The NetRexx Tutorial

Object Oriented Programming on the Internet

Alpha Internet pre-release

Version v0r0145; Updated 18 May 1998

Pierantonio Marchesini / ETH Zurich

- Review of this book - 7252(bytes)
- Preface - 9272(bytes)

PART ONE

- Basic concepts - 12857(bytes)
- Introduction to NetRexx - 54029(bytes)
- Language Basics - 28894(bytes)
- Operations on Numbers - 54911(bytes)
- Operations on Strings - 64198(bytes)
- Control Structures - 38112(bytes)

PART TWO

- Classes and Objects in NetRexx - 65221(bytes)
- More on NetRexx Classes - 37303(bytes)
- Operations on files - 63608(bytes)
- Threads - 18818(bytes) UPDATE!
- Socket and Networking - 75406(bytes) UPDATE!
- Interface with the system - 32918(bytes)
- Process Control and Exceptions - 18117(bytes)
- Database Operations - 5560(bytes)

PART THREE

- Applets - 5241(bytes)
- Graphical Interfaces - 883(bytes) **EMPTY**
- Advanced Graphics - 851(bytes) **EMPTY**
- Advanced Networking - 37617(bytes)
- Full OOP projects - 2040(bytes)

PART FOUR

- Additional Instructions - 63538(bytes) UPDATE!
- Advanced Algorithms - 14070(bytes)
- NetRexx for REXXers - 19789(bytes) UPDATE!
- Tools - 16214(bytes)
- The xclasses JAR library - 8830(bytes) UPDATE!
- Miscellaneous - 4682(bytes)
- Appendix A: Bibliography - 11470(bytes) UPDATE!
- Appendix I: Installation - 9480(bytes) UPDATE!
- Appendix Z: changes in this file - 2312(bytes)
- Index - 27279(bytes) UPDATE!

NOTE: This HTML version of the book is provided as-is for all those people that cannot use the **.ps** file, since they do not have access to a Poscript Printer.

Last update was done on 18 May 1998 21:47:31(GMT +2).

The NetRexx Tutorial
- Review of this book

```
DISCLAIMER:

ALL THE  EXAMPLES  PRESENTED  IN THIS BOOK HAVE BEEN TESTED  ON
SEVERAL PLATFORMS.
THIS DOCUMENT   IS   PROVIDED ON  AN  'AS-IS' BASIS. THE AUTHOR
TAKES  NO  RESPONSABILITY  FOR ERRONEOUS, MISSING OR MISLEADING
INFORMATION, OR  FOR  ANY LOSS OF DATA, BUSINESS  OR  HARDWARE,
DUE  TO  THE  USE OF ANY  INFORMATION  OR  CODE  GIVEN  IN THIS
BOOK.
```


You can get a copy of the latest version of this document
from **mpie@ch.ibm.com**.


```
#    #    ##    #####   #    #     #    #    #   ####
#    #   #  #   #    #  ##   #     #    ##   #  #    #
#    #  #    #  #    #  # #  #     #    # #  #  #
# ## #  ######  #####   #  # #     #    #  # #  #  ###
##  ##  #    #  #   #   #   ##     #    #   ##  #    #
#    #  #    #  #    #  #    #     #    #    #   ####
```

This document is available in an as-is format for all the people interested in NetRexx. This document is still in DRAFT form. All the sections marked:

```
***
***  MISSING PART
***
```

should be regarded as on-going or future work.

Look in "Appendix Z" to see the latest changes in the document.

Feel free to send me any comment, question, etc. on this document. My email is ***Pierantonio.Marchesini@cern.ch****.*

English is NOT my mother tongue, as you might have already guessed from those very first sentences. The final book will be corrected (I promise) by a professional editor. If a particular sentence is way too obscure (since I wrote it in my Italian-English) please let me know, and it will be corrected.

A full description of this document current status is available in the next page.

