an instruction following it. Rexx does not require an OTHERWISE clause. However, if none of the WHEN clauses evaluates to true and you omit OTHERWISE, an error occurs. Therefore, always include an OTHERWISE.

As with the IF instruction, you can use DO/END pairs for several clauses within SELECT cases. You do not need a DO/END pair if several clauses follow the OTHERWISE keyword:

Example 2.14. SELECT and OTHERWISE instructions

```
select
when name="Bob" then
    say "It's Bob"
when name="Mary" then do
    say "Hello Mary"
    marycount=marycount+1
    end
otherwise
    say "I'm sorry.  I don't know you."
    anonymous=anonymous+1
end /* select */
```

Many Basic implementations have several different instructions for loops. Rexx has the DO/END and LOOP/END pair. All of the traditional looping variations are incorporated into the DO and LOOP instructions, which can be used interchangeably for looping:

Example 2.15. DO and LOOP instructions

```
do i=1 to 10         /* Simple loop            */
   say i
end

do i=1 to 10 by 2    /* Increment count by two */
   say i
end

b=3; a=0             /* DO WHILE - the conditional expression  */
do while a<b         /* is evaluated before the instructions   */
   say a             /* in the loop are executed.  If the      */
   a=a+1             /* expression isn't true at the outset,   */
end                  /* instructions are not executed at all.  */

a=5                  /* DO UNTIL - like many other languages,  */
b=4                  /* a Rexx DO UNTIL block is executed at   */
do until a>b         /* least once.  The expression is         */
   say "Until loop"  /* evaluated at the end of the loop.      */
end
```

or, using LOOP

```
loop i=1 to 10       /* Simple loop            */
   say i
end

loop i=1 to 10 by 2  /* Increment count by two */
   say i
end

b=3; a=0             /* LOOP WHILE - the conditional expression*/
loop while a<b    /* is evaluated before the instructions   */
   say a             /* in the loop are executed.  If the      */
   a=a+1             /* expression isn't true at the outset,   */
end                  /* instructions are not executed at all.  */

a=5                  /* LOOP UNTIL - like many other languages,*/
b=4                  /* a Rexx LOOP UNTIL block is executed at */
do until a>b         /* least once.  The expression is         */
   say "Until loop"  /* evaluated at the end of the loop.      */
end
```

Rexx also has a FOREVER keyword. Use the LEAVE, RETURN, or EXIT instructions to break out of the loop:

Example 2.16. DO FOREVER instruction

```
                  /* Program to emulate your five-year-old child */
num=random(1,10)  /* To emulate a three-year-old, move this inside the loop! */
do forever
  say "What number from 1 to 10 am I thinking of?"
  pull guess
  if guess=num then do
     say "That's correct"
     leave
  end
  say "No, guess again..."
end
```

Rexx also includes an ITERATE instruction, which skips the rest of the instructions in that iteration of the loop:

Example 2.17. ITERATE instruction

```
loop i=1 to 100
   /* Iterate when the "special case" value is reached    */
   if i=5 then iterate

   /* Instructions used for all other cases would be here */

end
```

You can use loops in IF or SELECT instructions:

Example 2.18. ITERATE instruction

```
/* Say hello ten times if I is equal to 1 */
if i=1 then
   loop j=1 to 10
      say "Hello!"
   end
```

There is an equivalent to the Basic GOTO statement: the Rexx SIGNAL instruction. SIGNAL causes control to branch to a label:

Example 2.19. SIGNAL instruction

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

As with GOTO, you need to be careful about how you use SIGNAL. In particular, do not use SIGNAL to jump to the middle of a DO/END block or into a SELECT structure.

2.10. Subroutines and Procedures

In Rexx you can write routines that make all variables accessible to the called routine. You can also write routines that hide the caller's variables.

The following shows an example of a routine in which all variables are accessible:

Example 2.20. ROUTINE instruction

```
/* Routine example                        */
i=10              /* Initialize I      */
call myroutine    /* Call routine      */
say i             /* Displays 22       */
exit              /* End main program */

myroutine:        /* Label             */
i=i+12            /* Increment I       */
```

```
return
```

The CALL instruction calls routine *MYROUTINE*. A label (note the colon) marks the start of the routine. A RETURN instruction ends the routine. Notice that an EXIT instruction is required in this case to end the main program. If EXIT is omitted, Rexx assumes that the following instructions are part of your main program and will execute those instructions. The SAY instruction displays 22 instead of 10 because the caller's variables are accessible to the routine.

You can return a result to the caller by placing an expression in the RETURN instruction, like this:

Example 2.21. RETURN instruction

```
/* Routine with result                */
i=10               /* Initialize I     */
call myroutine     /* Call routine     */
say result         /* Displays 22      */
exit               /* End main program */

myroutine:         /* Label            */
return i+12        /* Increment I      */
```

The returned result is available to the caller in the special variable *RESULT*, as previously shown. If your routine returns a result, you can call it as a function:

Example 2.22. RESULT special variable

```
/* Routine with result called as function  */
i=10               /* Initialize I         */
say myroutine()    /* Displays 22          */
exit               /* End main program     */

myroutine:         /* Label                */
return i+12        /* Increment I          */
```

You can pass arguments to this sort of routine, but all variables are available to the routine anyway.

You can also write routines that separate the caller's variables from the routine's variables. This eliminates the risk of accidentally writing over a variable used by the caller or by some other unprotected routine. To get protection, use the PROCEDURE instruction, as follows:

Example 2.23. PROCEDURE instruction

```
/* Routine example using PROCEDURE instruction                     */
headcount=0
tailcount=0
/* Toss a coin 100 times, report results */
do i=1 to 100
   call cointoss                                 /* Flip the coin      */
   if result="HEADS" then headcount=headcount+1   /* Increment counters */
   else tailcount=tailcount+1
                                                 /* Report results     */
say "Toss is" result ||".  Heads=" headcount  "Tails=" tailcount
end /* do */
exit                                             /* End main program   */

cointoss: procedure             /* Use PROCEDURE to protect caller         */
```

```
   i=random(1,2)                /* Pick a random number: 1 or 2             */
   if i=1 then return "HEADS"   /* Return English string                    */
return "TAILS"
```

In this example, the variable i is used in both the main program and the routine. When the PROCEDURE instruction is placed after the routine label, the routine's variables become local variables. They are isolated from all other variables in the program. Without the PROCEDURE instruction, the program would loop indefinitely. On each iteration the value of **`i`** would be reset to some value less than 100, which means the loop would never end. If a programming error causes your procedure to loop indefinitely, use the Ctrl+C key combination or close the command window to end the procedure.

To access variables outside the routine, add an EXPOSE operand to the PROCEDURE instruction. List the desired variables after the EXPOSE keyword:

Example 2.24. EXPOSE keyword

```
/* Routine example using PROCEDURE instruction with EXPOSE operand         */
headcount=0
tailcount=0
/* Toss a coin 100 times, report results                                   */
do i=1 to 100
   call cointoss                                   /* Flip the coin      */
   say "Toss is" result ||".  Heads=" headcount  "Tails=" tailcount
end /* do */
exit                                               /* End main program   */

cointoss: procedure expose headcount tailcount /* Expose the counter variables */
   if random(1,2)=1 then do                    /* Pick a random number: 1 or 2 */
      headcount=headcount+1                    /* Bump counter...              */
      return "HEADS"                           /* ...and return English string */
   end
   else
      tailcount=tailcount+1
return "TAILS"
```

To pass arguments to a routine, separate the arguments with commas:

Example 2.25. Passing arguments

```
call myroutine arg1, "literal arg", arg3   /* Call as subroutine */
myrc=myroutine(arg1, "literal arg", arg3)  /* Call as function   */
```

In the routine, use the USE ARG instruction to retrieve the argument.

Into the Object World

Open Object Rexx includes features typical of an object-oriented language—features like data encapsulation, inheritance and polymorphism. Open Object Rexx is an extension of the traditional Rexx language, which has been expanded to include objects, classes and methods. These extensions do not replace traditional Rexx functions, or preclude the development or running of traditional Rexx programs. You can program as before, program with objects, or mix objects with regular Rexx instructions. The Rexx programming concepts that support the object-oriented features are described in this chapter.

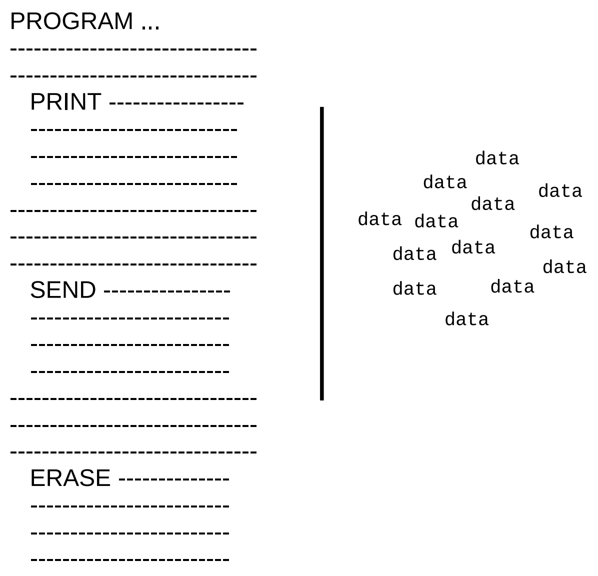
3.1. What Is Object-Oriented Programming?

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. Often several actions apply; you could also send or erase the report.

3.2. 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. These actions are independent of the data they operate on.



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.

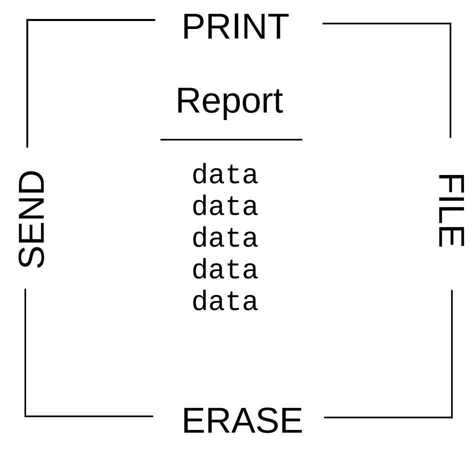


Figure 3.1. Modular Data—a Report Object

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.

3.3. 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:

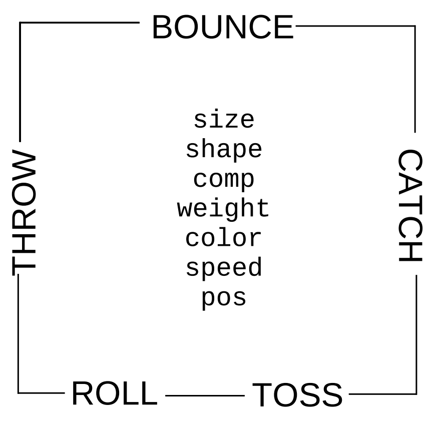


Figure 3.2. A Ball Object

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 variable name, and an associated data value that can change over time. These actions and characteristics are so closely associated that they cannot be separated:

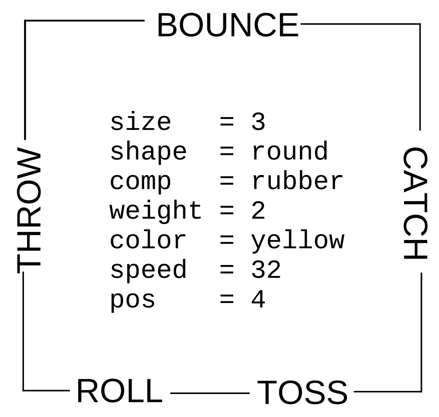


Figure 3.3. Ball Object with Variable Names and Values

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. Variables are associated with an object and exist for the lifetime of that object:

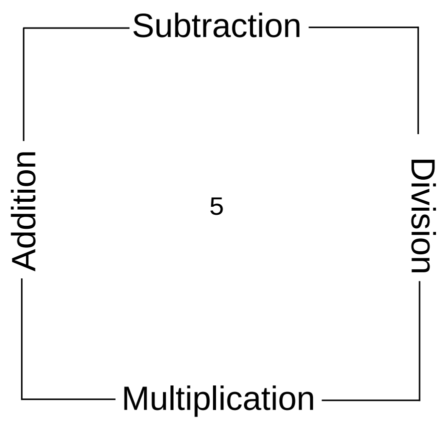


Figure 3.4. Encapsulated 5 Object

3.3.1. 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. Actions are invoked by sending messages. Objects respond to these messages by invoking methods (*Section 3.3.2, "Methods"*) that perform an action, return data, or both. A message to an object must specify:

- A receiving object

- The "message send" operator **~**, which is called the *twiddle*
- The name of the action and, optionally in parentheses, any parameters required

So the message format looks like this:

```
object~action(parameters)
```

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

```
"!iH"~reverse
```

returns the string object **`Hi!`.**

3.3.2. Methods

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

3.3.3. Polymorphism

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

```
"!iH"~reverse   /* Reverses the characters "!iH" to form "Hi!"  */
pen~reverse     /* Reverses the direction of a plotter pen      */
ball~reverse    /* Reverses the direction of a moving ball      */
```

As long as each object has its own REVERSE method, REVERSE runs even if the programming implementation is different for each object. This ability to hide different functions behind a common interface is called polymorphism. As a result of information hiding, each object in the previous example 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. The ability to reuse the same method name so that one message can initiate more than one function is another feature of polymorphism. 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.

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

Polymorphism involves a form of contract between two objects. One object will send a message to another object expecting a particular result. Different types of objects can implement different versions of this message as long as it fulfills the expectations of the invoking object. For example, the LOOP instruction has a form called OVER. Loop OVER will iterate over a number of elements provided by an object. For example,

Example 3.1. LOOP OVER

```
myarray = .array~of("Rick", "David", "Mark")
loop name over myarray
   say name
end
```

Will display the strings **`"Rick"`**, **`"David"`**, and **`"Mark"`**, in that sequence. The LOOP OVER instruction will work with Array, List, Stem, Streams variables, etc. The LOOP instruction itself does not know anything about Arrays or Lists or Stems or Streams. The LOOP instruction specifies a contract. LOOP will send the target object the message MAKEARRAY and expects the target object to return an Array object that is used for the LOOP iteration. Any object can participate in LOOP iteration by implementing this contract. Objects that do implement the MAKEARRAY contract are polymorphic with the LOOP instruction.

3.3.4. 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. All Icon objects will share the actions and characteristics defined by the class:

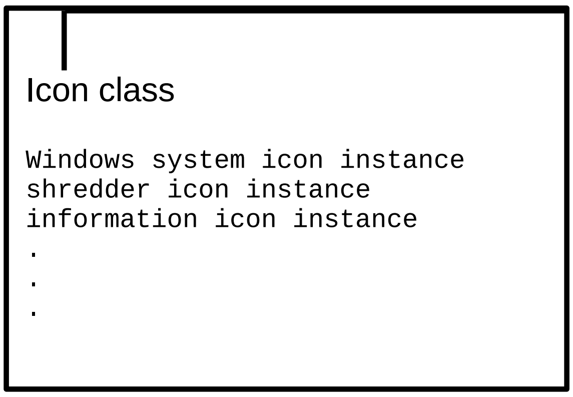


Figure 3.5. A Simple Class

All the icon objects will use common methods like DRAW or ERASE. They will have 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"`**

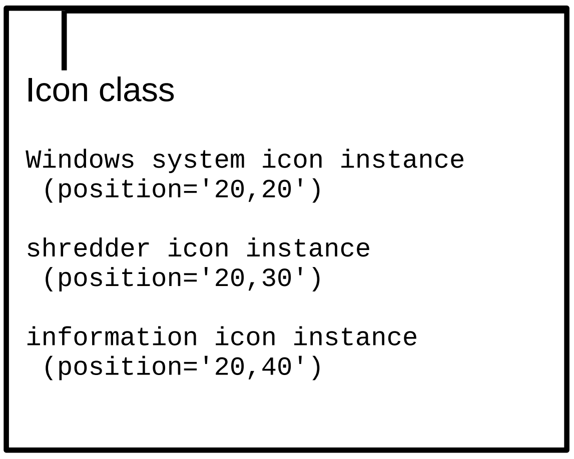


Figure 3.6. Icon Class

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 the 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.

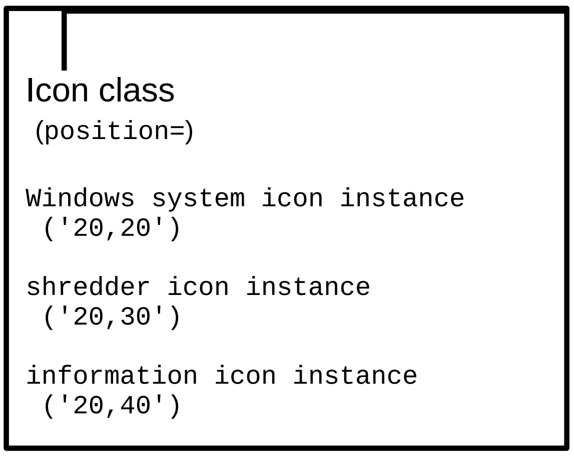


Figure 3.7. Instances of the Icon Class

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.

3.3.5. 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.

3.3.6. 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 of your own, according to your needs.

Rexx classes are hierarchical. The original class is called a base class or a superclass. The derived class is called a subclass. Subclasses inherit methods and data from one or more superclasses. A subclass can introduce new methods and data, and can override methods from the superclass.

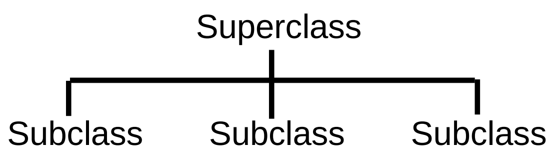


Figure 3.8. Superclass and Subclasses

You can add a class to an existing superclass. For example, you might add the Icon class to the Screen-Object superclass:

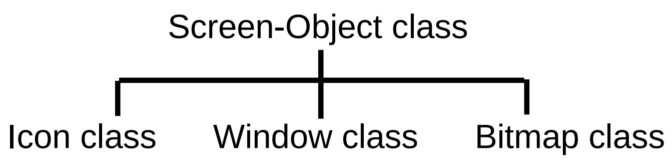


Figure 3.9. The Screen-Object Superclass

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:

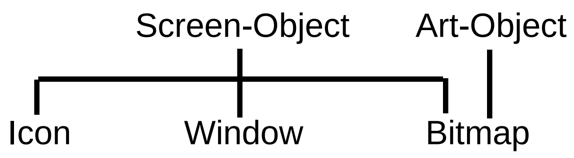


Figure 3.10. Multiple Inheritance

The Basics of Classes

Similar objects in Rexx are grouped into classes, forming a hierarchy. Rexx gives you a basic class hierarchy to start with. All of the classes in the hierarchy are described in detail in the *Open Object Rexx: Reference*.

Rexx provides the following classes on all operating systems by default. The classes are depicted in hierarchical order, subclasses get indented under their superclass. If a class can be used for multiple inheritance it gets flagged with *(mixin)*. Classes that inherit from mixin classes will list them in parentheses starting with the word *inherit* followed by the blank delimited list of inherited mixin classes.