Since, as I said, this is a 'living' document, the following table resumes the status of the various chapters, as they appear in this document. A 0 means that the chapter is still totally empty. A 10 means that the chapter is finished and only corrections are pending.

```
Part One

   - Basic concepts                      4
   - Introduction to NetRexx            7.5
   - Language Basics                    9
   - Operations on Numbers              9
   - Operations on Strings              9.5
   - Control Structures                 8

Part Two

   - Objects, Classes and Interfaces    5
   - Operations on files                 6
   - Sockets and Networking             5
   - System Interface                   5
   - Threads                             2
   - Database Operations                0

Part Three

   - Applets                             1
   - Graphical Interfaces               0
   - Advanced Graphics                  0
   - Advanced WEB server                1
   - Full OOP projects                   0

Part Four

   - Additional Instructions            7
   - More on Algorithms                  4
   - NetRexx for REXXers                4
   - Tools                               1
   - Miscellaneous                      0
```

Review of this book

- **What is NetRexx?** *Quoting NetRexx's author, Mike Cowlishaw, "NetRexx is a programming language derived from both REXX and Java(tm); NetRexx is a dialect of REXX, so it is as easy to learn and use as REXX, and it retains the portability and efficiency of Java." Using NetRexx you can create programs and applets for the Java environment more easily than programming in Java itself. Using NetRexx you rarely have to worry about the different types and numbers that Java requires. The "dirty" job is done by the language for you.*
- **What is REXX?** *REXX is an interpreted language originally developed by IBM in 1979. REXX was designed to be platform-independent and is the procedural language shipped with the operating system both on Mainframe Systems (MVS, VM/CMS) and on Personal Systems (OS/2, Amiga). REXX is available on almost any platform as a product, or as a public domain implementation. Due to its simplicity and ease of use, REXX can be thought as a 'Personal' Language - practical not only for the professional programmer, but also for the occasional one. For*

example, you can use it to quickly test an algorithm before implementation, even when using other languages.

- **To whom is this book addressed?** *This book is addressed both to neophytes and to experienced programmers starting to program on ANY system where the Java JDK is installed. Almost all the programming examples found in this book are taken from 'real-life' situations. Among other useful skills, you will learn how to write: a small routine for randomly accessing a 1.000.000 record file in a few milliseconds, a real client server application using sockets, a 'pocket calculator' with 200 significant digits, and pull-down menus using curses.*
- **What are the covered topics?**
  - *Introduction to the NetRexx language*
  - *Numbers,Strings and Control Structures*
  - *Class and Methods*
  - *Operations on files, sockets and threads*
  - *Applets*
  - *Graphical User Interfaces*
- **Is this a User Guide, a Tutorial or a Reference Manual?** *The answer is "something of all these". In fact, the best definition is probably an "Advanced User Guide with Reference Sections". Previous programming experience is needed in order to fully understand this book, and thus it is NOT a user guide in the true sense of the term. However, I felt it necessary to include reference information for those users who might not have the NetRexx reference book readily available to them. Some chapters also needed amplification, since they describe functions not documented elsewhere.*
- **Where can I find the examples?** *All the examples used in this book are available on Internet via WWW at the URL:*

  `http://wwwcn.cern.ch/news/netrexx/examples`

`File: nr_1.html.`

The NetRexx Tutorial
- Preface

Preface

Introduction.

This book is addressed to both professional programmers and end users who want to learn more about the **NetRexx** language.

NOTE: The following documentation refers to NetRexx version 1.00 and following.

```
*** This section is:
```


```
*** and will be available in next releases
```

When to use NetRexx.

This topic is likely to be a source of endless arguments (both pro and con).

This is my genuine opinion.

PROs:

- NetRexx is very easy to read. You can practically program in English (1)
- there is only ONE NetRexx native data type (the string);
- there is almost no need for special characters (like delimiters, identifiers, etc);
- NetRexx has very powerful features, such as arbitrary numeric precision, parsing, easy string handling, etc.
- NetRexx is not tailored to a particular operating system; the very same code can run on:
  - **Windows 95/NT** (TM),
  - **UNIX (TM) (eg. AIX (TM), HP/UX (TM), IRIX (TM), SunOS (TM), Solaris (TM), etc.),**
  - **OS/2** (TM),
  - **Macintosh** (TM),

  In fact NetRexx will run on any platform that supplies a Java Virtual Machine (JVM) (TM) (more on this later.
- If you are (or were) a FORTRAN, PL/I or PASCAL programmer you will probably find NetRexx closer to your 'way-of-programming' than any other language available for the JVM. NetRexx eases the transition for programmers familiar with "procedural" languages into the object oriented paradigm.

CONs:

- NetRexx is **not** (or at least not **exactly**) Rexx, so Rexx (or Object Rexx) fans will be faced with a "transition period". You cannot get your Rexx code immediately running in NetRexx (as you can do with Object Rexx) unless it is a very simple program.
- NetRexx compiles your program into Java byte-code. The code is then very much slower, in terms of execution, than a native Object Rexx or "classic" Rexx. I've measured up to a order of magnitude slower. This performance problem is due to the Java byte-code running in the JVM and is not an inherant problem with NetRexx; raw Java code is just as slow!