- *Object*
  - *Alarm*
  - *AlarmNotification* *NEW* (*mixin*)
  - *Array (inherit OrderedCollection)*
  - *Bag (inherit MapCollection SetCollection)*
  - *Buffer*
  - *Class*
  - *Collection* (*mixin*)
    - *MapCollection* (*mixin*)
    - *OrderedCollection* (*mixin*)
    - *SetCollection* (*mixin*)
  - *Comparable* (*mixin*)
  - *Comparator* (*mixin*)
    - *CaselessColumnComparator* (*mixin*)
    - *CaselessComparator* (*mixin*)
    - *CaselessDescendingComparator* (*mixin*)
    - *ColumnComparator* (*mixin*)
    - *DescendingComparator* (*mixin*)
    - *InvertingComparator* (*mixin*)
    - *NumericComparator* (*mixin*)
  - *DateTime (inherit Comparable Orderable)*
  - *Directory (inherit MapCollection)*
    - *Properties*
  - *EventSemaphore* *NEW*
  - *File (inherit Comparable Orderable)*
  - *IdentityTable (inherit MapCollection)*
  - *InputOutputStream* (*mixin*) *(inherit InputStream OutputStream)*
    - *Stream* (*mixin*)
  - *InputStream* (*mixin*)
  - *List (inherit OrderedCollection)*
  - *Message (inherit MessageNotification AlarmNotification)*
  - *MessageNotification* *NEW* (*mixin*)
  - *Method*
  - *Monitor*
  - *MutableBuffer*
  - *MutexSemaphore* *NEW*
  - *Orderable* (*mixin*)
  - *OutputStream* (*mixin*)
  - *Package*
  - *Pointer*
  - *Queue (inherit OrderedCollection)*
    - *CircularQueue*
  - *Relation (inherit MapCollection)*

RexxContext
RexxInfo *NEW*
RexxQueue
Routine
Set (inherit MapCollection SetCollection)
Singleton *NEW*
StackFrame
Stem (inherit MapCollection)
String (inherit Comparable)
StringTable *NEW* *(inherit MapCollection)*
Supplier
    StreamSupplier
Table (inherit MapCollection)
Ticker *NEW*
TimeSpan (inherit Comparable Orderable)
Validate *NEW*
VariableReference *NEW*
WeakReference

Note that there might also be other classes available, depending on the operating system.

4.1. Rexx Classes for Programming

The classes Rexx supplies provide the starting point for object-oriented programming. Some key classes that you are likely to work with are described in the following sections.

4.1.1. The Alarm Class

The Alarm class is used to create objects with timing and notification capability. An alarm object is able to send a message to an object at any time in the future, and until then, you can cancel the alarm.

The class Alarm subclasses (specializes) *Object*.

4.1.2. *NEW* The AlarmNotification Class

Implements the notification interface for the *Alarm* class.

The class AlarmNotification subclasses (specializes) *Object* and is a *mixin* class (it can be inherited by other classes). The following class inherits (specializes) this mixin class:

- *Message*

4.1.3. The Array Class

A sequenced collection of objects ordered by whole-number indexes.

The class Array subclasses (specializes) *Object* and inherits (specializes) in addition:

- *OrderedCollection*

4.1.4. The Bag Class

A collection where the index is equal to the value. Bag indexes can be any object (as with the Table class) and each index can appear more than once.

The class Bag subclasses (specializes) *Object* and inherits (specializes) in addition in the following order:

- *MapCollection*
- *SetCollection*

4.1.5. The 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 class Buffer subclasses (specializes) *Object*.

4.1.6. The CaselessColumnComparator Class

The CaselessColumnComparator class performs caseless orderings of specific substrings of String objects.

The class CaselessColumnComparator is a *mixin* class (it can be inherited by other classes), it subclasses (specializes) *Comparator*.

4.1.7. The CaselessComparator Class

The CaselessComparator class performs caseless orderings of String objects.

The class CaselessComparator is a *mixin* class (it can be inherited by other classes), it subclasses (specializes) *Comparator*.

4.1.8. The CaselessDescendingComparator Class

The CaselessDescendingComparator class performs caseless string sort orderings in descending order. This is the inverse of a CaselessComparator sort order.

The class CaselessDescendingComparator is a *mixin* class (it can be inherited by other classes), it subclasses (specializes) *Comparator*.

4.1.9. The 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 class CircularQueue subclasses (specializes) *Queue*.

4.1.10. The Collection Classes

The collection classes are used to manipulate collections of objects. A *collection* is an object that contains a number of *items*, which can be any objects. These manipulations might include counting objects, organizing them, or assigning them a supplier (for example, to indicate that a specific assortment of baked goods is supplied by the Pie-by-Night Bakery).

Rexx includes classes, for example, for arrays, lists, queues, tables, and directories. Each item stored in a Rexx *collection* has an associated index that you can use to retrieve the item from the *collection* with the AT or [] (left and right bracket) methods, and each collection defines its own acceptable index types:

The class Collection is a *mixin* class (it can be inherited by other classes), it subclasses (specializes) *Object*.

4.1.11. The ColumnComparator Class

The ColumnComparator class performs orderings based on specific substrings of String objects.

The class ColumnComparator is a *mixin* class (it can be inherited by other classes), it subclasses (specializes) *Comparator*.

4.1.12. The Comparable Class

Any object that inherits the Comparable mixin class and implements a compareTo() method can be sorted. The DateTime Class and TimeSpan Class are examples of built-in Rexx classes that can be sorted. Any user created class may also implement a compareTo() method to enable sorting.

The class Comparable subclasses (specializes) *Object* and is a *mixin* class (it can be inherited by other classes). The following classes inherit (specialize) this class:

- *DateTime*
- *File*
- *String*
- *TimeSpan*

4.1.13. The Comparator Class

The Comparator class is the base class for implementing Comparator objects that can be used with the Array sortWith() or stableSortWith() method. 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.

The class Comparator is a *mixin* class (it can be inherited by other classes), it subclasses (specializes) *Object*.

4.1.14. The DateTime Class

A DateTime object represents a point in between *1 January 0001 at 00:00.000000* and *31 December 9999 at 23:59:59.999999*. A DateTime object has methods to allow formatting a date or time in various formats, as well as allowing arithmetic operations between dates.

The class DateTime subclasses (specializes) *Object* and inherits (specializes) in addition in the following order:

- *Comparable*
- *Orderable*

4.1.15. The DescendingComparator Class

The DescendingComparator class performs sort orderings in descending order. This is the inverse of a Comparator sort order.

The class DescendingComparator is a *mixin* class (it can be inherited by other classes), it subclasses (specializes) *Comparator*.

4.1.16. The Directory Class

A collection of character string indexes. Indexes are compared using the string == comparison method to test for strict equality.

The class Directory subclasses (specializes) *Object* and inherits (specializes) in addition:

- *MapCollection*

4.1.17. The EventSemaphore Class

An EventSemaphore implements a synchronization mechanism that can be used to indicate to multithreaded activities when a particular condition—the event—has become **`.true`**.

The class EventSemaphore subclasses (specializes) *Object*.

4.1.18. The File Class

The File class provides services which are common to all the filesystems supported by ooRexx. A File object represents a path to a file or directory. The path can be relative or absolute.

The class File subclasses (specializes) *Object* and inherits (specializes) in addition in the following order:

- *Comparable*
- *Orderable*

4.1.19. The 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 in an identity table are made using an object identity comparison. That is, an index will only match if the same instance is used in the collection.

The class IdentityTable subclasses (specializes) *Object* and inherits (specializes) in addition:

- *MapCollection*

4.1.20. The 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.

The class InputOutputStream is a *mixin* class (it can be inherited by other classes), it subclasses (specializes) *Object* and inherits (specializes) in addition in the following order:

- *InputStream*
- *OutputStream*

4.1.21. The 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.

The class InputStream subclasses (specializes) *Object* and is a *mixin* class (it can be inherited by other classes). The following class inherits (specializes) this mixin class:

- *InputOutputStream*

4.1.22. The InvertingComparator Class

The InvertingComparator class inverts the comparison results of another Comparator object to reverse the resulting sort order.

The class InvertingComparator is a *mixin* class (it can be inherited by other classes), it subclasses (specializes) *Comparator*.

4.1.23. The 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.

The class List subclasses (specializes) *Object* and inherits (specializes) in addition:

- *OrderedCollection*

4.1.24. The MapCollection Class

The MapCollection class defines the basic set of methods implemented by all collections that create a mapping from a programmer defined *index* object to a *value*..

The class MapCollection subclasses (specializes) *Collection* and is a *mixin* class (it can be inherited by other classes). The following classes inherit (specialize) this class:

- *Bag*
- *Directory*
- *IdentityTable*
- *Relation*

- *Set*
- *Stem*
- *StringTable*
- *Table*

4.1.25. The Message Class

Message objects allow you to run concurrently methods on other threads or to invoke dynamically calculated messages. Methods of for this class are used, for example, to start a message on another thread, to notify the sender object when an error occurs or when message processing is complete, or to return the results of that processing to the sender or to some other object.

The class Message subclasses (specializes) *Object* and inherits (specializes) in addition in the following order:

- *MessageNotification*
- *AlarmNotification*

4.1.26. *NEW* The MessageNotification Class

Implements the notification interface for the *Message*.

The class MessageNotification subclasses (specializes) *Object* and is a *mixin* class (it can be inherited by other classes). The following class inherits (specializes) this mixin class:

- *Message*

4.1.27. The Method Class

The Method class creates method objects from Rexx source code.

The class Method subclasses (specializes) *Object*.

4.1.28. The Monitor Class

The Monitor class provides a way to forward messages to a specified destination. The Monitor creates a proxy that can dynamically route messages to different destinations. Monitor methods change or restore a destination object.

The class Monitor subclasses (specializes) *Object*.

4.1.29. The 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.

The class MutableBuffer subclasses (specializes) *Object*.

4.1.30. The MutexSemaphore Class

A MutexSemaphore, or a mutual exclusion semaphore is a synchronization mechanism which concurrent activities can use to control access to a common resource.

The class MutexSemaphore subclasses (specializes) *Object*.

4.1.31. The NumericComparator Class

The NumericComparator class compares strings using numeric comparison rules to determine sort order.

The class NumericComparator is a *mixin* class (it can be inherited by other classes), it subclasses (specializes) *Comparator*.

4.1.32. The 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. This class is defined as a mixin class.

The class Orderable subclasses (specializes) *Object* and is a *mixin* class (it can be inherited by other classes). The following classes inherit (specialize) this class:

- *DateTime*
- *File*
- *TimeSpan*

4.1.33. The OrderedCollection Class

The OrderedCollection class defines the basic set of methods implemented by all collections that have an inherent *index* ordering.

The class OrderedCollection subclasses (specializes) *Collection* and is a *mixin* class (it can be inherited by other classes). The following classes inherit (specialize) this class:

- *Array*
- *List*
- *Queue*

4.1.34. The 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.

The class OutputStream subclasses (specializes) *Object* and is a *mixin* class (it can be inherited by other classes). The following class inherits (specializes) this mixin class:

- *InputOutputStream*

4.1.35. The 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.

The class Package subclasses (specializes) *Object*.

4.1.36. The 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.

The class Pointer subclasses (specializes) *Object*.

4.1.37. The 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.

The class Properties subclasses (specializes) *Directory*.

4.1.38. The Queue Class

A sequenced collection of 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.

The class Queue subclasses (specializes) *Object* and inherits (specializes) in addition:

- *OrderedCollection*

4.1.39. The Relation Class

A collection of indexes that can be any object (as with the Table class). A relation can contain duplicate indexes.

The class Relation subclasses (specializes) *Object* and inherits (specializes) in addition:

- *MapCollection*

4.1.40. The 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.

The class RexxContext subclasses (specializes) *Object*.

4.1.41. *NEW* The RexxInfo Class

The RexxInfo class gives access to Rexx language information and other platform-specific information in a single place. its single instance can be obtained via the **`.REXXINFO`** environment symbol.

The class RexxInfo subclasses (specializes) *Object*, it can only be used by the interpreter.

4.1.42. The RexxQueue Class

The RexxQueue class provides object-style access to Rexx external data queues.

The class RexxQueue subclasses (specializes) *Object*.

4.1.43. The Routine Class

The Routine class creates routine objects from Rexx source code.

The class Routine subclasses (specializes) *Object*.

4.1.44. The Set Class

A collection where the indexes are equal to the values. Set indexes can be any object (as with the Table class) and each index is unique.

The class Set subclasses (specializes) *Object* and inherits (specializes) in addition in the following order:

- *MapCollection*
- *SetCollection*

4.1.45. The SetCollection Class

The SetCollection class defines no methods of its own, but serves as a collection type for collections that constrain the *index* and *item* to be be the same object.

The class SetCollection subclasses (specializes) *Collection* and is a *mixin* class (it can be inherited by other classes). The following classes inherit (specialize) this class:

- *Bag*
- *Set*

4.1.46. *NEW* The Singleton Class (Metaclass)

The Singleton class allows one to restrict the number of instances that can be created from a class to a single instance, the *singleton*.

The class Singleton subclasses (specializes) *Class*.

4.1.47. The StackFrame Class

The StackFrame class provides debugging information about a Rexx activity's execution chain.

The class StackFrame subclasses (specializes) *Object*.

4.1.48. The Stem Class

A Stem object is a collection with unique indexes that are character strings.

A stem variable is a symbol that must start with a letter and end with a period, like "**`FRED.`**" or "**`A.`**". The value of a stem variable is a Stem object. A Stem object is a collection of unique character string indexes. Stem objects are automatically created when a Rexx Stem variable or Rexx compound variable is used. In addition to the items assigned to the collection indexes, a Stem object also has a default value that is used for all uninitialized indexes of the collection.

The class Stem subclasses (specializes) *Object* and inherits (specializes) in addition:

- *MapCollection*

4.1.49. The Stream Class

Input and output Streams let Rexx communicate with external objects, such as people, files, queues, serial interfaces, displays, and networks. In programming there are many stream actions that can be coded as methods for manipulating the various Stream objects. These methods and objects are organized in the Stream class.

The methods are used to open streams for reading or writing, close streams at the end of an operation, move the line-read or line-write position within a file stream, or get information about a stream. Methods are also provided to get character strings from a stream or send them to a stream, count characters in a stream, flush buffered data to a stream, query path specifications, time stamps, size, and other information from a stream, or do any other I/O stream manipulation (see *Chapter 7, Input and Output* for examples).

The class Stream is a *mixin* class (it can be inherited by other classes), it subclasses (specializes) *InputOutputStream*.

4.1.50. The StreamSupplier Class

A subclass of the Supplier class that will provide 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.

The class StreamSupplier subclasses (specializes) *Supplier*.

4.1.51. The String Class

Strings are data values that can have any length and contain any characters. They are subject to logical operations like AND, OR, exclusive OR, and logical NOT. Strings can be concatenated, copied, reversed, joined, and split. When Strings are numeric, there is the need to perform arithmetic operations on them or find their absolute value or convert them from binary to hexadecimal, and vice versa. All this and more can be accomplished using the String class of objects.

The class String subclasses (specializes) *Object* and inherits (specializes) in addition:

- *Comparable*

4.1.52. The StringTable Class

A collection using unique character string indexes. The items of a StringTable can be any valid Rexx object.

The class StringTable subclasses (specializes) *Object* and inherits (specializes) in addition:

- *MapCollection*

4.1.53. The Supplier Class

All collections have suppliers. The Supplier class is used to enumerate items that a collection contained when the supplier was created. The supplier gives access to each *index/value* pair stored in the collection as a sequence.

The class Supplier subclasses (specializes) *Object*.

4.1.54. The Table Class

A collection of indexes that can be any object. For example, String objects, Array objects, Alarm objects, or any user-created object can be a table index. The Table class determines an index match by using the == comparison method to test for strict equality. A Table contains no duplicate indexes.

The class Table subclasses (specializes) *Object* and inherits (specializes) in addition:

- *MapCollection*

4.1.55. *NEW* Ticker Class

Provides a repeating notification capability by sending a notification message to a notification target each trigger interval.

The class Ticker subclasses (specializes) *Object*.

4.1.56. The TimeSpan Class

A TimeSpan object represents a point in between *1 January 0001 at 00:00.000000* and *31 December 9999 at 23:59:59.999999*. A TimeSpan object has methods to allow formatting a date or time in various formats, as well as allowing arithmetic operations between dates.

The class TimeSpan subclasses (specializes) *Object* and inherits (specializes) in addition in the following order:

- *Comparable*
- *Orderable*

4.1.57. *NEW* Validate Class

A class that provides helper methods to validate arguments being of correct class, logical or numeric type, or within a numeric range.

The class Validate subclasses (specializes) *Object*.

4.1.58. *NEW* VariableReference Class

An object that maintains a reference to a variable, its name and its value.

The class VariableReference subclasses (specializes) *Object*.

4.1.59. The 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.

The class WeakReference subclasses (specializes) *Object*.

4.2. Rexx Classes for Organizing Objects

Rexx provides several key classes that form the basis for building class hierarchies.

4.2.1. The Object Class

Because the root class in the hierarchy is the Object class, everything below it is an object. To interact with each other, objects require their own actions, called methods. These methods, which encode actions that are needed by all objects, belong to the Object class.

Every other class in the hierarchy inherits the methods of the root class. Inheritance is the handing down of methods from a "parent" class—called a superclass—to all of its "descendent" classes—called subclasses. Finally, instances acquire methods from their own classes. Any method created for the Object class is automatically made available to every other class in the hierarchy.

4.2.2. The Class Class (Metaclass)

The Class class is used for generating new classes. If a class is like a factory for producing instances, Class is like a factory for producing factories. Class is the parent of every new class in the hierarchy, and these all inherit Class-like characteristics. Class-like characteristics are methods and related variables, which reside in Class, to be used by all classes.

The class Class subclasses (specializes) *Object*.

A class that can be used to create another class is called a *metaclass* (*Section 4.5.4, “Metaclasses”*). The Class class is unique among Rexx classes in that it is the only metaclass that Rexx provides. As such, the Class's methods not only make new classes, they make methods for use by the new class and its instances. They also make methods that only the new class itself can use, but not its instances. These are called class methods. They give a new class some power that is denied to its instances.

Because each instance of Class is another class, that class inherits the Class's instance methods as class methods. Thus if Class generates a Pizza factory instance, the factory-running actions (Class's instance methods) become the class methods of the Pizza factory. Factory operations are class methods, and any new methods created to manipulate pizzas would be instance methods.