About the examples in this book.

>From the very first chapters, I will present and discuss some 'real' NetRexx program atoms (i.e. code fragments (usually methods)) that you can use in your programs after having learnt the language.

I have noticed that many 'user guides' present as examples, programs you will probably never use again in your life; in fact these programs are often totally useless, brought into existence only so that the author can show particular features of the language involved.

I prefer to give you something 'real'; program atoms you can insert in your code, or programs you can run and use even after having finished with this book. The obvious disadvantage in such an approach is that some constructs may not be entirely clear, since they will only be explained several chapters further on. Please be patient, and do not be concerned about things that, at a particular point in your progress through the book, are not completely understood. You can always come back to them later.

***** This section is:**

***** and will be available in next releases**

Book structure

This book is divided into four parts.

```
Part One
   (Writing simple programs)

   - Basic Concepts
   - Introduction to NetRexx
   - Language Basics
   - Operations on Numbers
   - Operations on Strings
   - Control Structures

Part Two
   (Object Oriented Programming)

   - Objects, Classes and Interfaces
   - Operations on files
   - Sockets and Networking
```

```
    - System Interface
    - Threads
    - Database Operations

Part Three
    (Interfacing with the WEB)

    - Applets
    - Graphical Interfaces
    - Advanced Graphics

Part Four
    (Advanced topics)

    - Additional Instructions
    - More on Algorithms
    - NetRexx for Rexxers
    - Tools
    - Miscellaneous
```

Conventions.

In order to be consistent, a 'standard' is being followed in presenting the various code samples and running examples.

When I show a full program example, the code appears like this:

```
+------------------------------------------------------------------------+
| /*
|  * Code example
|  */
| say
| say 'This is a code example'
| say
| exit 0
+------------------------------------------------------------------------+
                                                                codeex.nrx
```

Resources... [Download the source for the codeex.nrx example]

Line numbers may be used in comments related to the code. You will find the file id at the bottom right-hand corner of the code, making it easier to find the referenced portion of the code if you already have the sample code on your computer.

When referring to only a small piece of a program, the code appears like this:

```
------------------------------------------------------
if test then
  do
    say 'Running in test mode.'
  end
------------------------------------------------------
                                            if example
```

Example sessions are presented like this:

```
......................................................................
rsl3pm1 (201) codeex

test

rsl3pm1 (202) ls -la codeex
-rw-r--r--   1 marchesi system        50 Jan 24 20:03 codeex
rsl3pm1 (203)
......................................................................
                                                                 outex
```

What you should type is written in **bold** characters. The *rsl3pm1 (NNN)* prompts are simply those of the machine from which sample sessions were taken, so just ignore them.

Syntax examples appear as:

```
rc = socket('VERSION')
```

with the method invocation in **bold** characters, and the arguments in *italics*.

Data File Samples appear as:

```
+--------------------------------------------------------------------+
| data file                                                          |
| sample                                                             |
+--------------------------------------------------------------------+
                                                           sample.DATA
```

with, again, the file id at the bottom right-hand corner.

Acknowledments

Several reviewers have helped, by questions and comments, to clarify the aims and exposition of the book. **Mark Hessling** was extremely helpful in reading the preliminary version of the book, which was typeset using his XEDIT-like text editor; THE.

Many thanks also to **Bernard Antoine** and **David Asbury** of the CERN CN division for their help and suggestions.

Summary

Let us now make a resume' of what we have seen so far in this chapter.

***** This section is:**

***** and will be available in next releases**

File: nr_2.html.

Last update was done on 18 May 1998 21:47:33(GMT +2).

The NetRexx Tutorial
- Basic concepts

Basic concepts

In this chapter I'll try to give an overview of all the basic concepts which, in my opinion, are required to fully understand the following chapters.

If you're familiar with the concepts exposed here, you can just jump immediately to the next chapter.

The Java language

Java is an object-oriented programming language developed by Sun Microsystems (TM). A Java program might look like a C or C++ program, due to Java's similarities to those languages. Indeed, Java is not based on C, neither on C++. There has been no effort to make Java compatible with those two languages.