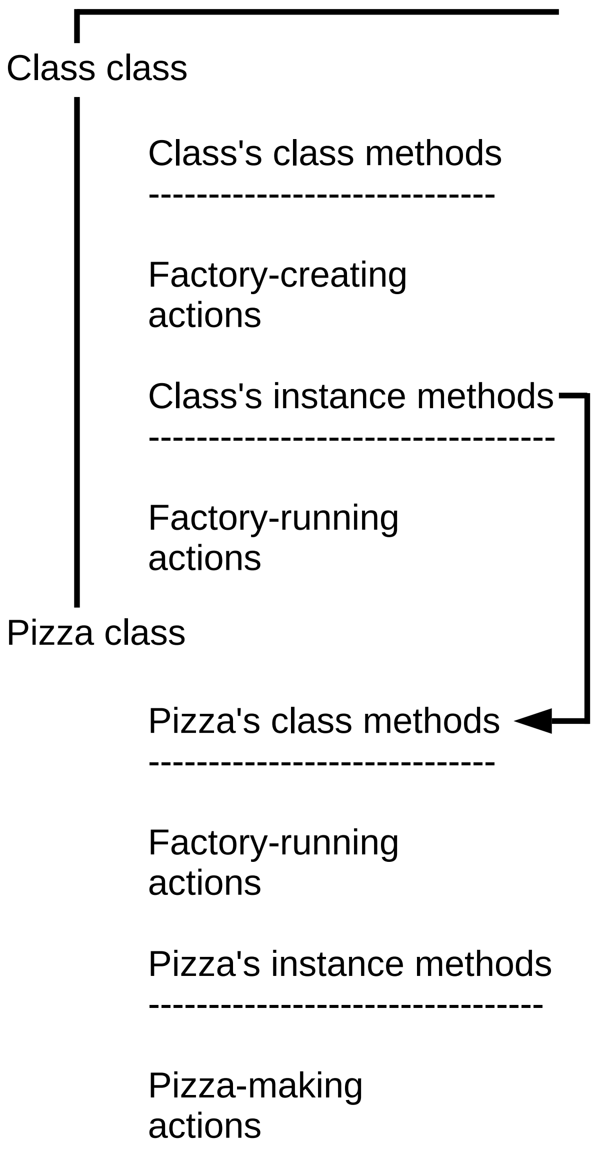


Figure 4.1. How Subclasses Inherit Instance Methods from the Class Class

As a programmer, you typically create classes by using directives, rather than the methods of the Class class. In particular, you'll use the ::CLASS directive, described later in this section. The ::CLASS directive is a kind of Rexx clause that declares class definitions in a simple, static form in your programs.

4.3. Creating Your Own Classes Using Directives

By analyzing your problem in terms of objects, you can determine what classes need to be created. You can create a class using messages or directives. Directives are a new kind of Rexx clause, and they are preferred over messages because the code is easier to read and understand, especially in large programs. They also provide an easy way for you to share your class definitions with others using the PUBLIC option.

4.3.1. What Are Directives?

A Rexx program is made up of one or more executable units. Directives separate these units, which themselves are Rexx programs. Rexx processes all directives first to set up any classes, methods, or routines needed by the program. Then it runs any code that exists before the first directive. The first directive in a program marks the end of the executable part of the program. A directive is a kind of clause that begins with a double-colon (::) and is non-executable (a directive cannot appear in the expression of an INTERPRET instruction, for example).

4.3.2. The Directives Rexx Provides

The following is a short summary of all the Rexx directives. See the *Open Object Rexx: Reference* for more details on, or examples of, any of these Rexx directives.

4.3.2.1. The ::CLASS Directive

You use the ::CLASS directive to create a class. Programs can then use the new class by specifying it as a Rexx environment symbol (the class name preceded by a period) in the program. For example, in *Section 4.3.4, “A Sample Program Using Directives”*, the Savings class is created using the ::CLASS directive. A program can then use the new class by specifying it as an environment symbol, "**`.savings`**".

The new class that you create acquires any methods defined by subsequent ::METHOD directives within the program, until either another ::CLASS directive or the end of the program is reached.

You can use the ::CLASS directive's SUBCLASS option to make the new class the subclass of another. In *Section 4.3.4, “A Sample Program Using Directives”*, the Savings class is made a subclass of the Account class. A subclass inherits instance and class methods from its specified superclass; in the sample, Savings inherits from Account.

Additional ::CLASS directive options are available for:

- Inheriting instance methods from a specified metaclass as class methods of the new class (the METACLASS option). For more information on metaclasses, see *Section 4.5.4, “Metaclasses”*.
- Making the new class available to programs outside its containing Rexx program (the PUBLIC option). The outside program must refer to the new class by using a ::REQUIRES directive.
- Subclassing the new class to a mixin class in order to inherit its instance and class methods (the MIXINCLASS option).
- Adding the instance and class methods of a mixin class to the new class, without subclassing it (the INHERIT option).

When you create a new class, it is always a subclass of an existing class. If you do not specify the SUBCLASS or MIXINCLASS option on the ::CLASS directive, the superclass for the new class is the Object class.

Your class definition can be in a file of its own, with no executable code preceding it. For example, when you define classes and methods to be shared by several programs, you put the executable code in another file and refer to the class file using a ::REQUIRES directive.

Rexx processes ::CLASS directives in the order in which they appear, unless there is a dependency on some later directive's processing. You cannot create two classes that have the same class name in one program. If several programs contain classes with the same name, the last ::CLASS directive processed is used.

4.3.2.2. The ::METHOD Directive

The ::CLASS directive is usually followed by a ::METHOD directive, which is used to create a method for that class and define the method's attributes. The next directive in the program, or the end of the program, ends the method.

Some classes you define have an INIT method. INIT is called whenever a NEW message is sent to a class. The INIT method must contain whatever code is needed to initialize the object.

The ::METHOD directive can be used for:

- Creating a class method for the most-recent ::CLASS directive (the CLASS option).
- Creating a private method; that is, a method that works like a subroutine and can only be activated by the objects of the same type it belongs to—otherwise the method is public by default, and any sender can activate it.
- Creating a method that can be called while other methods are active on the same object, as described in *Section 5.7.2.1, “Activating Methods”* (the UNGUARDED option).

4.3.2.3. The ::ATTRIBUTE Directive

A ::CLASS directive can also be followed by ::ATTRIBUTE directives, which are used to create methods that directly access internal attributes of an object. For example, the Account class could define

```
::attribute balance
```

Which would allow the account balance to be set or retrieved.

```
  anAccount~balance = 10000     -- set a new account balance
  say anAccount~balance         -- display the current account balance
```

The access methods can created as read-only, or given private scope.

4.3.2.4. The ::ROUTINE Directive

You use the ::ROUTINE directive to create a named routine within a program. The ::ROUTINE directive starts the named routine and another directive (or the end of the program) ends the routine.

The ::ROUTINE directive is useful organizing functions that are not specific to a particular class type.

The ::ROUTINE directive includes a PUBLIC option for making the routine available to programs outside its containing Rexx program. The outside program must reference the routine by using a ::REQUIRES directive for the program that contains the routine.

4.3.2.5. The ::REQUIRES Directive

You use the ::REQUIRES directive when a program needs access to the classes and objects of another program. This directive has the following form:

```
::REQUIRES program_name
```

::REQUIRES directives are processed before other directives and the order of the ::REQUIRES directives determines the search order for the classes and routines defined in the named programs.

Local routine or class definitions within a program override routines or classes imported through ::REQUIRES directives.

4.3.3. How Directives Are Processed

You place a directive (and its method code) after the program code. When you run a program containing directives, Rexx:

1. Processes the directives first, to set up the program's classes, methods, and routines.
2. Runs any program code preceding the first directive. This code can use any classes, methods, and routines set up by the directives.

   Once Rexx has processed the code preceding the directive, any public classes and objects the program defines are available to programs having the appropriate ::REQUIRES directive.

4.3.4. A Sample Program Using Directives

Here is a program that uses directives to create new classes and methods:

Example 4.1. Using directives

```
asav = .savings~new              /* executable code begins */
say asav~type                    /* executable code        */
asav~name= "John Smith"          /* executable code ends   */

::class Account                  /* directives begin ...   */

  ::method type
    return "an account"

  ::attribute name

::class Savings subclass Account

  ::method type
    return "a savings account"   /* ... directives end     */
```

The preceding program uses the ::CLASS directive to create two classes, the Account class and its Savings subclass. In the **`::class Account`** expression, the ::CLASS directive precedes the name of the new class, Account.

The example program also uses the ::METHOD directive to create a TYPE method and ::ATTRIBUTE to create NAME and NAME= methods for Account. In the **`::method type`** instruction, the ::METHOD directive precedes the method name, and is immediately followed by the code for the method. Methods for any new class follow its ::CLASS directive in the program, and precede the next ::CLASS directive.

In the **`::attribute name`** directive, we're creating a pair of methods. The NAME ("getter") method returns the current value of the *NAME* object variable. The NAME= ("setter") method can assign a new value to the *NAME* object variable.

You do not have to associate object variables with a specific object. Rexx keeps track of object variables for you. Whenever you send a message to savings account **`asav`**, which points to the Name object, Rexx knows what internal object value to use. If you assign another value to **`asav`** (such as "Mary Smith"), Rexx recovers the object that was associated with **`asav`** ("John Smith") as part of its normal garbage-collection operations.

In the Savings subclass, a second TYPE method is created that supersedes the TYPE method Savings would otherwise have inherited from Account. Note that the directives appear after the program code.

4.3.5. Another Sample Program

A directive is nonexecutable code that begins with a double colon (::) and follows the program code. The ::CLASS directive creates a class; in this example, the Dinosaur class. The sample provides two methods for the Dinosaur class, INIT and DIET. These are added to the Dinosaur class using the ::METHOD directives. After the line containing the ::METHOD directive, the code for the method is specified. Methods are ended either by the start of the next directive or by the end of the program.

Because directives must follow the executable code in your program, you put that code first. In this case, the executable code creates a new dinosaur, Dino, that is an instance of the Dinosaur class. Rexx then runs the INIT method. Rexx runs any INIT method automatically whenever the NEW message is received. Here the INIT method is used to identify the type of dinosaur. Then the program runs the DIET method to determine whether the dinosaur eats meat or vegetables. Rexx saves the information returned by INIT and DIET as variables in the Dino object.

In the example, the Dinosaur class and its two methods are defined following the executable program code:

Example 4.2. Defining methods

```
dino=.dinosaur~new          /* Create a new dinosaur instance and
                                             /* initialize variables */
dino~diet                   /* Run the DIET method          */
exit
```

```
::class Dinosaur            /* Create the Dinosaur class  */

  ::method init             /* Create the INIT method     */
    expose type
```

```
     say "Enter a type of dinosaur."
     pull type
     return

   ::method diet            /* Create the DIET method     */
     expose type
     select
     when type="T-REX" then string="Meat-eater"
     when type="TYRANNOSAUR" then string="Meat-eater"
     when type="TYRANNOSAURUS REX" then string="Meat-eater"
     when type="DILOPHOSAUR" then string="Meat-eater"
     when type="VELOCIRAPTOR" then string="Meat-eater"
     when type="RAPTOR" then string="Meat-eater"
     when type="ALLOSAUR" then string="Meat-eater"
     when type="BRONTOSAUR" then string="Plant-eater"
     when type="BRACHIOSAUR" then string="Plant-eater"
     when type="STEGOSAUR" then string="Plant-eater"
     otherwise string="Type of dinosaur or diet unknown"
     end
     say string
```

4.4. Defining an Instance

You use the NEW method to define an instance of the new class, and then call methods that the instance inherited from its superclass. To define an instance of the Savings class named "John Smith," and send John Smith the TYPE and NAME= messages to call the related methods, you enter:

```
newaccount = savings~new
say newaccount~type
newaccount~name = "John Smith"
```

4.5. Types of Classes

There are four kinds of classes:

- Object classes
- Mixin classes
- Abstract classes
- Metaclasses

The following sections explain these.

4.5.1. 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.

4.5.2. 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.

4.5.3. 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 like object or mixin classes. No extra messages or keywords on the ::CLASS directive are necessary. Rexx does not prevent users from creating instances of abstract classes. It is possible to create abstract methods on a class. An abstract method is a placeholder that subclasses are expected to override. Failing to provide a real method implementation will result in an error when the abstract version is called.

4.5.4. Metaclasses

A *metaclass* is a class you can use to create another class. The only metaclass that Rexx provides is **`.Class`**, the Class 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 4.3. Creating an Array subclass

```
/* 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)
```

```
::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
```

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

Example 4.4. Adding a class method

```
/* 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
                                       /* ask the point class how many */
                                       /* instances it has created     */
say "The point class has created" .point~instances "instances."

::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         */
  return "("xVal","yVal")"             /* to return string value       */
```

A Closer Look at Objects

This chapter covers the mechanics of using objects in more detail. First, a quick refresher.

A Rexx object consists of:

- Actions coded as methods
- Attributes, coded as variables, and their values, sometimes referred to as "state data"

Sending a message to an object causes it to perform a related action. The method with the matching name performs the action. The message is the interface to the object, and with information hiding, only methods that belong to an object can access its variables.

Objects are grouped hierarchically into classes. The class at the top of the hierarchy is the Object class. Everything below it in the hierarchy belongs to the Object class and is therefore an object. As a result, all classes are objects.

In a class hierarchy, classes, superclasses, and subclasses are relative to one another. Unless designated otherwise, any class directly above a class in the hierarchy is a superclass, and any class below is a subclass.

From a class you can create instances of the class. Instances are merely similar objects that fit the template of the class; they are "of" the class, but are not classes themselves.

Both the classes and their instances contain variables and methods. The methods a class provides for use by its instances are called instance methods. The instance methods define which messages an object can respond to.

The methods available to the class itself are called class methods. Many of the methods are actually the instance methods of the Class class, but a class many have its own unique class methods. They define messages that only the class—and not its instances—can respond to.

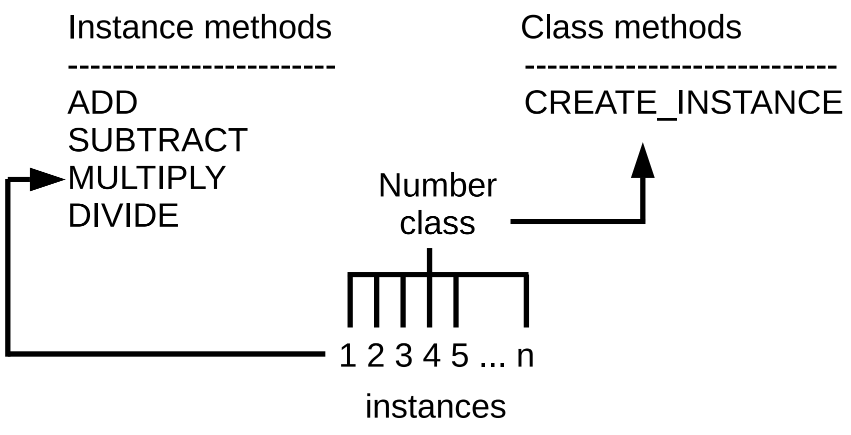


Figure 5.1. Instance Methods and Class Methods

5.1. Using Objects in Rexx

The following examples with *myarray* illustrate how to use new objects in Rexx programs.

```
myarray=.array~new(5)   /* array with initial capacity of 5 items          */
```

creates a new instance of the Array class, and assigns to the variable *myarray*. The period precedes a class name in an expression, to distinguish the class environment symbol from other variables. The *myarray* array object has an initial capacity for five items.

After the array is created, you can assign values (items) to it. One way is with the PUT method. The PUT has two arguments, which must be enclosed in parentheses. The first argument is the value (item) added to the array, the second is the index number, the number of the location at which to store the value (item) in the array object. Here, the string object **`Hello`** is stored at the index numbered **`3`** of *myarray*:

```
myarray~put("Hello",3)  /* storing "Hello" at position 3                   */
```

One way to retrieve values from an array object is by sending it an AT message. In the next example, the SAY instruction displays the value (item) stored at position (index) **`3`** of *myarray*:

```
say myarray~at(3)       /* fetching item from position 3                   */

Results:

Hello
```

The SAY instruction expects a string object as input, which is what AT returns in this case. If you try to display a non-string object in the SAY instruction, SAY requests a string object, which will cause the MAKESTRING message to be sent to the object if present, and if not the STRING message instead (the exact procedure is documented in the *Open Object Rexx: Reference*, section *4.2.11 Required String Values* and applies to any instruction or built-in function that requires a string value). In this example, the MAKESTRING method for an Array object returns the items currently stored in the array, in this case the string **`Hello`**.

```
say myarray            /* SAY causes the MAKESTRING message to be sent to myarray   */

Results:

Hello
```

By contrast the STRING method of the Array class returns the string "**`an Array`**":

```
say myarray~string     /* sending explicitly the STRING message            */

Results:

an Array
```

Whenever a method returns a string, you can use it within expressions that require a string. Here, the element of the array that AT returns is a string, so you can put an expression containing the AT method inside a string function like COPIES():

```
say copies(myarray~at(3),4)   /* fetching item from position 3, copying it 4 times   */

Results:

HelloHelloHelloHello
```

This example produces the same result using only messages that cause the appropriate methods to be run:

```
say myarray~at(3)~copies(4)   /* fetching item from position 3, copying it 4 times  */

Results:

HelloHelloHelloHello
```

Notice that the expression is evaluated from left to right. You can also use parentheses to enforce an order of evaluation.

Almost all messages are sent using the twiddle, but there are exceptions. The exceptions are to improve the readability of the language. The following example uses the []= (left-bracket right-bracket equal-sign) and [] methods to set and retrieve array elements:

```
myarray[4]="the fourth element"  /* storing string at position 4           */
say myarray[4]                /* fetching item (a string) from position 4  */

Results:

the fourth element
```

Although the previous instructions look like an ordinary array assignment and array reference, they are actually messages to the Array object referenced by *myarray*. You can prove this by executing these equivalent instructions, which use the twiddle to send the messages:

```
myarray~"[]="("a new test",4)    /* storing string at position 4           */
say myarray~"[]"(4)           /* fetching item (a string) from position 4  */

Results:

a new test
```

Similarly, expression operators (such as +, -, /, and *) are actually methods, but you do not have to use the twiddle to send them:

```
say 2+3                  /* adds 3 to 2, displays result: 5 */
say 2~"+"(3)             /* message version: adds 3 to 2, displays result: 5 */
```

In the second SAY instruction, "+" must be a literal string because the message name contains characters not allowed in a Rexx symbol.

5.2. Common Methods

When running your program, four methods that Rexx looks for, and runs automatically when appropriate, are INIT, *UNINIT*, MAKESTRING and STRING.

5.2.1. Initializing Instances Using INIT

Object classes can create instances. When these instances require initialization, you'll want to define an INIT method to set a particular starting value or initiate some startup processing. Rexx looks for an INIT method whenever a new object is created and runs it.

The purpose of initialization is to ensure that the instance variables can be initialized, if needed, before being used in an operation. After the NEW method has created the new instance, but before returning it, the new instance gets the INIT message sent to it. Any (initialization) arguments specified with the NEW message are passed on to the INIT method in the same order, which can use them to set the initial states of object variables.