One important point to keep in mind is that **Java was designed with the idea to allow execution of code across a network.**

The main feature of Java is that the COMPILED code is platform independent. To achieve this, Java compiles to an intermediate form; Java byte-code. This code is then interpreted "on-the-fly" by a platform-dependent, Java interpreter.

```
+------------+
|            |
| SOURCE     |
|            |
+------------+
    hello.java
     =
     =     (compilation)
     =     javac hello.java
     V
+------------+
|            |
|JAVA BYTE   |
|CODE        |
+------------+
   hello.class
```

The Java Compiler creates a Java class file, which does not contain any instruction which is architecture dependent. You will not find Pentium, rs6000, MAC, etc. instructions in a class file: you will find code which is understood by a Java Virtual Machine: an interpreter which knows how to translate the Java byte-code into your machine's instructions.

```
+-----------------------+
| JAVA BYTE CODE        | <- architecture independent
```

```
| (class)               |
+-----------------------+
|                       |
|  JAVA VIRTUAL MACHINE | <- architecture dependent
|                       |
+-----------------------+
|//////////////////////|
| MACHINE (PC,MAC,WS)  | <- your machine
|//////////////////////|
+-----------------------+
```

So, the **Java Virtual Machine** is just a special interpreter, that "understands" a class file.

The idea is not new: it was available in UCSD Pascal, and the intermediate code was the so-called **p-code**.

When running code across a network, you must eliminate some of the language features which might allow any malicious code to gain access to your computer. Notably, Java's designers had to take away the concept of a "pointer", largely used in C and C++. Java programs cannot access arbitrary addresses in your machine memory.

Java and the WEB

The capability to interpret Java byte-code is available on most WWW browsers available today.

If an HTML document contains a <class> statement, the browser will fetch the class, (if you don't already have it on your machine) and execute the code. The important thing to stress is that the code, at this point, runs on YOUR machine, not on the server from where you downloaded the HTML document.

```
WWW Browser                        http Daemon
-----------                        -----------
(client)                           (server)

  (URL)
            --(URL request)---->
            <--(HTML doc)------

 ...
 <class >
            --(class request)-->
            <--(class)----------

 CLASS runs
 HERE
```

Java adds **local** interaction to the WEB, and offloads processing from the server to the client.

What's the gain in such an approach? Why not run the code directly in the server side? (like you do whenever you issue a **cgi-bin** command)?

If your application manipulates data and displays it graphically, the Java approach is definitely more efficient, both in terms of reduced network traffic and perceived execution speed.

For example: suppose that your company wants to display several histograms on their WWW home page. You could have the pictures (in **gif** or **jpg** format) stored in the HTML daemon directory. Each time the document is requested, potentially hunders of kilobytes of data is transfered across the network. Using Java, you download the application that implements a histogram viewer, and the data to build the histogram to your machine; usually significantly less data than the pre-built images.

JDK

JDK is an acronym for Java Developer Kit. It is a set of programs that allows you to compile your java code and to execute it (using the Java Interpreter).

The JDK is distributed freely by Sun, and you can download it from Sun's site:

http://java.sun.com

See the Appendix I for more details.

The JDK is made up by the following tools:

- a compiler *javac*
- a debugger
- an interpreter, or, if you prefer, a Java Virtual Machine (Java VM) *java*
- an applet viewer *appletviewer*
- other miscellaneous tools

The JDK also includes all the Java class files that you need to compile and run your java programs.

The JDK is NOT a visual development environment, like Microsoft's J++ or Symatec's Cafe'. Sun's JDK has been defined as "primitive" [GREHAN, 1997] by some authors, since all the package's tools run from the command line. Other people [HAROLD, 1997] definitely prefer JDK's "minimalist" approach vs. more fancy products, sometimes still in beta test.

If you are an "old fashion" programmer like me, you'll probably prefer JDK's approach, which resembles the development process I followed on VM/370 and VS/COBOL; edit, compile, and run all from the command line.

For NetRexx there is no IDE at the moment, so you are forced to use JDK's approach anyway.

Java Classes

Like other languages; notably FORTRAN, Java is a relatively simple language. The power of these languages is derived, not from the language itself, but from the extensibility of the language. Without high level mathematical packages and functions in FORTRAN, you would not be able to do much of any significance. Java, without its Class Libraries is the same.

Applications

An **application** is, generally speaking, a stand-alone program which you launch from the command line. An application has unrestricted access to the host system. An application can read/write files on your system using your access privileges, it can open socket connections with any address, etc.

Technically, a NetRexx application is a NetRexx program that has a main() method, or no method at all (NetRexx will add the main() for you).

Applets

An **applet** is a program which is run in the context of an **applet viewer** or of a **WEB browser**. An **applet** has very limited access to the system where it runs; for example, an applet cannot read files, neither can it establish socket connections to systems other than the one from where the applet was downloaded.

Technically speaking, an applet is a NetRexx class which extends the Java class **java.applet.Applet**.