If a class overrides the INIT method it inherits from a superclass, the new INIT method must forward the INIT message up the hierarchy, to properly initialize the instance, using the *SUPERCLASS overrides*, so that each inherited INIT method has the opportunity to run (using e.g. the statement **`self~init:super`**). An example in the next section demonstrates the use of INIT.

5.2.2. The MAKESTRING and STRING Methods

The MAKESTRING method is optional and if present in a class allows for rendering each of its instances as string objects. Each such rendering will usually encode its attributes (object variables) as strings in one form or another.

Whenever an object is supplied as an argument to an instruction or built-in function that works on strings only, a *required string value* gets requested. If a MAKESTRING method is available it will be run and the resulting string will be used instead.

In the case that there is no MAKESTRING method, then the STRING gets employed instead, which is guaranteed to exist as it gets defined in the Rexx root class Object. The returned string value in this case will usually consist of the indefinite article for the class name and gets concatenated with a blank and with the class name itself.

The exact procedure for getting at a required string value is documented in the *Open Object Rexx: Reference*, section *4.2.11 Required String Values*

You can take advantage of this automatic use of MAKESTRING and STRING by overriding (implementing) either or both methods in your own Rexx classes.

The following programs demonstrate STRING and INIT. In the first program, the Part class is created, and along with it, the two methods under discussion, STRING and INIT:

Example 5.1. STRING and INIT Methods

```
/* partdef.rex - Class and method definition file */

/* Define the Part class as a public class */
::class Part public

/* Define the INIT method to initialize object variables */
::method init
   expose name description number
   use arg name, description, number

/* Define the STRING method to return a string with the part name */
::method string
   expose name
   return "Part name:" name
```

In the ::CLASS directive, the keyword PUBLIC indicates that the class can be shared with other programs. The two ::METHOD directives define INIT and STRING. Whenever Rexx creates a new instance of a class, it calls the INIT method of the new instance. The sample INIT method uses an EXPOSE instruction to allow the method direct access to the object variables `name`, `description`, and `number`:

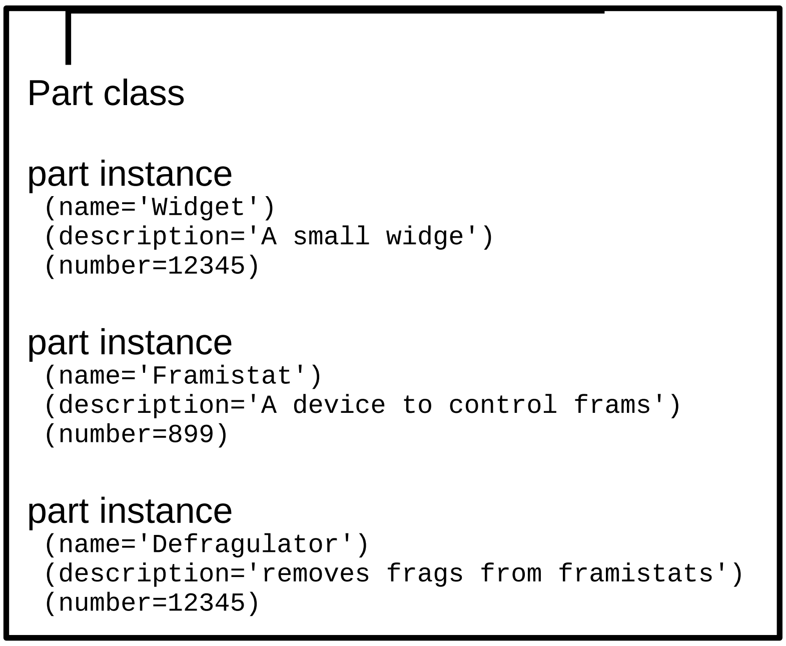


Figure 5.2. Instances in the Part Class

The INIT method expects to be passed three arguments. The USE ARG instruction assigns these three arguments to the exposed object variables `name`, `description`, and `number`, respectively.

The STRING method returns the string **`Part name:`**, followed by the name of a part. The STRING method (of the Part class) does not expect any arguments. It uses the EXPOSE instruction to be able to directly access the object variable `name`. The RETURN instruction returns the result string.

The following example shows how to use the Part class:

Example 5.2. How to use the Part class

```
/* usepart.rex - use the Part class */
mypartA=.part~new("Widget","A small widge",12345)
mypartB=.part~new("Framistat","Device to control frams",899)
say mypartA
say mypartB
exit
::requires partdef

Results:

Part name: Widget
Part name: Framistat
```

The usepart.rex program creates two parts, which are instances of the Part class. It then displays the names of the two parts.

Rexx processes all directives before running your program. The ::REQUIRES directive indicates that the program needs access to public class and public routine definitions that are in another program. In this case, the ::REQUIRES directive refers to the partdef.rex program, which contains the Part definition.

The assignment instructions for *mypartA* and *mypartB* create two objects that are instances of the Part class. The objects are created by sending a NEW message to the Part class. The NEW message causes the NEW method of the Part class to be invoked, which sends the INIT message together with the three received arguments to the newly created instance, which in turn runs the INIT method as part of the object creation. The INIT method takes the three arguments you provide and saves them in its object's variables: `name`, `description`, and `number`:

The SAY instruction causes Rexx to create the required string value: as there is no MAKESTRING method in the Part class the STRING method gets invoked to return a string value. The STRING method uses EXPOSE to become able to directly access the `name` object variable and returns it as part of the string (**`return "Part name:" name`**).

5.2.3. Uninitializing and Deleting Instances Using UNINIT

Normally, object classes can create instances but have no direct control over their deletion. Once an object is no longer referenced in the program, it turns to garbage and Rexx automatically reclaims the storage it occupies in the computer's memory in a process called garbage collection.

If the instance has allocated other system resources, Rexx cannot automatically release these resources because it is unaware that the instance has allocated them. An UNINIT method gives an object the opportunity to perform resource cleanup before the object is reclaimed by the garbage collector.

The following example shows how Rexx automatically invokes INIT and UNINIT.

Example 5.3. UNINIT Method

```
/* uninit.rex - example of UNINIT processing */

a = .scratchpad~new("Of all the things I've lost")
a = .scratchpad~new("I miss my mind the most")
say "Exiting program."

::class scratchpad

  ::method init
    expose text
    use arg text
    say "Remembering" text

  ::method uninit
    expose text
    say "Forgetting" text
```

Whether uninitialization processing is needed depends on the circumstances, If the object only contains references to normal Rexx objects, an UNINIT method is generally not needed. If the object contains references to external system resources such as open network connections or database connections, an UNINIT method might be required to release those resources. If an object requires uninitialization, define an UNINIT method to perform the cleanup processing you require.

If an object has an UNINIT method, Rexx runs it before reclaiming the object's storage. If an instance overrides an UNINIT method of a superclass, each UNINIT method is responsible for sending the UNINIT message up the hierarchy, using the *SUPERCLASS overrides* (by using the statement: **`"self~uninit:super"`**), so that each inherited UNINIT method has the opportunity to run.

5.3. Special Method Variables

When writing methods, there are several special variables that are set automatically when a method runs. Rexx supports the following variables:

SELF

is set when a method is activated. Its value is the object that forms the execution context for the method (that is, the object that received the activating message).

You can use *SELF* to:

- Send messages to the currently active object. 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 regarding 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.

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.

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 5.4. SELF Variable

```
::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. For example:

```
self~init:super(balance)
```

This instruction sends the INIT message to the Account class rather than recursively invoking 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 as this is the Account class' direct super class:

```
self~init:super
```

calls the INIT method of the Object class.

5.4. Public, Local, and Built-In Environment Objects

In addition to the special variables, Rexx provides a unique set of objects, called *environment objects*. Rexx makes the following environment objects available:

5.4.1. The Global Environment Object (.environment)

The environment object **`.environment`** is a directory (an instance of the Rexx Directory class) of public objects that are always accessible throughout the whole process. Each entry in this directory can itself be referred to by an environment symbol that starts with a dot, immediately followed by its index name. All the public Rexx built-in classes are stored in the environment directory **`.environment`**, therefore the public Rexx array class can be fetched by its environment symbol **`.ARRAY`** or alternatively with **`.environment~array`**. To place something in the environment directory, you use the form:

```
.environment~your.object = value
```

Include a period (`.`) in any index name you use, to avoid conflicts with current or future Rexx entries to the environment directory **`.environment`**. In the above example the index name used for storing *value* within the environment directory is **`YOUR.OBJECT`**. To retrieve the *value* stored with the index name **`YOUR.OBJECT`** from the **`.environment`** directory, you use one of the following two forms:

```
say .environment~your.object
say .your.object        /* same as above        */
```

The scope of **`.environment`** is the current process, which means it is shared among all interpreter instances executing in this process.

You use an environment symbol to access the entries of this directory. An *environment symbol* starts with a period and has at least one other character, and the symbol is not a valid numeric value. You have seen environment symbols earlier; for example in:

```
asav = .savings~new
```

`.savings` is an environment symbol, and refers to the Savings class. The classes you create can be referenced with an environment symbol. There is an environment symbol for each public Rexx-defined class, as well as for each of the unique environment objects this section describes, such as *The Nil Object*.

5.4.1.1. The NIL Object (.nil)

The NIL object **`.nil`** is a special environment object that does not contain any data. It represents the absence of an object, the way a null string represents a string with no characters. Its only methods are those of the Object class. You can use *The NIL object* (rather than the null string) to test for the absence of data in an array entry:

```
if board[row,column] = .nil
then ...
```

5.4.1.2. The False Environment Object (.false)

The environment object **`.false`** simply allows for retrieving the value **`0`** stored in the environment directory **`.environment`**, which is the value used in Rexx to represent the boolean value *"false"* (*"not true"*). You can use the environment symbol **`.false`** instead of the literal **`0`** to make clear that you are using a boolean value (and not a number or a string).

5.4.1.3. The True Environment Object (.true)

The environment object **`.true`** simply allows for retrieving the value **`1`** stored in the environment directory **`.environment`**, which is the value used in Rexx to represent the boolean value *"true"* (*"not false"*). You can use the environment symbol **`.true`** instead of the literal **`1`** to make clear that you are using a boolean value (and not a number or a string).

All the environment objects Rexx provides are single symbols. Use compound symbols when you create your own, to avoid conflicts with future Rexx-defined entries.

5.4.2. The Local Environment Object (.local)

The local environment object **`.local`** is a directory (an instance of the Rexx Directory class) of interpreter instance-specific objects that are always accessible. To place something in the Local environment directory, you use the form:

```
.local~your.object =  value
```

Include a period (**`.`**) in any index name you use, to avoid conflicts with current or future Rexx entries to the local environment directory **`.local`**. In the above example the index name used for storing *value* within the local environment directory is **`YOUR.OBJECT`**. To retrieve the *value* stored with the index name **`YOUR.OBJECT`** from the **`.local`** directory, you use one of the following two forms:

```
say .local~your.object
say .your.object        /* same as above        */
```

The scope of **`.local`** is the current interpreter instance. Each Rexx interpreter instance has its own copy of the local environment directory **`.local`**.

You access objects in the **`.local`** directory like in the global **`.environment`** directory. Rexx provides the following objects in the local environment:

`.input`
: is the input monitor (an instance of the Rexx class Monitor) object which is the source for the PARSE LINEIN instruction, the LINEIN method and CHARIN method of the Stream class, and (if you do not specify a stream name) the LINEIN, respectively CHARIN built-in functions. It is also the source of the PULL and PARSE PULL instructions if the external data queue is empty. By default **`.input`** forwards the received messages to the **`.stdin`** stream object.

`.output`
: is the output monitor (an instance of the Rexx class Monitor) object, which is the destination of output from the SAY instruction, the LINEOUT (SAY) and CHAROUT methods of the Stream class, and (if you do not specify a stream name) the LINEOUT built-in function. You can replace this object's destination to redirect such output elsewhere, for example to a transcript window. By default **`.output`** forwards the received messages to the **`.stdout`** stream object.

`.error`
: is the error monitor (an instance of the Rexx class Monitor) object to which Rexx writes error messages to. By default **`.error`** forwards the received messages to the **`.stderr`** stream object.

`.debuginput`
: is the debug input monitor (an instance of the Rexx class Monitor) object from which the Rexx TRACE instruction reads debug input while in interactive mode. By default **`.debuginput`** forwards the received messages to the **`.input`** monitor object.

`.traceoutput`
: is the trace output monitor (an instance of the Rexx class Monitor) object to which Rexx writes trace output to. By default **`.traceoutput`** forwards the received messages to the **`.error`** monitor object.

`.stdin`
: is the operating system's standard input stream (an instance of the Rexx class Stream), by default the keyboard input device. The standard input stream can be redirected via the operating system and has always the file descriptor value **`0`**.

`.stdout`
: is the operating system's standard output stream (an instance of the Rexx class Stream), by default the console (terminal) device. The standard output stream can be redirected via the operating system and has always the optional file descriptor value **`1`**.

`.stderr`
: is the operating system's standard error stream (an instance of the Rexx class Stream), by default the console (terminal) device. The standard output stream can be redirected via the operating system and has always the file descriptor value **`2`**.

`.syscargs`
: is optional and will be set by Rexx if the Rexx program was called with arguments from the commandline. Rather than returning the commandline argument as a single string like the ARG() built-in function does, **`.syscargs`** array object will have the commandline argument string decomposed according to the rules of the programming language *C*. If this environment object is not set, the name of the environment symbol will be returned instead in uppercase letters, i.e. **`.SYSCARGS`** (note the leading dot which makes this symbol an environment symbol).

Example 5.5. .SYSCARGS Environment Object

```
/* results for running this program with command line arguments
     one "second argument" three

   3 arguments
   one
   second argument
   three
*/
say .sysCargs~items "arguments"
say .sysCargs
```

5.4.3. Built-In Environment Objects

Rexx provides environment objects that all programs can use. To access these built-in objects, you use their environment symbols whose first character is a period (`.`).

`.line`

The **`.line`** environment symbol returns the line number of the current instruction being executed. If the current instruction is defined within an INTERPRET instruction, the value returned is the line number of INTERPRET instruction.

`.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

Tracing interactively does not change the value of **`.rs`**. The initial value of **`.rs`** is **`0`**.

5.4.4. The Default Search Order for Environment Objects

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

1. The classes and routines maintained in the current package by the interpreter. This package can be retrieved programmatically with the statement **`.context~package`**.

2. *NEW* The current package *local* directory. The package local directory can be accessed programmatically with the statement **`.context~package~local`**.

3. The program local environment directory **`.local`**, which includes interpreter instance-specific objects such as the **`.input`** and **`.output`** objects. You can directly access the local environment directory by using the **`.local`** environment symbol.

4. The global environment directory **`.environment`**, which includes all "global" Rexx objects such as the Rexx-supplied classes (for example, **`.Array`**) and constants such as **`.true`** and **`.false`**. You can directly access the global environment by using the **`.environment`** symbol or using the VALUE built-in function with a null string for the *selector* argument.

5. Rexx defined symbols. Other simple environment symbols are reserved for use by Rexx for built-in objects.

If an entry is not found for an environment symbol, the default character string value (uppercased) of the environment symbol is used including its leading dot.

Note

You can place entries in the **`.context~package~local`**, **`.local`** and **`.environment`** directories for programs to use, but **`.local`** should be preferred over **`.environment`** to avoid accidentally overwriting system-defined values. To avoid conflicts with future Rexx-defined entries, it is recommended that entries you place in either of these directories include at least one period in the entry name.

Example 5.6. .Local Object

```
/* establish a settings directory in the local environment directory */
.local~setentry("my.settings", .directory~new)
```

5.5. Determining the Scope of Methods and Variables

Methods may interact with object variables and their associated data. But a method cannot directly interact with any object variable, only with those object variables that are defined in the same class as the method (encapsulation). This protects the object variables' data from being changed by "unauthorized" methods belonging to other classes.

All Methods of a particular class are able to directly access object variables by using as their very first instruction the EXPOSE instruction followed by a blank delimited list of the object variable names the method wishes to interact directly.

5.5.1. Objects with a Class Scope

Encapsulation usually takes place at the class level. The class is designed as a template of methods and variables. The instances themselves retain only the values of their object variables.

Within the hierarchy, the class structure ensures the integrity of a class's object variables, controlling the methods allowed to operate on them. The class structure also provides for easy updating of the method code. If a method requires a change, you only have to change it once, at the class level. The change then is acquired by all the instances sharing the method.

Associated methods and variables have a certain scope, which is the class to which they belong:

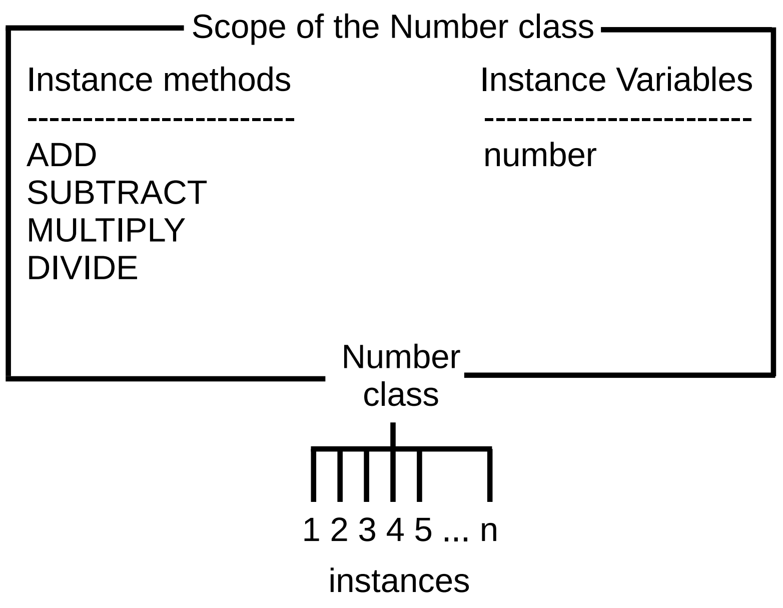


Figure 5.3. Scope of the Number Class

Each class in a class hierarchy has a scope different from any other class. This is what allows an object variable in a subclass to have the same name as an object variable in a superclass, even though the methods that use the object variables for completely unrelated purposes.

5.5.2. Objects with Their Own Unique Scope

The methods and attributes (object variables) used by instances in a class are usually found at the class level. But sometimes an instance differs in some respect from the others in its class. It might perform an additional action or require some unique handling. In this case one or more methods and related attributes (object variables) can be added directly to the instance. These additional methods and attributes (object variables) form a separate scope, independent of the class scopes found throughout the rest of the hierarchy.

Methods can be added directly to an instance's collection of object methods using SETMETHOD, a method of the Object class. All subclasses of the Object class inherit SETMETHOD. Alternately, the Class class provides an ENHANCED method that lets you create new instances of a class, whose object methods are the instance methods of its class, but enhanced with the additional methods from a supplied collection of methods.

5.6. More about Methods

A method name can be any character string. When an object receives a message, Rexx searches for a method whose name matches the message name independent of its case.

You must surround a method name with quotation marks when it is the same as an operator. The following example illustrates how to do this correctly. It creates a new class (Cost), defines a new method (%), creates an instance of the Cost class (*mycost*), and sends a % message to *mycost*:

Example 5.7. Messages

```
say "Enter a price:"
pull p
mycost=.Cost~new(p)        /* Create a Cost instance        */
```

```
say "price:" mycost~price  /* ask current price            */
mycost~"%"                 /* send % (increase) message     */
say "price:" mycost~price  /* ask current price            */
say "%    :" mycost~"%"    /* send % (increase) message     */

::class Cost               /* Cost class                   */
::attribute price          /* allow access from others      */
::method init
  expose price increase    /* establish direct access to these attributes */
  use arg price            /* save argument with attribute  */
  increase=25              /* increase in percent           */

::method "%"               /* Increase price % method       */
  expose price increase    /* Produces: Enter a price.      */
  price=price*(100+increase)/100 /* increase                */
     return price              /* return increased price        */

Results (example

Enter a price:
100
price: 100
price: 125
%    : 156.25
```

5.6.1. The Default Search Order for Selecting a Method

When a message is sent to an object, Rexx looks for a method whose name matches the message string ignoring case. If the message is ADD, for example, Rexx looks for a method named ADD. Because, in the class hierarchy, there may be more than one method with the same name, Rexx begins its search at the object specified in the message. If the sought method is not found there, the search continues up the hierarchy. Rexx searches in the following order:

1. A method the object defines itself (with SETMETHOD or ENHANCED).

2. A method the object's class defines.

   An object acquires the methods of its parent class (that is, the class that created the object). If the class subsequently receives new methods, objects predating the new methods *do not* acquire them.

3. A method an object's superclass(es) define.

   As with the object's class, only methods that existed in the superclass when the object was created are valid. Rexx searches the superclass method definitions in the order that INHERIT messages were sent to an object's class.

If Rexx does not find a match for the message name, Rexx checks the object for a method named UNKNOWN. If it exists, Rexx calls the UNKNOWN method, and returns whatever the UNKNOWN method returns. For more information on the UNKNOWN method, see *Section 5.6.4, “Defining an UNKNOWN Method”*. If the object does not have an UNKNOWN method, Rexx raises a NOMETHOD condition. Any trapped information can then be inspected using Rexx's CONDITION built-in function.

Rexx searches up the hierarchy so that methods existing in classes at higher levels can be found. This search realizes the *inheritance of methods* from superclasses.

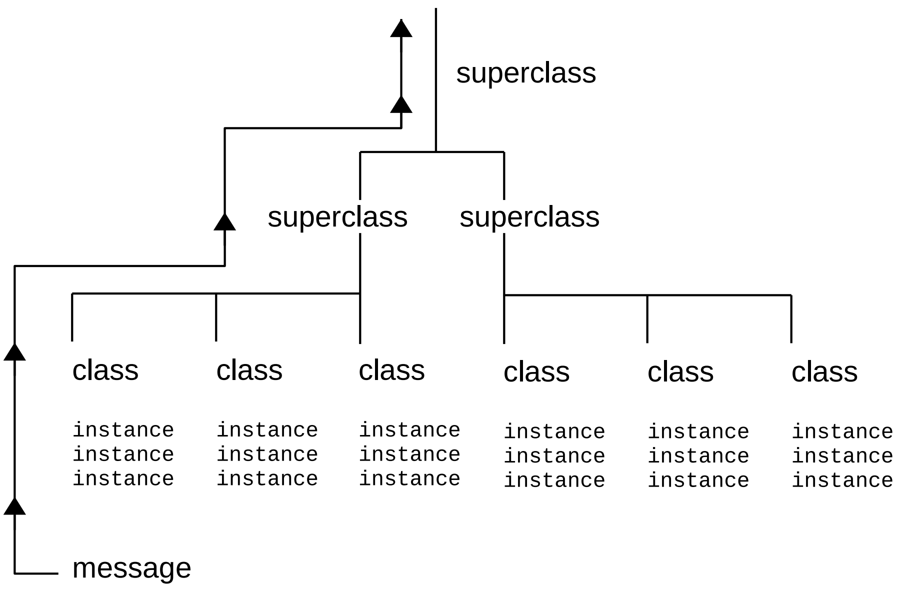


Figure 5.4. Searching the Hierarchy for a Method

For example, suppose you wrote a program that allows users to look up other users' phone numbers. Your program includes a class called Phone_Directory, and all its instances are users' names with phone numbers. You have included a method in Phone_Directory called NOTIFY that reports some data to a file whenever someone looks up a number. All instances of Phone_Directory use the NOTIFY method.

Now you decide you want NOTIFY, in addition to its normal handling, to personally inform you whenever anyone looks up your number. To accommodate this special case for your name only, you create your own NOTIFY method that adds the new task and replicates the file-handling task. You save the new method as part of your own name instance, retaining the same name, NOTIFY.

Now, when a NOTIFY message is sent to your name instance, the new version of NOTIFY is found first. Rexx does not look further up the class hierarchy. The instance-level version overrides the version at the class level. This technique of overriding lets you change a method used by one instance ("one-off" instance) without disturbing the common method used by all the other instances. It is very powerful for that reason.

5.6.2. Changing the Search Order for Methods

When composing a message, you can change the default search order for methods by doing both of the following:

1. Making the receiver object the sender object. You usually do this by specifying the special variable *SELF*. *SELF* holds the value of the object in which a method is running.

2. Specifying a colon and a starting scope after the message name. The starting scope is a variable or environment symbol that identifies the scope object to use as the method search starting point. This scope object can be:
   - A direct superclass of the class that defines the active method
   - The object itself (for example, the value of the variable *SELF*), if you used SETMETHOD to add methods to the object.

   The scope variable is usually the special variable *SUPER*, but it can be any environment symbol or variable name whose value is a valid superclass.

In *Section 4.3.4, “A Sample Program Using Directives”*, an Account subclass of the Object superclass is created. It defines a TYPE method for Account, and creates the Savings subclass of Account.. The example defines a TYPE method for the Savings subclass, as follows:

Example 5.8. SUBCLASS Option

```
::class Savings subclass Account

  ::method "TYPE"
    return "a savings account"
```

To change the search order so Rexx searches for TYPE in the Account rather than Savings subclass, enter this instead:

Example 5.9. Changing the Subclass Method Search Order

```
  ::method "TYPE"
    return self~type:super -- returns the result of invoking the TYPE method of the
 superclass
```

When you create an **asav** instance of the Savings subclass and send a TYPE message to **asav**:

```
say asav~type
```

Rexx displays:

```
an account
```

rather than:

```
a savings account
```

because Rexx searches for TYPE in the Account class first.

5.6.3. Public versus Private Methods

A method can be either public or private. Any object can send a message that runs a *public* 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 5.10. PUBLIC and PRIVATE options

   ```
   ::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 5.11. PUBLIC and PRIVATE Options

```
::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.

5.6.4. Defining an UNKNOWN Method

When an object receives a message that has no matching method, Rexx checks to see if the object has a method named UNKNOWN. If it does, Rexx calls the UNKNOWN method, passing two arguments. The first is the name of the method that was not located. The second is an array containing the arguments passed with the original message.

5.7. Concurrency

In object-oriented programming, as in the real world, objects interact with each other. Assume, for example, throngs of people interacting at rush hour in the business district of a big city. A program that aspires to simulate the real world would have to enable many objects to interact at any given time. That could mean thousands of objects running simultaneously, all of them sending messages to each other. In Rexx, the term for this simultaneous activity is called concurrency. To be precise, the concurrency is *object-oriented* concurrency because it involves objects, as opposed to, for example, processes or threads.

Rexx objects are inherently concurrent, and this concurrency takes the following forms:

- *Inter-object concurrency*, where several objects are active (exchanging messages, synchronizing, running their methods, etc.) at the same time

- *Intra-object concurrency*, where several methods are able to run on the same object at the same time

The default settings in Rexx allow full inter-object concurrency but limited intra-object concurrency. Some situations, however, call for full intra-object concurrency.

5.7.1. Inter-Object Concurrency

Rexx provides for inter-object concurrency, where several objects in a program can run at the same time, in the following ways:

- By early reply, using the REPLY instruction
- Using message objects

Early reply allows the object that sends a message to continue processing after the message is sent. Meanwhile, the receiving object runs the method corresponding to the message. This method contains the REPLY instruction, which returns any results to the sender, interrupting the sender just long enough to reply. The sender and receiver continue operating simultaneously.

Alternatively, an independent message object can be created and sent to a receiver. One difference in this approach is that any reply returned does not interrupt the sender. The reply waits until the sender asks for it. In addition, message objects can notify the sender about the completion of the method it sent, and even specify synchronous or asynchronous method activation.

The chains of execution represented by the sender and receiver methods are called *activities*. An *activity* is a thread of execution that can run methods concurrently with methods on other activities. In other words, *activities* can run at the same time.

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
- A message invocation

An invocation is pushed onto an *activity* when an executable unit is invoked. It is removed (or popped) when execution completes.

5.7.1.1. Object (Instance) Variables

Every object has its own set of instance variables, also known as attributes. These are variables associated solely with the object. When an object's method runs, it first identifies the instance variables it intends to work with. Technically, it "exposes" these instance (object) variables, using the Rexx instruction EXPOSE. Exposing the object's variables distinguishes them from variables used by the method itself, which are not exposed. Every method an object owns—that is, all the instance methods in the object's class—can expose variables from the object's instance variables.

Therefore, an object's instance variables includes variables:

- Exposed by methods defined by the object's class. This set of instance variables is called an object variable pool.

- Exposed by methods defined by other classes in the inheritance hierarchy. The methods of each class share object variables in a pool scoped to just that class.

A class's object variable pool, together with the methods that expose them, is called a class scope. Rexx exploits this class scope to achieve concurrency. To explain in more detail, the object's instance variables are contained in a collection of instance variable pools. Each instance variable pool is at a different scope in the object's inheritance chain. Methods defined at different class scopes do not directly share data, and can therefore run simultaneously.

Scopes, like objects, hide and protect data from outside manipulation. Methods of the same scope share the instance variable pool of that scope. The scope shields the instance variable pool from methods operating at other scopes. This is why you can reuse object variable names from class to class, without the variables being accessed and possibly corrupted by a method outside their own class. So class scopes serve to divide an object's instance variables into pools that can operate independently of each another. Several methods can use the same object instance variables concurrently, as long as they confine themselves to instance variables in their own scope.

5.7.1.2. Prioritizing Access to Instance Variables

Even with class scopes and subpools, an object variable is vulnerable if several methods within the scope try to access it at the same time. To handle this, Rexx ensures that when a particular method is activated and exposes object variables from its scope, that method has exclusive use of the scope variable pool until processing is complete. Until then, Rexx delays the execution of any other method that needs the same scope object variables.

Thus if different activities send several messages to the same object, Rexx forces the methods to run sequentially within a single scope. This "first-in, first-out" processing of methods in a scope prevents them from simultaneously accessing one object variable, and possibly corrupting the data.

5.7.1.3. Sending Messages within an Activity

Rexx makes one exception to sequential processing—when a method sends a message to itself. Assume that method *M1* has exclusive access to object *O*, and then tries to run a second, internal method *M2*, also belonging to *O*. Internal method *M2* would try to run, but Rexx would delay it until the original method *M1* finished. Yet *M1* would be unable to proceed until *M2* ran. The two methods would become deadlocked. In actual practice Rexx intervenes by treating internal method *M2* like a subroutine call. In this case, Rexx runs method *M2* immediately, then continues processing method *M1.*

The mechanism controlling this is the *activity*. Typically, whenever a message is invoked on an object, the *activity* acquires exclusive access by locking the object's scope. Any other *activity* sending a message to the object whose scope is locked must wait until the first activity releases the lock. The situation is different, however, if the messages originate from the same *activity*. When an invocation running on an *activity* sends another message to the same object, the method is allowed to run because the activity has already acquired the lock for the scope. Thus, Rexx permits nested, nonconcurrent method invocations on a single activity. No deadlocks occur because Rexx treats these additional messages as subroutine calls.

5.7.2. Intra-Object Concurrency

Several methods can access the same object at the same time only if they are operating at different scopes. That is because they are working with separate variable subpools. If two methods in the same scope try to run on the object, Rexx by default processes them on a "first-in, first-out" basis, while treating internal methods as subroutines. You can, however, achieve full intra-object concurrency. Rexx offers several mechanisms for this, including:

- The UNGUARDED option of the ::METHOD directive, which provide unconditional intra-object concurrency.
- The GUARD OFF and GUARD ON instructions, which permit switching between intra-object and default concurrency.

When intra-object concurrency at the scope level is needed, you must specifically employ these mechanisms (see the following section). Otherwise, Rexx sequentially processes the methods when they are competing for the same object variables.

5.7.2.1. Activating Methods

By default, Rexx assumes that an active method requires exclusive use of its scope variable pool. If another method attempts access at that time, it is locked out until the first method finishes. This default intra-object serialization maintains the integrity of the instance variable pool and prevents unexpected results. Rexx manages queues for incoming requests that result in messages being sent to the same object.

Some methods can run concurrently without affecting instance variable pool integrity or yielding unexpected results. When a method does not need exclusive use of its object variable pool, use the UNGUARDED option of the ::METHOD directive to provide unconditional intra-object concurrency. These mechanisms control the locking of an object's scope when a method is invoked.

Many methods cannot use the UNGUARDED option because they sometimes require exclusive use of their object variable pool. At other times, they must perform some action that involves the concurrent use of the same pool by a method on another activity. In this case, you can use the GUARD keyword instruction. When the method reaches the point in its processing where it no longer requires exclusive use of the object variable pool, it can use the GUARD OFF instruction to allow methods running on different activities to become active on the same scope. If the method needs to regain exclusive use, it uses the GUARD ON instruction.

For more flexibility when activating methods, you can use GUARD ON/OFF with the "WHEN *expression*" option. Add this instruction to the method code at the point where exclusive use of the object variable pool becomes conditional. When processing reaches this point, Rexx evaluates *expression* to determine if it is true or false.

For example, if you specify "GUARD OFF WHEN *expression*", the active method keeps running until *expression* becomes true. To become true, another method must assign or drop an object variable that is named in *expression*. Whenever an object variable changes, Rexx reevaluates *expression.* If *expression* becomes true, GUARD is turned off, exclusive use of the object variable pool is released, and other methods needing exclusive use can begin running. If *expression* becomes false again, GUARD is turned on and the active method regains exclusive use.

Note

If *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. A second activity can make *expression* invalid before GUARD ON WHEN can use it.

Commands

From a Rexx program you can pass commands to Windows and Unix/Linux shells or to applications designed to work with Rexx. When used to run operating system commands, Rexx becomes a powerful substitute for the Windows Batch Facility or Unix shell scripts. You can use variables, control structures, mathematics, and parsing, and the full object oriented features of Rexx.

Applications that are designed to work with Rexx are often referred to as scriptable applications. To work with Rexx, a scriptable application registers a command environment with Rexx. An environment serves as a kind of workspace shared between Rexx and the application that accepts application commands issued from your Rexx programs.

For example, many editors provide a command prompt or dialog box from which you can issue commands to set margins or add lines. If the editor is scriptable from Rexx, you can issue the same editor commands from a Rexx program. These Rexx programs are referred to as macros.

When an application runs a Rexx macro, Rexx directs commands to the application's environment. The application processes the command, and returns a status indicator as a return code.

The Rexx ADDRESS instruction allows you select which named command environment commands get directed to.   There is always at least one active command environment, and all Rexx programs start with a default environment selected. For programs launched from a command shell, an operating-system-specific command handler is the normal default. Applications (such as an editor) can choose whether or not to make their own command environment the default.

6.1. How to Issue Commands

Rexx makes it easy to issue commands. The basic rule is that whatever Rexx cannot process directly gets passed to the current command environment. You can:

- Allow Rexx to evaluate part or all of a clause as an expression. Rexx automatically passes the resulting string to the default environment.
- Enclose the entire clause in quotation marks. This makes it a literal string for Rexx to pass to the default environment.
- Send a command explicitly to a command environment using the ADDRESS instruction.

Rexx processes your program one clause at a time. It examines each clause to determine if it is:

- A directive, such as ::CLASS or ::METHOD
- A message instruction, such as:

  ```
  .array~new
  ```

- A keyword instruction, such as:

  ```
  say "Type total number"
  ```

  or

  ```
  pull input
  ```

- A variable assignment (any valid symbol followed by an equal sign), such as:

```
price = cost * 1.2
```

- A label for calling other routines
- A null (empty) clause

If the clause is none of the above, Rexx evaluates the entire clause as an expression and passes the resulting string to the current command environment.

If the string is a valid command for that environment, the command handler will process it as if you had entered it at the command prompt.

The following example shows a Rexx clause that uses the Windows *DIR* command to display a list of files in the current directory.

```
/* display current directory */
say "DIR command using Rexx"
dir
```

The clause **`dir`** is not a Rexx instruction or a label, so Rexx evaluates it and passes the resulting string to Windows. Windows recognizes the string *DIR* as one of its commands and processes it.

Letting Rexx evaluate the command as an expression might cause problems, however. Try adding a path to the DIR command in the above program (such as, **`dir c:\Windows`**). The Windows command in this case is an incorrect Rexx expression. The program ends with an error.

A safer way to issue commands is by enclosing the command in quotes, which makes the command a literal string. Rexx does not evaluate the contents of strings, so the string is passed to Windows as-is. Here is an example using the *PATH* command:

```
/* display current path      */
say "PATH command using Rexx"
"path"
```

The following example, **`dp.rex`**, shows a program using the *DIR* and *PATH* commands. The *PAUSE* command is added to wait for the user to press a key before issuing the next instruction or command. Borders are added too.

Example 6.1. DIR and PATH commands

```
/* dp.rex -- Issue DIR and PATH commands to Windows */

say "="~copies(40)    /* display line of equal   */
                      /* signs (=) for a border  */

"dir"                 /* display listing of      */
                      /* the current directory   */

"pause"               /* pauses processing and   */
                      /* tells user to "Press    */
                      /* any key to continue."   */

say "="~copies(40)    /* display line of =       */
"path"                /* display the current     */
                      /* PATH setting            */
```

When you specify the following:

```
[C:\]rexx dp
```

a possible output would be:

```
========================================

The volume label in drive C is WIN.
Directory of C:\EXAMPLES

.            <DIR>     10-16-94  12:43p
..           <DIR>     10-16-94  12:43p
EX4_1    CMD      nnnn  10-16-94   1:08p
DEMO     TXT       117  10-16-94   1:10p
4 File(s)   12163072 bytes free
Press any key when ready . . .