Javascript

You might have found, in several WEB pages, portions of code that are executed by the browser. This code is written using **javascript**. To make it clear, **javascript** has nothing to do with **java**. The black beverage that you find in fast-foods has nothing to do with the nectar you drink at "La Tazza d'oro" (Via degli Orfani 82, in Rome). People (not the same people, indeed) call both of them coffees, but that's the only thing they share. So Java and Javascript just share (a portion of) the name. "The intersection of Java and Javascript is the empty set." [VAN DER LINDEN, 1997].

Javascript was invented by Netscape Inc., and it is a simple scripting language, imbedded in HTML files. It offers loops and conditional tests.

As an example of Javascript, look at the following code:

```
+------------------------------------------------------------------------+
| <HTML>                                                                 |
| <PRE>                                                                  |
| Here I snoop some info about you:                                      |
|                                                                        |
| <script language= "JavaScript">                                        |
|   <!--                                                                 |
|   var where = document.referrer                                        |
|   var name = navigator.appName                                         |
|   var vers = navigator.appVersion                                      |
|   document.writeln ("You came here from:'"+where+"'.")                 |
|   document.write ("You use:'"+name+"  "+vers+"'.")                     |
|  // -->                                                                |
| </script>                                                              |
| </PRE>                                                                 |
| </HTML>                                                                |
+------------------------------------------------------------------------+
                                                                 jsc.html
```

Just in time Compilers

JavaBeans

JavaBeans is a public specification developed by Sun, in consultation with other vendors and with the Java community. JavaBeans is a component model, which lets you build and use Java-based components.

The **beans** is just a Java class with some additional descriptive information. Why this additional information? Because this information is used to make beans reusable software components, which can be manipulated by building tools. This allows non-programmers, using an **authoring tool**, to assemble an application using the provided components.

Additional sources of information

Java

The "home" of Java is:

http://java.sun.com/

JavaBeans

The first place is definitely

http://splash.javasoft.com/beans/spec.html

contains a good tutorial, and the specifications for JavaBeans 1.0.

You should then look at:

http://www2.hursley.ibm.com/netrexx/nrbean.htm

for the NetRexx implementation.

For more general informations, look at

http://splash.javasoft.com/beans/

Summary

File: nr_4.html.

Last update was done on 18 May 1998 21:47:34(GMT +2).

The NetRexx Tutorial
- Introduction to NetRexx

Introduction to NetRexx

Introduction

In this chapter I try to give a global overview of the NetRexx language, along with a bit of history and some information on how to install and run it, etc. Probably the most interesting part starts from the paragraph ***A Small Journey Around NetRexx****, where I try to develop some small programs, purely with the aim of giving you a "feeling" for this language. You can happily jump straight to this section, and leave all the details for later.*

History of Rexx and NetRexx

Rexx was conceived, designed and developed by Mike Cowlishaw of IBM UK. The original motivation was to replace the then (1979) inadequate IBM command language (JCL and EXEC2). The basic idea was to develop something similar to PL/I, but easier to use. During the last 25 years Rexx developped a large community of users, since IBM was/is shipping it as part of it's major Operating Systems (MVS, VM, OS/2). IBM estimates that there are about 6 millions of Rexx Programmers around the world.

NetRexx was again conceived, designed and developed by Mike Cowlishaw IBM Fellow, in 1996. The motivation is is to create a language easier and simpler than Java, but keeping Java's main advantages.

Like Rexx, NetRexx is a real general-purpose language, tuned for both scripting and application development.

Availability of NetRexx.

The latest versions of NetRexx are available on IBM's WEB site at the following URLs:

`http://www.ibm.com/Technology/NetRexx/nrdown.htm`

USA Server or at

`http://www2.hursley.ibm.com/netrexx/nrdown.htm`

UK Server

On those sites you will find the NetRexx toolkit and the NetRexx Language Reference document, written by Mike

Cowlishaw.

The NetRexx documentation and software are distributed free of charge under the conditions of the IBM Employee-Written Software program.

NetRexx is distributed in 2 formats:

- **zip** format for Windows/95, Windows/NT and OS/2;
- **tar+compress** format for UNIX platforms (like AIX, Solaris, HP/UX, IRIX, Linux, DecOSF, etc.)

Installing NetRexx on your machine.