========================================
PATH=C:\WINDOWS
[C:\]
```

Note

Usually, when executing a host command addressed to the Windows or Unix/Linux command shell, a new process is created in the system command handler to execute the command. Changes in a child process environment do not change the parent process environment. Therefore, any change in the environment, such as a directory change, made by a host command executed in a child process would not be reflected in the process running the Rexx program.

The interpreter attempts to mitigate this to some extent by executing some host commands in the process running the Rexx program, rather than in a child process. This is done so that changes to the environment made by executing the host command are visible in the process running the Rexx program.

This is only done when the host command line is simple. That is, the command line must contain a single command, without redirection and without pipe. On Windows this applies to the **`CD`** and **`SET`** commands. On Unix-like systems, including Linux, this applies to **`cd`**, **`set`**, **`unset`** and **`export`**. Rather than remembering the rules, it may be easier to avoid a potential problem by using the built in **`directory()`** or **`value()`** functions rather than issuing a host command for **`cd`**, **`set`**, etc.

Example 6.2. Environment commands (Windows)

```
'cd c:\tmp'               /* executed in Rexx program process */
'cd "c:\R&D (secret)"'    /* executed in Rexx program process */
'cd c:\windows && dir c:' /* executed in child process (2 commands) */
'd:'                      /* executed in Rexx program process */
'set myvar=my value'      /* executed in Rexx program process */
```

Example 6.3. Environment commands (Unix)

```
'cd'                      /* executed in Rexx program process: go to $HOME directory
 */
'cd ~/"R&D (secret)"'     /* executed in Rexx program process: go to $HOME/R&D
 (secret) */
'cd ~/"R&D \"secret\""'   /* executed in Rexx program process: go to $HOME/R&D
 "secret" */
'cd ~john'                /* executed in Rexx program process: go to John's home
 directory */
'cd /tmp && pwd'          /* executed in child process (2 commands) */
'set myvar=my value'      /* executed in Rexx program process */
'export myvar=my value'   /* executed in Rexx program process */
'unset myvar'             /* executed in Rexx program process */
```

6.2. Rexx and Batch Files

You can use a Rexx program whenever you now use Windows batch files or Unix/Linux shell scripts. The following example shows a Windows batch file that processes user input to display a help message:

Example 6.4. Windows batch file

```
@echo off
if %1.==. goto msg
if %1 == on goto yes
if %1 == off goto no
if %1 == ON goto yes
if %1 == OFF goto no
if %1 == On goto yes
if %1 == oN goto yes
if %1 == OFf goto no
if %1 == OfF goto no
if %1 == Off goto no
if %1 == oFF goto no
if %1 == oFf goto no
if %1 == ofF goto no
helpmsg %1
goto exit
:msg
helpmsg
goto exit
:yes
prompt $i[$p]
goto exit
:no
cls
prompt
:exit
```

Here is the equivalent program in Rexx:

Example 6.5. Rexx program

```
/* help.rex -- Get help for a system message */
arg action .
select
  when action=""    then     "helpmsg"
  when action="ON"  then     "prompt $i[$p]"
  when action="OFF" then do
    "cls"
    "prompt"
  end
  otherwise "helpmsg" action
end
exit
```

6.3. Using Variables to Build Commands

You can use variables to build commands. The **`showfile.rex`** program is an example. **`showfile`** types a file that the user specifies. It prompts the user to enter a file name and then builds a variable containing the *TYPE* command and the input file name.

To have Rexx issue the command to the operating system, put the variable containing the command string on a line by itself. Rexx evaluates the variable and passes the resultant string to Windows:

Example 6.6. showfile.rex (Windows)

```
/* showfile.rex - build command with variables  */
/* prompt the user for a file name             */
say "Type a file name:"

/* assign the response to variable FILENAME   */
pull filename

/* build a command string by concatenation    */
commandstr = "TYPE" filename

/* Assuming the user typed "demo.txt,"        */
/* the variable COMMANDSTR contains           */
/* the string "TYPE DEMO.TXT" and so...       */

commandstr           /* ...Rexx passes the    */
                     /* string on to Windows  */
```

Rexx displays the following on the screen when you run the program:

```
[C:\]rexx showfile
Type a file name:
demo.txt

This is a sample text file. Its sole
purpose is to demonstrate how
commands can be issued from Rexx
programs.

[C:\]
```

6.4. Using Quotation Marks

The rules for forming a command from an expression are the same as those for forming expressions. Be careful with symbols that are used in Rexx and Windows programs. The **`dir.rex`** program below shows how Rexx evaluates a command when the command name and a variable name are the same:

Example 6.7. dir.rex (Windows)

```
/* dir.rex - assign a value to the symbol DIR  */
say "DIR command using Rexx"
dir = "echo This is not a directory."

/* pass the evaluated variable to Windows         */
dir
```

Because dir is a variable that contains a string, the string is passed to the system. The *DIR* command is not executed. Here are the results:

```
[C:\]rexx dir.rex
DIR command using Rexx:
This is not a directory.
[C:\]
```

Rexx evaluates a literal string--a string enclosed in matching quotation marks--exactly as it is. To ensure that a symbol in a command is not evaluated as a variable, enclose it in matching quotation marks as follows:

Example 6.8. Passing values to command environments

```
/* assign a value to the symbol DIR          */
say "DIR command using Rexx"
dir = "echo This is another string now."

/* pass the literal string "dir" to Windows */
"dir"
```

Rexx displays a directory listing.

The best way to ensure that Rexx passes a string to the system as a command is to enclose the entire clause in quotation marks. This is especially important when you use symbols that Rexx uses as operators.

If you want to use a variable in the command string, leave the variable outside the quotation marks. For example:

```
extension = "BAK"
"delete *." || extension

option = "/w"
"dir" option
```

6.5. ADDRESS Instruction

To send a command to a specific environment, use this format of the ADDRESS instruction:

```
ADDRESS environment expression
```

For *environment* specify the destination of the command. To address the Windows environment, use the symbol CMD. For *expression*, specify an expression that results in a string that Rexx passes to the environment. Here are some examples:

Example 6.9. ADDRESS instruction

```
address CMD "dir"      /* pass the literal string      */
                       /* "dir" to Windows             */

address "bash" "ls"    /* pass the literal string      */
                       /* "ls" to the Linux bash shell */

cmdstr = "dir *.txt"   /* assign a string              */
                       /* to a variable                */

address CMD cmdstr     /* Rexx passes the string       */
                       /* "dir *.txt" to Windows       */
address edit "rain"    /* Rexx passes the "rain"       */
                       /* command to a fictitious      */
                       /* environment named edit       */
```

Notice that the ADDRESS instruction lets a single Rexx program issue commands to two or more environments.

The ADDRESS instruction allows to redirect the command's standard input (stdin), standard output (stdout) and standard error (stderr) files directly to Rexx. The following example will define a command to list all environment variables of the current process, sort it on Unix operating systems and redirect the standard output line by line to a Rexx array which then gets listed by the Rexx program in a DO loop.

Example 6.10. Redirecting the standard output to a Rexx array:

```
parse source os .                /* get operating system name                 */

   /* define external command to list all environment variables              */
if os~startsWith("Win") then command='set'   /* Windows (sorted by default)   */
                        else command='env | sort -f'  /* Unix (needs sorting) */

vars=.array~new   /* array to store command's output (environment variables)  */

   /* let the operating system shell execute the command, redirect            */
   /* command's output (stdout) line by line to our Rexx vars array           */
address system command with output using (vars)

len=vars~items~length            /* get number of digits/characters           */
if rc=0 then      /* return code 0: indicates command executed successfully   */
do counter i var over vars       /* list environment variables one by one     */
   say "#" i~right(len)":" var   /* show value of counter i right adjusted    */
end

say "operating system:" os', command was: "'command'", # vars:' vars~items
```

Running the above Rexx program on Darwin (Apple) may yield an output like:

```
#  1: Apple_PubSub_Socket_Render=/private/tmp/com.apple.launchd.BNhGgJRBpz/Render
#  2: HOME=/Users/some_username
#  3: LC_CTYPE=UTF-8
... cut ...
# 20: XPC_FLAGS=0x0
# 21: XPC_SERVICE_NAME=0
# 22: _=/usr/bin/env
operating system: DARWIN, command was: "env | sort -f", # vars: 22
```

Running the above Rexx program on Linux may yield an output like:

```
#  1: CLUTTER_BACKEND=x11
#  2: CLUTTER_IM_MODULE=xim
#  3: COLORTERM=truecolor
... cut ...
# 77: XDG_SESSION_TYPE=x11
# 78: XDG_VTNR=7
# 79: XMODIFIERS=@im=ibus
operating system: LINUX, command was: "env | sort -f", # vars: 79
```

Running the above Rexx program on Windows may yield an output like:

```
#  1: ALLUSERSPROFILE=C:\ProgramData
#  2: APPDATA=C:\Users\Administrator\AppData\Roaming
#  3: asl.log=Destination=file
... cut ...
# 50: windir=C:\WINDOWS
# 51: windows_tracing_flags=3
# 52: windows_tracing_logfile=C:\BVTBin\Tests\installpackage\csilogfile.log
operating system: WindowsNT, command was: "set", # vars: 52
```

6.6. Using Return Codes from Commands

With each command it processes, Windows and Unix/Linux command shells produce a number called a return code. When a Rexx program is running, this return code is automatically assigned to a special built-in Rexx variable named *RC*.

If the command was processed without problems, the return code is almost always **`0`**. If something goes wrong, the return code issued is a nonzero number. The number depends on the command itself and the error encountered.

This example shows how to display a return code:

Example 6.11. getrc.rex (Windows)

```
/* getrc.rex report */
"TYPE nosuch.fil"
say "the return code is" RC
```

The special variable *RC* can be used in expressions like any other variable. In the next example, an error message is displayed when the *TYPE* command returns a nonzero value in *RC:*

Example 6.12. RC special variable

```
/* Simple if/then error handler */
say "Type a file name:"
pull filename
"TYPE" filename
if RC \= 0
then say "Could not find" filename
```

This program tells you only that the system could not find a nonexistent file.

A system error does not stop a Rexx program. Without some provision to stop the program, in this case a trap, Rexx continues running. You might have to press the Ctrl+Break key combination to stop processing. Rexx includes the following instructions for trapping and controlling system errors:

- CALL ON ERROR
- CALL ON FAILURE
- SIGNAL ON ERROR
- SIGNAL ON FAILURE

6.7. Subcommand Processing

Rexx programs can issue commands or subcommands to programs other than Windows. To determine what subcommands you can issue, refer to the documentation for the application.

6.8. Trapping Command Errors

The most efficient way to detect errors from commands is by creating condition traps, using the SIGNAL ON and CALL ON instructions, with either the ERROR or the FAILURE condition. When used in a program, these instructions enable, or switch on, a detector in Rexx that tests the result of every command. Then, if a command signals an error, Rexx stops usual program processing, searches the program for the appropriate label (ERROR:, or FAILURE:, or a label that you created), and resumes processing there.

SIGNAL ON and CALL ON also tell Rexx to store the line number (in the Rexx program) of the command instruction that triggered the condition. Rexx assigns that line number to the special variable SIGL. Your program can get more information about what caused the command error through the built-in function CONDITION.

Using the SIGNAL and CALL instructions to handle errors has several advantages; namely, that programs:

- Are easier to read because you can confine error-trapping to a single, common routine
- Are more flexible because they can respond to errors by clause (*SIGL*), by return code (*RC*), or by other information (CONDITION method or built-in function)
- Can catch problems and react to them before the environment issues an error message
- Are easier to correct because you can turn the traps on and off (SIGNAL OFF and CALL OFF)

For other conditions that can be detected using SIGNAL ON and CALL ON, see the *Open Object Rexx: Reference.*

6.8.1. Instructions and Conditions

The instructions to set a trap for errors are SIGNAL and CALL. Example formats are:

```
SIGNAL ON condition NAME trapname
CALL   ON condition NAME trapname
```

The SIGNAL ON instruction initiates an exit subroutine that ends the program. You use CALL ON to recover from a command error or failure. Both SIGNAL ON and CALL ON will cause a branch to the specified (or default) trapname whenever the condition occurs. With CALL ON you can RETURN to the clause following the one that raised the condition. With SIGNAL ON processing continues with the instructions following the trapname (maybe some cleanup or diagnostic code usually but not necessarily followed by EXIT).

The command conditions that can be trapped are:

ERROR
: Detects any nonzero error code the default environment issues as the result of a Rexx command.

FAILURE
: Detects a severe error, preventing the system from processing the command.

A failure, in this sense, is a particular category of error. If you use SIGNAL ON or CALL ON to set a trap only for ERROR conditions, then it traps failures as well as other errors. If you also specify a FAILURE condition, then the ERROR trap ignores failures.

With both the SIGNAL and the CALL instructions, you can specify the name of the trap routine. Add a NAME keyword followed by the name of the subroutine. If you do not specify the name of the trap routine, Rexx uses the value of *condition* as the name (Rexx looks for the label ERROR:, FAILURE:, and so on).

For more information about other conditions that can be trapped, see the *Open Object Rexx: Reference.*

6.8.2. Disabling Traps

To turn off a trap for any part of a program, use the SIGNAL or CALL instructions with the OFF keyword, such as:

Example 6.13. SIGNAL

```
SIGNAL OFF ERROR
SIGNAL OFF FAILURE
CALL OFF ERROR
CALL OFF FAILURE
```

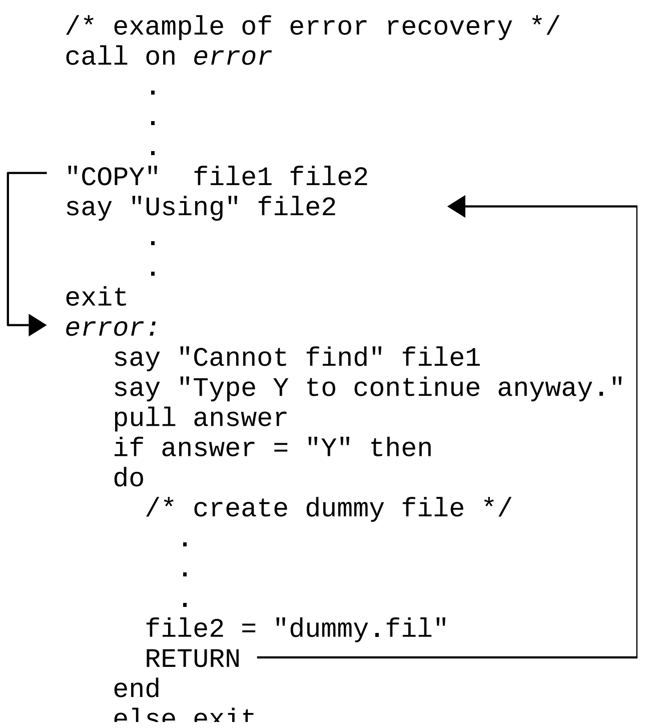
6.8.3. Using SIGNAL ON ERROR

The following example shows how a program can use SIGNAL ON to trap a command error in a program that copies a file. In this example, an error occurs because the name of a nonexistent file is stored in the variable file1. Processing jumps to the clause following the label ERROR:

```
/* example of error trap                                   */
signal on error                 /* Set the trap.           */
     .
     .
     .
"COPY"  file1 file2             /* When an error occurs... */
     .
     .
exit
error:                          /* ...REXX jumps to here   */
  say "Error" rc "at line" sigl
  say "Program cannot continue."
  exit                          /* and ends the program.   */
```

6.8.4. Using CALL ON ERROR

If there were a way to recover, such as by typing another file name, you could use CALL ON to recover and resume processing:



```
/* example of error recovery */
call on error
     .
     .
     .
"COPY"  file1 file2
say "Using" file2
     .
     .
exit
error:
   say "Cannot find" file1
   say "Type Y to continue anyway."
   pull answer
   if answer = "Y" then
   do
     /* create dummy file */
       .
       .
       .
     file2 = "dummy.fil"
     RETURN
   end
   else exit
```

6.8.5. A Common Error-Handling Routine

The following example shows a simple error trap that you can use in many programs:

Example 6.14. Common error handling routine

```
/* Here is a sample "main program" with an error             */
signal on error        /* enable error handling              */
"ersae myfiles.*"      /* mistyped "erase" instruction       */
exit

/* And here is a fairly generic error handler for this       */
/* program (and many others...)                              */
error:
say "error" rc "in system call."
say
say "line number =" sigl
say "instruction = "  sourceline(sigl)
exit
```

Input and Output

Rexx supports a stream I/O model. Using streams, your program reads data from various devices (such as hard disks, CD-ROMs, and keyboards) as a continuous stream of characters. Your program also writes data as a continuous stream of characters.

In the stream model, a text file is represented as a stream of characters with special line-end characters marking the end of each line of text in the stream.

We use the expression "line-end characters" because they are platform dependent, and may be a single character or a character pair. UNIX, Linux and Darwin use a single line feed character **`'0a'x`**, Windows uses a carriage-return and line-feed character pair **`'0d 0a'x`**. Henceforth, when we use the expression "line-end characters", we mean whatever character sequence constitutes the line termination sequence on whichever platform you are working with Rexx on.

A binary file is a stream of characters without an inherent line structure. A Rexx stream object allows you read from a data stream using either the text-file line methods or using a continuous data stream method.

The Rexx Stream class is the mechanism for accessing I/O streams. To input or output data, you first create an instance of the Stream class that represents the device or file you want to use. For example, the following clause creates a stream object for the file **`out.dat`**:

```
/* Create a stream object for out.dat */
file = .Stream~new("out.dat")
```

Then you use the appropriate stream methods to read and/or write the data. **`out.dat`** is a text file, so you would normally use the LINES, LINEIN, and LINEOUT methods that read or write data as delimited lines. If the stream represents a binary file (such as a **`wav`**, **`gif`**, or **`tif`**), you would use the CHARS, CHARIN, and CHAROUT methods that read and write data as characters.

The Stream class includes additional methods for opening and closing streams, flushing buffers, seeking to specific file locations, retrieving stream status, and other I/O operations.

7.1. More about Stream Objects

To use streams in Rexx, you create new instances of the Stream class. These stream objects represent the various data sources and destinations available to your program, such as hard disks, CD-ROMs, keyboards, displays, printers, serial interfaces or devices on a network.

Stream objects can be transient or persistent. An example of a transient (or dynamic) stream object is a serial interface. Data can be sent or received from serial interfaces, but the data is not stored permanently by the serial interface itself. Consequently, you cannot, for example, read from or write to a random position in the data stream–it can only be read and/or written as a sequential stream of characters. Once you read from or write to the stream, the data cannot be re-accessed.

A disk file is an example of a persistent stream object. Because the data is stored on disk, you can search forward and backward in the stream and reread data that you have previously read. Rexx maintains separate read and write pointers to a stream that you can move independently of each other using arguments on methods such as LINEIN, LINEOUT, CHARIN, and CHAROUT. The Stream class also provides SEEK and POSITION methods for setting the read and write positions.

7.2. Reading a Text File

The following shows an example of reading a file. Program count.rex counts the words in a text file. To run it, enter **`rexx count`** followed by the name of the file to be processed:

```
rexx count myfile.txt
rexx count /rexx/articles/devcon7.scr
```

count.rex uses the String method WORDS to count the words, so count.rex actually counts whitespace-delimited tokens:

Example 7.1. count.rex with automatic open