Prerequisites

In order to install and run NetRexx, you need to have already installed:

- the Java runtime and toolkit (from the 1.x Java development kit)
- a text editor

Installation

Installing NetRexx is an easy process. In a nutshell, you need to:

- download the code using your preferred WEB browser
- unpack the distribution
- install the some files from the distribution inside the Java **bin** and **lib** subdirectories.
- change the CLASSPATH environment variable
- check the installation

You should consult the URL

http://www2.hursley.ibm.com/netrexx/doc-nrinst.htm

for more information about the installation. In Appendix I you'll find some examples of installation.

Additional sources of documentation

You can find additional informations at the URLs:

http://www2.hursley.ibm.com/netrexx/nrlinks.htm

For a collection of applets and classes written in NetRexx look at:

http://www.multitask.com.au/netrexx/fac/

The NetRexx Mailing list archives are at:

http://ncc.hursley.ibm.com/majordomo/IBM-NETREXX/archives/

The IBM redbook devoted to NetRexx can be found at:

http://www.redbooks.ibm.com/SG242216/2216ht.htm

The IBM reference is SG24-2216-0.

A Small Journey Around NetRexx

In this section I present a series of small programs, with which we will increase functionality and complexity. With these examples, I want to give you the 'feel' of NetRexx. Of course, if you are an experienced REXX programmer, you can quickly skip this section and go to the next chapter.

The "Hello, world!" Program.

Here is an example of your first NetRexx program, which you can call 'hello.nrx'.

```
+------------------------------------------------------------------------+
| -- Our first NetRexx Program                                           |01
| --                                                                     |02
| say 'Hello World!'                                                     |03
| exit 0                                                                 |04
+------------------------------------------------------------------------+
                                                                 hello.nrx
```

Resources... [Download the source for the hello.nrx example]

The third line contains a print statement to your terminal. Note that you DO NOT need to put a semi-colon (';') at the end of a line. You need one only if you want to put two or more statements on the same line, like it would be for:

```
say 'Hello World!'; exit 0
```

In the fourth line, the **exit** statement is not mandatory; this means you can even avoid writing it. But it is indeed good practice always to exit from a program with the **exit** instruction. Even better, exit also with a return code, as in **exit 0**.

To run your program you now need to type:

```
java COM.ibm.netrexx.process.NetRexxC hello
```

If the compilation was successful, you can now run the program typing:

```
java hello
```

Adding some variables

Suppose that you now want to add some variables in your program. An example:

```
+------------------------------------------------------------------------+
| -- another very simple NetRexx program                                 |
| --                                                                     |
| month_name = 'December'                    -- string                   |
| no_of_days = 31                            -- number                   |
| say 'The month of' month_name 'has' no_of_days 'days.'                 |
| exit 0                                                                 |
+------------------------------------------------------------------------+
                                                              simple1.nrx
```

Resources... [Download the source for the simple1.nrx example]

As you see, the variable assignment operation is a very easy one, in NetRexx. You just need to type:

```
variable = value
```

You do NOT need to declare the variable before the assignment. The only important thing to remember is that ALL variables are treated as strings, so the value you want to associate with them MUST go between single quotes ('). You might ask yourself: "Also numbers are treated as strings?". And, yes, also numbers are strings, so it is little wonder that the following example lines are perfectly equivalent:

```
days = 31
days = days + 1

days = '31'
days = days + '1'
```

Of course, as you have seen, you can avoid the (') marks when you deal with numeric quantities.

Asking Questions and Displaying the Result

If you want to make your first program a little more complex, the usual way is to ask a question. Here is the final result:

```
+------------------------------------------------------------------------+
| -- simple2.nrx                                                         |
| -- ask a question and display the answer                               |
| --                                                                     |
```

```
| say 'How many days are in December?'
| answ = ask
| say 'Your answer is' answ'.'
| exit 0
+----------------------------------------------------------------------+
                                                            simple2.nrx
```

Resources... [Download the source for the simple2.nrx example]

The instruction that tells NetRexx to get the input from the keyboard and put it into the variable named 'answ' is:

`answ = ask`

Adding Choices

Well, as it is the program is not really useful: you can answer anything, even a string of characters, and the program blindly accepts the answer. To make the code a little more 'intelligent' we try to distinguish between a good and a bad answer. Here is how: The code:

```
+----------------------------------------------------------------------+
| -- simple3.nrx
| -- ask a question and check the answer
| --
| say 'How many days are in December?'
| answ = ask
| if answ = 31
|   then say 'Correct Answer.'
|   else say 'Wrong Answer.'
| exit 0
+----------------------------------------------------------------------+
                                                            simple3.nrx
```

Resources... [Download the source for the simple3.nrx example]