```
/* count.rex - counts the words in a file */
parse arg filename            -- get file name from command line
count = 0                     -- initialize a counter
file = .Stream~new(filename)  -- create a stream object for the file
do file~lines                 -- loop for the number of lines
  text = file~lineIn          -- read a line from the file
  count += text~words         -- count words and add to counter
end
say count                     -- display the count
```

To read a file, count.rex first creates a Stream object for the file by sending the NEW message to the Stream class. The file name (with or without a path) is specified as an argument on the NEW method.

Within the DO loop, count.rex reads the lines of the file by sending LINEIN messages to the stream object (pointed to by the variable File). The first LINEIN message causes Rexx to automatically open the file if it was not already open (the NEW method does not open the file). LINEIN, by default, reads one line from the file, starting at the current read position.

Rexx returns only the text of the line to your program without any line-end characters.

The DO loop is run **`file~lines`** times. The LINES method returns the number of lines in the file, so Rexx processes the loop until no lines remain to be read.

In the count.rex program, the LINEIN request forces Rexx to open the file, but you can also open the file yourself using the OPEN method of the Stream class. By using the OPEN method, you control the mode in which Rexx opens the file. When Rexx implicitly opens a file because of a LINEIN request, it tries to open the file for both reading and writing. If that fails, it tries to open the file for reading. To ensure that the file is opened only for reading, you can modify count.rex as follows:

Example 7.2. count.rex with explicit open

```
/* count.rex - counts the words in a file */
parse arg filename            -- get file name from command line
count = 0                     -- initialize a counter
file = .Stream~new(filename)  -- create a stream object for the file
status = file~open("read")    -- open the file for reading
if status <> "READY:" then do -- check the open status
  say "Could not open" filename":" file~description
  exit
end
do file~lines                 -- loop for the number of lines
  text = file~lineIn          -- read a line from the file
  count += text~words         -- count words and add to counter
end
file~close                    -- close the file
say count                     -- display the count
```

The CLOSE method, used near the end of the previous example, closes the file. A CLOSE is not required. Rexx closes the stream for you when the program ends. However, it is a good idea to CLOSE streams to make the resource available for other uses.

7.3. Reading a Text File into an Array

Rexx provides a Stream method, named ARRAYIN, that reads the contents of a stream into an array object. ARRAYIN is convenient when you need to read an entire file into memory for processing. You can read the entire file with a single Rexx clause–no looping is necessary.

The following example (cview.rex) uses the ARRAYIN method to read the entire log.txt file into an array object. cview.rex displays selected lines from log.txt. A search argument can be specified when starting cview.rex:

```
rexx cview libpath
```

cview.rex prompts for a search argument if you do not specify one.

If cview.rex finds the string, it displays the line(s) in which the string was found. cview.rex continues to prompt for a new search string until you enter **`Q`** in response to the prompt.

Example 7.3. cview.rex

```
/* cview.rex - display lines from file log.txt */
parse arg search                -- get search string from command line
file = .Stream~new("log.txt")   -- create stream object
lines = file~arrayIn            -- read file into LINES array object
loop
  if search = "" then do        -- prompt for user input
    say "Enter a search string or Q to quit:"
    parse pull search
    if search = "Q" then exit
  end
  loop line over lines          -- scan the array
    if line~caselessContains(search) then
      say line                  -- display any line that matches
  end
  search = ""                   -- reset for next search
end
```

7.4. Reading Specific Lines of a Text File

You can read a specific line of a text file by entering a line number as an argument on the LINEIN method. In this example, line 3 is read from **`log.txt`**:

Example 7.4. LINEIN

```
/* Read and display line 3 of file log.txt */
infile = .Stream~new("log.txt")
say infile~lineIn(3)
```

You do not reduce file I/O by using specific line numbers. Because text files do not have a specific record length, Rexx must read through the file counting occurrences of line-end characters to find the line number you want.

7.5. Writing a Text File

To write lines of text to a file, you use the LINEOUT method. By default, LINEOUT appends to an existing file. The following example adds an item to a to-do list that is maintained as a simple text file:

Example 7.5. todo.rex

```
/* todo.rex - add to a todo list */
parse arg text
file = .Stream~new("todo.dat")    -- create a Stream object
file~lineOut(date() time() text)  -- append a line to the file
exit
```

In todo.rex, a text string is provided as the only argument on LINEOUT. Rexx writes the line of text to the file with line-end characters appended. You do not have to provide line-end characters in the string to be written.

If you want to overwrite a line, specify a line number as a second argument to position the write pointer:

```
file~lineOut("13760-0006", 35)    -- Replace line 35
```

Rexx does not prevent you from overwriting existing line-end characters in the file. Consequently, if you want to replace a line of the file without overlaying the following lines, the line you write must have the same length as the line you are replacing. Writing a line that is shorter than an existing line leaves part of the old line in the file.

Also, positioning the write pointer to line 1 does not replace the file. Rexx starts writing over the existing data starting at line 1, but if you happen to write fewer bytes than previously existed in the file, your data is followed by the remainder of the old file.

To replace a file, use the OPEN method with WRITE REPLACE or BOTH REPLACE as an argument. In the following example, a file named **`temp.dat`** is replaced with a random number of lines. **`temp.dat`** is then read and displayed. You can run the example repeatedly to verify that **`temp.dat`** is replaced on each run.

Example 7.6. repfile.rex

```
/* repfile.rex - demonstrates file replacement */
testfile = .Stream~new("temp.dat") -- create a new stream object
testfile~open("both replace")      -- open for read, write, and replace
numlines = random(1, 100)          -- pick a number from 1 to 100
runid = random(1, 9999)            -- pick a run identifier
do i = 1 to numlines               -- write the lines
  testfile~lineOut("Run ID" runid "Line number" i)
end

/*
   Now read and display the file.  The read pointer is already at the
   beginning of the file.  MAKEARRAY reads from the read position to
   the end of the file and returns an array object containing the
   lines.
*/
do line over testfile~makeArray
   say line
end
testfile~close
```

The repfile.rex example also demonstrates that Rexx maintains separate read and write pointers to a stream. The read pointer is still at the beginning of the file while the write pointer is at the end of it.

7.6. Reading Binary Files

A binary file is a file whose data is not organized into lines ending with line-end characters. In most cases, you use the character I/O methods (such as CHARS, CHARIN, CHAROUT) on these files.

Suppose, for example, that you want to read the data in the **`chord.wav`** file into a variable:

Example 7.7. getchord.rex

```
/* getchord.wav - reads chord.wav into a variable */
chordf = .Stream~new("chord.wav")
say "Number of characters in the file" chordf~chars

-- Read the whole WAV file into a single Rexx variable.
-- Rexx variables are only limited by available memory.
mychord = chordf~charin(1, chordf~chars)
say "Number of characters read into variable" mychord~length
```

The CHARIN method returns a string of characters from the stream, which in this case is **`chord.wav`**. CHARIN accepts two optional arguments. If no arguments are specified, CHARIN reads one character from the current read position and then advances the read pointer.

The first argument is a start position for reading the file. In the example, 1 is specified so that CHARIN begins reading with the first character of the file. Omitting the first argument achieves the same result.

The second argument specifies how many characters are to be read. To read all the characters, **`chordf~chars`** was specified as the second argument. The CHARS method returns the number of characters remaining to be read in the input stream receiving the message. CHARIN then returns all the characters in the stream.

7.7. Reading Text Files a Character at a Time

You can use the CHARIN and other character methods on text files. Because you read the file as characters, CHARIN returns the line-end characters to your program. Line methods, on the contrary, do not return the line-end characters to your program. Rexx adds line-end characters to the end of every line written using the LINEOUT method. Text editors typically also add line-end characters. As a convenience, Rexx has an environment symbol name **`.ENDOFLINE`** that returns the line-end characters used by the Stream class on the current system. When reading a text file with CHARIN, interpret the character sequence in **`.ENDOFLINE`** as the end of a line.

As an example, run the following program. It writes lines to a file using LINEOUT and then reads those lines using CHARIN. You can mix line methods and character methods. Rexx maintains separate read and write pointers, so there is no need to close the file or search for another position before reading the lines just written.

Example 7.8. linechar.rex

```
/* linechar.rex - demonstrate line end characters */
file = .Stream~new("test.dat") -- create a new stream object
file~open("both replace")      -- open the file for reading and writing
```

```
do i = 1 to 3                        -- write three lines to the file
  file~lineOut("Line" i)
end

do file~chars                        -- read the file a character at a time
  byte = file~charin                 -- read a character
  decimal = byte~c2d                 -- convert character to a decimal value
  if decimal = 13 then               -- carriage return?
     say "Carriage return"
  else if decimal = 10 then          -- line feed?
     say "Line feed"
  else say byte decimal              -- other character
end
file~close                           -- close the file
```

It is not recommended to use line methods to read binary files. Your binary file might not contain any new-line characters. And, if it did, the characters probably are not meant to be interpreted as new-line characters.

7.8. Writing Binary Files

To write a binary file, you use CHAROUT. CHAROUT writes only the characters that you specify in an argument of the method. CHAROUT does not add line-end characters to the end of the string. Here is an example:

Example 7.9. jack.rex

```
/* jack.rex - demonstrate that CHAROUT does not add line-end characters */
filebin = .Stream~new("binary.dat") -- create a new stream object
filebin~open("replace")             -- open the file for replacement
do i = 1 to 10                      -- write ten strings
  filebin~charOut("All work and no play makes Jack a dull boy. ")
end
                                    -- display the file just created
say filebin~charIn(1, filebin~query("size"))
filebin~close                       -- close the file
```

Because line-end characters are not added, the text displayed by the SAY instruction is concatenated into one contiguous line.

CHAROUT writes the string specified and advances the write pointer. If you want to position the write pointer before writing the string, specify the starting position as a second argument:

```
filebin~charOut("Jack is losing it.", 30) -- start writing at character 30
```

In the example, the file is explicitly opened and closed. If you do not open the file, Rexx attempts to open the file for both reading and writing. If you do not close the file, Rexx closes it when the procedure ends.

7.9. Closing Files

If you do not explicitly close a file, Rexx closes the file when the Stream object is reclaimed by the garbage collector. This frequently does not occur until your program exits, so it is good practice to explicitly close files when you are finished with them.

7.10. Direct File Access

Rexx provides several ways for you to read records of a file directly (that is, in random order). The following example, direct.rex, shows several cases that illustrate some of your options.

direct.rex opens a file for both reading and writing, which is indicated by the BOTH argument of the OPEN method. The REPLACE argument of the OPEN method causes any existing **`direct.dat`** file to be replaced.

The OPEN method also has the arguments BINARY and RECLENGTH, which are useful for direct file access.

The BINARY argument opens the stream in binary mode, which means that line-end characters are ignored. Binary mode is useful if you want to process binary data using line methods. It is easier to use line methods for direct access. With line methods, you can search a position in a file using line numbers. With character methods, you must calculate the character displacement of the file.

The RECLENGTH argument defines a record length of 60 for the file. It enables you to use line methods in a binary-mode stream. Because Rexx now knows how long each record is, it can calculate the displacement of the file for a given record number and read the record directly.

Example 7.10. direct.rex

```
/* direct.rex - demonstration of direct file access */
db = .Stream~new("direct.dat")
db~open("both replace binary reclength 60")

-- Write three records of 60 bytes each using LINEOUT
db~lineOut("Athos, Count de la Fere")
db~lineOut("Porthos, Baron du Vallon de Bracieux de Pierrefonds")
db~lineOut("Rene d'Herblay, alias Aramis")

-- Case 1: Read the records in order using LINEIN.
say "Case 1: Sequential reads with LINEIN..."
do i = 1 to 3
   say db~lineIn
end
say "Press Enter to continue"; parse pull resp

-- Case 2: Read records in random order using LINEIN
say "Case 2: Random reads with LINEIN..."
do i = 1 to 5
   lineno = random(1, 3)
   say "Record" lineno "=" db~lineIn(lineno)
end
say "Press Enter to continue"; parse pull resp

-- Case 3: Read entire file with CHARIN
say "Case 3: Read entire file with a single CHARIN..."
say db~charIn(1, 60 * 3)
say "Press Enter to continue"; parse pull resp

-- Case 4: Read file sequentially with CHARIN
say "Case 4: Sequential reads with CHARIN..."
db~seek("1 read")        -- reposition read pointer
do i = 1 to 3
   say db~charIn(, 60)
end
say "Press Enter to continue"; parse pull resp

-- Case 5: Read records in random order with CHARIN
say "Case 5: Random reads with CHARIN..."
```

```
do i = 1 to 5
   lineno = random(1,3)
   charno = (lineno - 1) * 60 + 1
   say "Record" lineno "Character" charno "=" db~charIn(charno, 60)
end
say "Press Enter to continue"; parse pull resp

-- Case 6: Write records in random order with LINEOUT
say "Case 6: Replace record 2 with LINEOUT"
db~lineOut("This should replace line 2", 2)
do i = 1 to 3
   say db~lineIn(i)
end
say "Press Enter to continue"; parse pull resp

-- Case 7: Write records in random order with CHAROUT
say "Case 7: Replace record 2 with CHARIN..."
db~charout("New record 2 from CHAROUT"~left(60, "."), 61)
db~seek("1 read")        -- reposition read pointer
do i = 1 to 3
   say db~charin(, 60)
end
say "Press Enter to continue"; parse pull resp
db~close
```

After opening the file, direct.rex writes three records using LINEOUT. The records are not padded to 60 characters, Rexx itself handles that. Because the file is opened in binary mode, Rexx does not write line-end characters at the end of each line. It only writes the strings one after another to the stream.

In Case 1, the LINEIN method is used to read the file. Because the file is open in binary mode, LINEIN does not look for line-end characters to mark the end of a line. Instead, it relies on the record length that you specify on open. In fact, if there were a carriage-return or line-feed sequence of the line, Rexx would return those characters to your program.

Case 2 demonstrates how to read the file in random order. In this case, the RANDOM function is used to choose a record to be retrieved. Then the desired record number is specified as an argument on LINEIN. Note that records are numbered starting from 1, not from 0. Because the file is opened in binary mode, Rexx does not look for line-end characters. It uses the RECLENGTH value to determine where to begin reading. The LINEIN method can, therefore, retrieve a line directly, without having to scan through the file counting line-end characters.

Case 3 proves that no line-end characters exist in the file. The CHARIN method reads the entire file. SAY displays the returned string as one long string. If Rexx inserted line-end characters, each record would be displayed on a separate line.

Case 4 shows how to read the binary mode file sequentially using CHARIN. But before reading the file, the read pointer must be reset to the beginning of the file. (Case 3 leaves the read pointer at the end of the file.) The SEEK method resets the read pointer to character 1, which is the beginning of the file. As with lines, Rexx numbers characters starting with 1, not 0. Position 1 is the first character of the file.

By default, the number specified with SEEK refers to a character position. You can also search by line number or by offsets. SEEK allows offsets from the current read or write position (a relative position), or from the beginning or ending of the file (an absolute position). If you prefer typing longer method names, you can use POSITION as a synonym for SEEK.

In the loop of Case 4, the first argument on CHARIN is omitted. The first argument tells where to position the read pointer. If it is omitted, Rexx automatically advances the read pointer based on the number of characters you are reading.

Case 5 demonstrates how to read records in random order with CHARIN. In the loop, a random record number is selected and assigned to the variable **`lineno`**. This record number is then converted to a character number, which can be used to specify the read position on CHARIN. Compare Case 5 with Case 2. In Case 2, which uses line methods, it is not necessary to perform a calculation, you just request the record you want.

Cases 6 and 7 write records in random order. Case 6 uses LINEOUT, while Case 7 uses CHAROUT. Because the file is opened in binary mode, LINEOUT does not write line-end characters. You can write over a line by specifying a line number. With CHAROUT, you need to calculate the character position of the line to be replaced. Unlike LINEOUT, you need to ensure that the string being written with CHAROUT is padded to the appropriate record length. Otherwise, part of the record being replaced remains in the file.

Consequently, for random reading of files with fixed length records, line methods are often the better choice.

7.11. Checking for the Existence of a File

To check for the existence of a file, you use the QUERY method of the Stream class. The following isthere.rex program accepts a file name as a command line argument and checks for the existence of that file.

Example 7.11. isthere.rex

```
/* isthere.rex - test for the existence of a file */
parse arg fid                         -- get the file name
qfile = .Stream~new(fid)              -- create stream object
if qfile~query("exists") = "" then -- check for existence
   say fid "does not exist."
else
   say fid "exists."
```

In the example, a stream object is created for the file even though it might not exist. This is acceptable because the file is not opened when the stream object is created.

The QUERY method accepts one argument. To check for the existence of a file, you specify the string **`exists`** as previously shown. If the file exists, QUERY returns the full-path specification of the stream object. Otherwise, QUERY returns a null string.

7.12. Getting Other Information about a File

The QUERY method can also return date and time stamps, read position, write position, the size of the file, and so on. The following example shows most of the QUERY arguments.

Example 7.12. infoon.rex

```
/* infoon.rex - display information about a file */
parse arg fid
qfile = .Stream~new(fid)
fullpath = qfile~query("exists")
if fullpath = "" then do
   say fid "does not exist."
   exit
```

```
end
qfile~open("both")
say
say "Full path name:" fullpath
say "Date and time stamps (U.S. format):" qfile~query("datetime")
say "            (International format):" qfile~query("timestamp")
say
say "Handle associated with stream:" qfile~query("handle")
say "                  Stream type:" qfile~query("streamtype")
say
say "          Size of the file (characters):" qfile~query("size")
say " Read position (in terms of characters):" qfile~query("seek read")
say "Write position (in terms of characters):" qfile~query("seek write")
qfile~close
```

7.13. Using Standard I/O

All of the preceding topics dealt with the reading and writing of files. You can use the same methods to read from standard input (usually the keyboard) and to write to standard output (usually the display). You can also use the methods to write to the standard error stream. In Rexx, these default streams are represented by public objects of the Monitor class: **`.input`**, **`.output`**, and **`.error`**.

The streams STDIN, STDOUT, and STDERR are transient streams. For transient streams, you cannot use any method or method argument for positioning the read and write pointers. You cannot, for example, use the SEEK method on STDOUT.

Writing to STDOUT has the same effect as using the SAY instruction. However, the SAY instruction always writes line-end characters at the end of the string. By using the CHAROUT method to write to STDOUT, you can control when line-end characters are written.

The following example shows a modified count.rex program previously shown in *Section 7.2, “Reading a Text File”*. count.rex has been modified to display a progress indicator. For every line processed, count.rex now uses CHAROUT to display a single period. count.rex does not write any line-end characters, so the periods wrap to the next line when they reach the end of the line.

Example 7.13. Modified count.rex

```
/* count.rex - counts the words in a file */
parse arg path             -- get the file name
count = 0                  -- initialize the count
file = .Stream~new(path)   -- create a stream object for the input file
do file~lines              -- process each line of the file
  text = file~lineIn       -- read a line
  count += text~words      -- count blank-delimited tokens
  .output~charout(".")     -- write period to STDOUT
end
say
say count
```

Reading from STDIN using LINEIN is similar to reading with the PARSE PULL instruction:

Example 7.14. inexam.rex