Guessing the correct answer

Now we want our program to ask another question, in a case where the first has been answered correctly. We allow the user to make mistakes with the second question. The program will continue until a correct answer is given (or the user gets fed-up and hits CNTRL-C!).

```
+----------------------------------------------------------------------+
| /* simple3.nrx
|  * ask a question and check the answer
|  */
```

```
correct_answ = 31
loop forever
  say 'How many days are in December?'
  answ = ask
  if answ = correct_answ
    then
      do
        say 'Correct.'
        leave
      end
    else say 'Wrong Answer. Try again.'
end
exit 0
```

simple4.nrx

Resources... [Download the source for the simple4.nrx example]

More than one correct answer

Suppose we now ask a question for which there is more than one correct answer. We need to get the answer from the user, and test it against a series of good answers. It can be done with this program:

```
/* simple5.nrx
 * verify answer from a list
 */
good_answ = 'APRIL JUNE SEPTEMBER NOVEMBER'
loop forever
  say 'Tell me a month with 30 days.'
  answ = ask                    -- get the input
  parse answ answ .             -- only the 1st word
  answ = answ.upper()           -- uppercase it
  if good_answ.wordpos(answ) = 0
    then
      do
        say 'You said "'answ'". It is wrong.'
        say 'Try again.'
      end
    else
      do
        say 'Correct.'
        leave
      end
end
```

```
22  exit 0
+----------------------------------------------------------------------+
                                                            simple5.nrx
```

Resources... [Download the source for the simple5.nrx example]

There are several new things introduced. Let us look at them: **line 4:** Here we enter a loop from which we will never exit, (**loop forever**). This might seem dangerous, but is not. The instruction **leave** in line 19 gives us an escape path: the only way to exit the loop is to enter a good answer. **lines 7,8,9:** The instructions are meant to "grab the answer, get only the first world, and uppercase it". This will make life much easier later.

In fact, what **parse answ answ .** does is:

```
user types                answw value
------------------------- ------------------------
January                   JANUARY
I don't know              I
February                  FEBRUARY
please, stop it!          PLEASE,
```

NOTE: The lines

```
answ = ask                      -- get the input
parse answ answ .               -- only the 1st word
answ = answ.upper()             -- uppercase it
```

can be written as:

```
parse ask.upper() answ .
```

which is the NetRexx equivalent for the Classical REXX:

```
parse upper pull anws .
```

line 10: The instruction **good_answ.wordpos(answ)** is the key to the program's functioning. It says: Look in the list good_answ and try to find answ. If you find it, tell me its position. Otherwise, tell me 0. Thus, if the answer is wrong, we get 0, and we continue to loop. An alternative way to perform this task as follows:

```
+----------------------------------------------------------------------+
| /* simple6.nrx
|  * verify answer from a list
|  */
| good    = 0
| good[0] = 4
| good[1] = 'APRIL'
| good[2] = 'JUNE'
| good[3] = 'SEPTEMBER'
| good[4] = 'NOVEMBER'
| loop forever
|   say 'Tell me a month with 30 days.'
```

```
11    answ = ask                     -- get the input
12    parse answ answ .              -- only the 1st word
13    answ = answ.upper()            -- uppercase it
14    found = 0
15    loop i = 1 to good[0]
16      if good[i] <> answ then iterate
17      found = 1
18      leave
19    end
20    if found = 0
21      then
22        do
23          say 'You said "'answ'". It is wrong.'
24          say 'Try again.'
25        end
26      else
27        do
28          say 'Correct.'
29          leave
30        end
31  end
32  exit 0
33
+-----------------------------------------------------------------------+
                                                              simple6.nrx
```

Resources... [Download the source for the simple6.nrx example]

In line 04 we initialise an ARRAY to a default value. The initialization practice is not needed, in a program so short as **simple6.nrx**; but it is a must in more complicated programs. This line tells NetRexx: "initialise any **good[]** array variable to **0**."

Classical REXX users will remember the "standard" initialization of a **STEM** variable:

```
good. = 0
```

In lines 05-09, we define the values of **good[]** array. An ARRAY variable is an array of values, and usually (even if it is not mandatory) the 0 element (**good[0]**) contains the information "how many elements are there in this array?". Since there are four elements, **good[0]** is equal to 4. Here is another example of ARRAY:

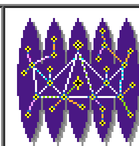
```
variable      value
------------- ------------------------------------
line[0]       3
line[1]       Test line no 1
line[2]       Another one
line[3]       third line
```

If we then want to see if an answer is correct, we need to set a flag (**found**) to FALSE (0) and 'scan' the array until we find the right answer, when we set the flag to TRUE, and exit from the loop (line 14). Then, depending on the value of the flag, we display the appropriate answer as in the previous example. You may have noticed from the length of the two examples that as a rule of thumb it is easier to have data structures in the form of strings than in the form of STEMS Ñ at least when you have very simple entities such as those used in these examples.