```
/* inexam.rex - example of reading STDIN with LINEIN  */

-- Prompt for input with SAY and PARSE instructions
say "What is your name?"
```

```
parse pull response
say "Hi" response
say ""

-- Now prompt using LINEOUT and LINEIN
.output~lineOut("What is your name?")
response = .input~lineIn
.output~lineOut("Hi" response)
```

Using character methods with STDIN and STDOUT gives you more control over the reading and writing of line-end characters. In the following example, the prompting string is written to STDOUT using CHAROUT. Because CHAROUT does not add any line-end characters to the stream, the display cursor is positioned after the prompt string on the same line.

Example 7.15. inchar.rex

```
/* inchar.rex - example of reading STDIN with CHARIN  */
.output~charout("What is your name? ")
response = .input~charin( ,10)
.output~charout("Hi" response)
```

CHARIN is used to read the user's response. The user's keystrokes are not returned to your program until the user presses the Enter key. In the example, a length of 10 is specified. If fewer characters than the specified length are available, CHARIN waits until they become available. Otherwise, the characters are returned to your program. CHARIN does not strip any carriage-return or line-feed characters before returning the string to your program. You can observe this with inchar.rex by typing several strings that have less than ten characters and pressing Enter after each string:

```
What is your name? John
Public
Hi John
Publ
```

7.14. Using Windows Devices

You can use Windows devices by substituting a device name (such as PRN, LPT1, LPT2, COM1, and so on) for the file name when you create a stream object. Then use line or character methods to read or write the device.

The following example sends data to a printer (device name PRN in the example). In addition to sending text data, the example also sends a control character for starting a new page. You can send other control characters or escape sequences in a similar manner. (Generally, these are listed in the manual for the device.)

Usually the control characters are characters that you cannot type at the keyboard. To use them in your program, send a D2C message to the character's ASCII value as shown in the example.

Example 7.16. printit.rex

```
/* printit.rex - Prints a text file */
say "Type file name: "        -- prompt for a file name
pull filename
infile = .Stream~new(filename)
printer = .Stream~new("prn:")
```

```
newpage = 12~d2c              -- form-feed, the page-eject character

-- loop through the input file
do count = 1 to infile~lines
  if printer~lineOut(infile~lineIn) <> 0 then do
    say "Unable to write to PRN:" printer~description
    leave
  end
  -- if the line count is a multiple of 50,
  -- then start a new page by sending the form feed
  if count // 50 = 0 then
    printer~charout(newpage)
end
infile~close
```

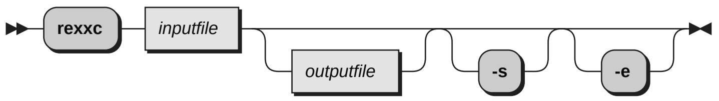
Appendix A. Distributing Programs without Source

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:

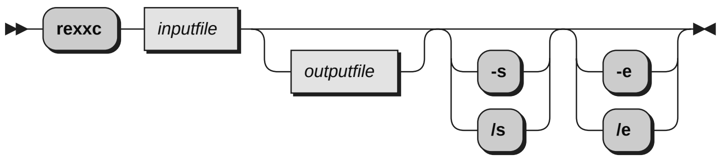
1. The SOURCELINE built-in function returns **`0`** for the number of lines in the program and raises an error for all attempts to retrieve a line.

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 on Unix systems is:



The syntax of the rexxc utility on Windows systems is:



If you specify the *outputfile*, the language processor processes the *inputfile* and writes the executable version of the program to the *outputfile* (a binary file). 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 (*-s* or */s*) to suppress the display of the information about the interpreter used.

You can use the *e* option (*-e* or */e*) to *base64* encode the resulting binary file which makes it possible to deploy it in environments where usually character transformations take place on scripts before Rexx gets them for execution.

Note

Binary files produced by rexxc can be run only on an interpreter of the same bitness and the same (or higher) language level as that of the rexxc that created them.

Appendix B. Sample Rexx Programs

Rexx supplies the following sample programs as .rex , .cls (a package meant to be used via the ::REQUIRES directive) and .frm (a package meant to be used via the ::REQUIRES directive) files.

Platform independent samples

arithmeticEvaluation.rex

: This program demonstrates a number of principles of object-oriented programming.

  This implements a simple expression evaluator that builds a parse tree for an arithmetic expression that can also evaluate the This program demonstrates how to given expression. use reply to run two methods at the same time.

arrayCallback.rex

: This program demonstrates how routines can be used to perform a function over all elements in an array.

ccreply.rex

: A concurrent programming example.

  This program demonstrates how to use reply to run two methods at the same time.

complex.cls

: A complex number class.

  This program demonstrates how to create a complex number class using the ::CLASS and ::METHOD directives. An example of subclassing the complex number class (the Vector subclass) is also shown. Finally, the Stringlike class demonstrates the use of a mixin to provide to the complex number class with some string behavior.

concurrency.rex

: This is a simple demonstration of multi-threaded execution with some synchronization between the threads.

constrained.rex

: Demonstrate how to use the isA method to check for particular types of objects.

delegation.rex

: Demonstrate the concept of a object method delegation, where one object can pass on a method to another object

drives.rex

: A sample use of the Sys... functions.

  This program displays information about drives using the utility functions SysDriveMap, SysDriveInfo, SysFileSystemType, and SysBootDrive.

dynamicMethod.rex

: This program demonstrates how methods can be dynamically added to an object instance.

factorial.rex

: A sample that demonstrates a factorial method routine.

  This sample defines a floating method (one that is not assigned to a class) that implements a factorial algorithm. Using the `.methods` environment symbol this method gets fetched and added under the name factorial to the class named factorial_class, created as a subclass of the String class.

greply.rex
: An example contrasting the GUARDED and UNGUARDED methods.

  This program demonstrates the difference between GUARDED and UNGUARDED methods with respect to their use of the object variable pool.

guess.rex
: An animal guessing game.

  This sample creates a simple node class and uses it to create a logic tree. The logic tree is filled in by playing a simple guessing game.

interface.rex
: This program demonstrates to define an interface class in ooRexx. object instance.

ktguard.rex
: A GUARD instruction example.

  This program demonstrates the use of the START method and the GUARD instruction to control the running of several programs. In this sample, the programs are controlled by one "guarded" variable.

makestring.rex
: A sample to demonstrate a combination of makearray method of the stream and makestring method of the array class.

  The sample creates a temporary file with text, reads in all data using the makearray method of the stream class, removes all comments of the file and writes back data to a temporary file using the makestring method of the array class. After displaying the content of the input and output file, the temporary files are removed from the system.

month.rex
: An example that displays the days of the month **`January 1994`**.

  This version demonstrates the use of arrays to replace stems.

philfork.rex
: Philosophers' Forks: Console window version.

  Sample for concurrency with command line output.

pipe.cls
: A pipeline implementation.

  This program demonstrates the use of the ::CLASS and ::METHOD directives to create a simple implementation of a CMS-like pipeline function.

properties.rex
: This program demonstrates the Properties class.

qdate.rex
: An example that types or pushes today's date and moon phase, in English date format.

qtime.rex
: An example that lays or stacks time in English time format, and also chimes.

rexxcps.rex
: Measuring REXX clauses/second.

scclient.rex
: Simple socket client using socket class (package socket.cls).

scserver.rex
: Simple socket server using socket class (package socket.cls).

semcls.cls
: This Rexx package demonstrates how to implement public semaphore classes (EventSemaphore, MutexSemaphore) in Object Rexx.

sfclient.rex
: Simple socket client using socket function package.

sfserver.rex
: Simple socket server using socket function package.

singleLinkedList.rex
: A simple implementation of a linked list using objects.

sortComposite.rex
: Demonstrates on how to sort non-string objects using the built in sort methods.

stack.rex
: A stack class.

  This program demonstrates how to implement a stack class using the ::CLASS and ::METHOD directives. Also included is a short example of the use of a stack.

synchronousConcurrency.rex
: This uses a work queue to synchronize activity between two independent threads

timezone.rex
: Sample of performing timezone manipulations. Also shows a handy technique for embedding data in program.

treeDirectory.cls
: A directory implementation using a balanced binary tree. Unlike the directory class, indexes are maintained in sorted order.

treeTraversal.rex
: Build a binary tree structure and demonstrate different traversal methods (preorder, inorder, postorder and levelorder traversals).

usecomp.rex
: A simple demonstration of the complex number class.

  This program demonstrates the use of the ::REQUIRES directive, using the complex number class implemented in the package *complex.cls*.

usepipe.rex

Sample uses of the pipe implementation in package *pipe.cls*.

This program demonstrates how you could use the pipes implemented in the package pipe.cls.

usesingleton.rex

Sample use of the metaclass named Singleton, which makes sure that only a single instance of the TestSingleton class can be created as opposed to the Test class, which allows for any number of instances.

usetree.rex

Sample uses of the pipe implementation in package treeDirectory.cls.

This program demonstrates how you could use the treeDirectory class implemented in the package treeDirectory.cls.

Windows only samples

`ole\apps\AOO_scalc_chart.rex`

Using *OLE* create a new *OpenOffice scalc* worksheet, create random data and a chart based on it. Demonstrates how to use *UNO* reflection and incorporate *UNO_CONSTANTS* and *UNO_ENUM* values into a Rexx directory for easier use.

`ole\apps\AOO_simpress_present.rex`

Using *OLE* create a new *OpenOffice simpress* presentation about *REXX* and *ooRexx* and then start it as a presentation. Demonstrates how to use enumeration to enumerate paragraphs of a text and query their *'NumberingLevel'* property.

`ole\apps\AOO_swriter_paragraphs.rex`

Using *OLE* create a new *OpenOffice swriter* document, add paragraphs that get aligned in four different ways. Demonstrates how to use *UNO* reflection and incorporate *UNO_CONSTANTS* and *UNO_ENUM* values into a Rexx directory for easier use.

`ole\apps\AOO_swriter_table.rex`

Using *OLE* create a new *OpenOffice swriter* document, a *TextTable*, a *TextFrame*, paragraphs and apply various formattings.

`ole\apps\Lotus123.rex`

Creates a new spreadsheet in *Lotus 1-2-3* and fills in a table with fictional revenue numbers. The table also contains a calculated field and different styles. A second sheet is added with a 3D chart displaying the revenue data.

`ole\apps\LotusNotes_mail.rex`

Creates a mail message in *Lotus Notes* and sends it to a number of recipients automatically.

`ole\apps\LotusWordPro.rex`

Creates a new document in *Lotus WordPro 97*, enters some text with different attributes, and finally saves and prints the document.

`ole\apps\LotusWordPro_clickHereFields.rex`

Creates a new document in *Lotus WordPro 97* with a provided *Smartmaster*. Fills in some *"Click here"* fields with data prompted by the program or queried from the system. Finally the document is saved to the directory in which this Rexx program is located and is sent to the printer.

`ole\apps\MSAccessDemo.rex`

Demonstrate how one can interact with *Jet (MS Access)* databases using *ActiveX/OLE*.

`ole\apps\MSAccess_contacts.rex`

Demonstrate how one can create a database in *Microsoft Access* using *ActiveX/OLE*.

`ole\apps\MSExcel.rex`

Creates a *Microsoft Excel* sheet, enters some data, and saves it.

`ole\apps\MSExcel_cURL.rex`

Using *OLE* creates a new *Microsoft Excel* worksheet, queries the weather with the *cURL* command and inserts the received data. Using the worksheet temperature data a simple chart gets created.

`ole\apps\MSExcel_usingRexxArray.rex`

Using *OLE* creates a new *Microsoft Excel* instance, creates and places additional sheets, renames sheets, iterates over sheets, fills in the first sheet with generated data from an ooRexx array, then creates a chart from the data.

`ole\apps\MSInternetExplorer_events.rex`

Demonstrates the use of events with Microsoft Internet Explorer:

- Navigate to the RexxLA homepage and disallow the changing of the URL to a page not in that "address space".

`ole\apps\MSInternetExplorer_getMethods.rex`

Example from *"winextensions.pdf"*, *"Example 8.11. OLEObject - getKnownMethods method"*: query and list all known methods from Microsoft Internet Explorer

`ole\apps\MSInternetExplorer_navigate.rex`

Starts *Internet Explorer* and shows the *IBM* homepage. After 10 seconds the *RexxLA* events page will be displayed for 10 seconds.

`ole\apps\MSInternetExplorer_search.rex`

Demonstrates the use of events with Microsoft Internet Explorer:

- Search for the string "Rexx" on the RexxLA Web page, and go randomly to one of the found sites.

`ole\apps\MSInternetExplorer_stockPrice.rex`

Gets the stock price from the RexxLA internet page with *Microsoft Internet Explorer*, and stores it in a Rexx variable.

`ole\apps\MSOutlook.rex`

Demonstrate how one can interact with *Microsoft Outlook* databases using *ActiveX/OLE*.

`ole\apps\MSOutlook_monitorInbox.rex`

Example from *"winextensions.pdf"*, *"Example 8.3. OLEObject - Monitor Outlook"*: open *Microsoft Outlook*, attach an ooRexx *"ItemAdd"* event listener to monitor the inbox for adding new mails. The program takes commands by watching every second whether one of the files **`pause.monitor`**, **`restart.monitor`** or **`stop.monitor`** got created in the meantime by the user.

`ole\apps\MSPowerPoint_layouts.rex`

Using *OLE* open *Microsoft PowerPoint* and add a title slide with *Title* and *SubTitle* shapes (sections). Append slides with various styles, get slide count and amount of shapes per slide, list names of shapes on each slide. Save *pptx* file with a unique name and close *MS PowerPoint*.

`ole\apps\MSPowerPoint_present.rex`

Using *OLE* create a new *MS PowerPoint* presentation about REXX and ooRexx and then start it as a presentation.

`ole\apps\MSWord_createModify.rex`

Creates a *Microsoft Word* document, enters some text, and saves it. The program then loads the document again and modifies it.

`ole\apps\MSWord_useStyles.rex`

Using *OLE* creates a *Micorsoft Word* document, adds text using various styles and waits for the user to press the *enter* (*return*) key before changing font attributes of the styles named *"Title"* and *"Heading 1"*.

`ole\apps\oleUtils.frm`

Provides some useful utilities for working with the OLEObject class.

`ole\apps\ScriptingFileSystemObject.rex`

Uses the *Windows Script Host FileSystemObject* to obtain information about the drives of the system.

`ole\apps\ShellApplication_listSystemFolders.rex`

Using *OLE* creates a new Windows' *Shell.Application* and queries all Windows system folders.

`ole\apps\WScriptNetwork.rex`

Shows some features of the *Windows Scripting Host Network* object:

- Query computer name, user name
- List network connections for drives and printers

`ole\apps\WScriptShell.rex`

Shows some features of the *Windows Scripting Host Shell* object:

- Query environment string
- List special folders
- Create a shortcut on the desktop
- Use a GUI popup

ole\oleinfo\oleinfo.rex

This application is a "small" browser for OLE objects.

ole\adsi\adsi1.rex

Retrieves information about a computer with *ADSI*.

ole\adsi\adsi2.rex

Gets a user's full name and changes it.

ole\adsi\adsi3.rex

Shows the use of *ADSI* containers.

ole\adsi\adsi4.rex

Shows the use of filters with *ADSI* collections.

ole\adsi\adsi5.rex

Displays namespaces and domains.

ole\adsi\adsi6.rex

Enables you to inspect the properties of an object.

ole\adsi\adsi7.rex

Creates a group, and places several users in it.

ole\adsi\adsi8.rex
: Removes the users and the group that were created in sample adsi7.rex.

ole\methinfo\main.rex
: This application demonstrates the use of the GetKnownMethods method.

ole\wmi\accounts.rex
: This is a demo application for displaying all the accounts of the windows system with *WMI*. It also shows how to display all the properties of a *WMI* object in general.

ole\wmi\services.rex
: This application demonstrates how to list, start, stop, pause, or resume windows services with *WMI*.

ole\wmi\process.rex
: This application displays by means of *WMI* the processes of a windows system that are running.

ole\wmi\osinfo.rex
: This sample script uses a *Windows Management Instrumentation (WMI)* object (*"Win32_OperatingSystem"*) to obtain information about the installed operating system(s).

ole\wmi\sysinfo\sysinfo.rex
: This is a demo application for inspecting some system properties using WMI.

oodialog\winsystem\deskicon.rex
: A WindowsProgramManager class example.

  This sample uses the method addDesktopIcon of the *WindowsProgramManager* class to create a shortcut to a program or an application on the Windows desktop.

oodialog\winsystem\desktop.rex
: This program demonstrates how you could use the WindowsProgramManager class to manipulate program groups and program items.

oodialog\winsystem\displayAnyMenu.rex
: Displays the menu hierarchy for a window that the user picks.

  It uses the functionality of the windowsSystem.frm Rexx package which makes it easy to use the most important functionality of the winsystm.cls and ooDialog.cls Rexx packages. In addition it demonstrates the usage of winSystemDlgs.h and winSystemDlgs.rc files that define Windows graphical user interfaces (GUI).

oodialog\winsystem\displayWindowTree.rex
: Displays the window hierarchy for a top level window that the user picks.

  It uses the functionality of the windowsSystem.frm Rexx package which makes it easy to use the most important functionality of the winsystm.cls and ooDialog.cls Rexx packages. In addition it demonstrates the usage of winSystemDlgs.h and winSystemDlgs.rc files that define Windows graphical user interfaces (GUI).

oodialog\winsystem\eventlog.rex
: A sample use of the WindowsEventLog class.

  This sample demonstrates how to read from and write to the Windows event log using the methods of the WindowsEventLog class.

oodialog\winsystem\getTheWindow.rex

This example shows how to find a window without using the exact window title.

Displays the window containing part of the user supplied window title irrespective of case. It uses the functionality of the winsystm.cls Rexx package.

oodialog\winsystem\menuNotepad.rex

This example displays the hierarchy of the Notepad menu.

It uses the functionality of the windowsSystem.frm Rexx package which makes it easy to use the most important functionality of the winsystm.cls and ooDialog.cls Rexx packages.

oodialog\winsystem\quickShowAllMenus.rex

This program prints out a menu outline of every open window that has a menu.

It uses the functionality of the windowsSystem.frm Rexx package which makes it easy to use the most important functionality of the winsystm.cls and ooDialog.cls Rexx packages.

oodialog\winsystem\registry.rex

This program demonstrates how you could use the WindowsRegistry class to work with the Windows registry.

oodialog\winsystem\windowsSystem.frm

A Rexx package (a collection of public routines and classes) to help work with the winsystm.cls package.

It demonstrates the usage of winSystemDlgs.h and winSystemDlgs.rc files that define Windows graphical user interfaces (GUI).

Appendix C. Notices

C.1. Trademarks

C.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 D, 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 D. 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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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 E. Revision History

Revision 0-0	**Aug 2016**

Initial creation for 5.0

Index