More than one list

Suppose you want a program that shows the number of days in a particular month. Since we are lazy, we will not write the full month name, the three first letters are enough. In this case we need two lists: one containing the month names (**month_list**), and another containing, IN THE SAME ORDER, the number of days of the given month (**days_list**).

```
+------------------------------------------------------------------------+
| /* simple7.nrx
  * use two lists
  */
 month_list = 'JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC'
 days_list  = ' 31  28  31  30  31  30  31  31  30  31  30  31'
 good = 0
 loop while good = 0
   say 'Tell me a month (JAN, FEB, etc.)'
   parse ask.upper() answ .
   if month_list.wordpos(answ) <> 0
     then good = 1
     else say 'Wrong, Try again.'
 end
 days = days_list.word(month_list.wordpos(answ))
 say 'Month "'answ'" has' days 'days.'
 exit 0
+------------------------------------------------------------------------+
                                                              simple7.nrx
```



Resources... [Download the source for the simple7.nrx example]

Dealing with files (I)

In the previous example, the two variable **month_list** and **days_list** are long strings. In real life this kind of information is stored in files containing the data used by the program. A file example can be the following:

```
+------------------------------------------------------------+
|* This file contains the month list, with the number        |
|* of days corresponding.                                    |
|*                                                           |
|                                                            |
| January   31                                               |
| February  28                                               |
| March     31                                               |
```

```
| April     30
| May       31
| June      30
| July      31
| August    31
| September 30
| October   31
| November  30
| December  31
+----------------------------------------------------------+
                                                 month.list
```

To make the example a little more interesting, we have added comment lines (all lines starting with an asterisk ("*")) and blank lines. The following program reads the file **month.list** and counts the number of months, printing the total number of months and days in a year.

```
+------------------------------------------------------------------------+
01 -- monthfile.nrx
02 -- test file I/O
03 --
04 infid = xfile('month.list')
05
06 rc = infid.read()
07 if rc <> 0 then
08   do
09     say 'Error reading' infid.name'.'
10     exit 1
11   end
12
13 total  = 0
14 monthl = ''
15 loop i = 1 to infid.lines
16   parse infid.line[i] month days .
17   if month = '' then iterate
18   if month.left(1) = '*' then iterate
19   monthl = monthl month
20   total = total+days
21 end
22
23 say 'There are' monthl.words() 'months.'
24 say 'For a total of' total 'days.'
25 exit 0
+------------------------------------------------------------------------+
                                                            monthfile.nrx
```

Resources... [Download the source for the monthfile.nrx example]

In line '06' we issue a read over the file. All the lines are moved into the STEM list and are ready to process. See below for more information about this instruction. Note line '07': if something is not right (such as the file being non-existent) we exit with an error message. It is always a GOOD IDEA to check return codes from operations that might otherwise disturb the correct functioning of the program. The skipping of the comment and blank lines is done in lines '17' and '18'. **NOTE:** The reading of the file was performed using some instructions:

```
infid = xfile('month.list')  -- define the file
rc = infid.read()            -- issue the read
```

those instructions are **not** part of the native NetRexx, but they are part of an extension package of this book. This extension package is called **xfile** and it should be installed in order to correctly run the example shown above. In a nutshell, you need to:

- grab **xfile.nrx** from the NetRexx Tutorial WWW site;
- compile it;

Look at the "Tools" section for more information about this subject. A tool is also available to compile all the "library" files in an easy way (look for **xbuild**).

"Real" Example no. 1

I don't know about you, but for me this story of months is becoming a bit tedious. I suggest trying a REAL program, which you might even want to write down (or copy from the repository) and use.

Write a tailored finger command.

The standard **finger** UNIX command is a good and simple example of a socket client-server application: a client application **finger** running on your local machine goes to query a server (which runs a **fingerd** daemon) who answers giving a list of the logged on people on the server machine itself.

We will write a simple **finger** client and will format the **fingerd**'s output in a more compact form.

Finger output format

The output of the **fingerd** daemon is in the following format:

```
.....................................................................
rsl3pm1 (201) finger @shift3.cern.ch
(... lines omitted...)
nahn      steven nahn            r31 1:00 Tue 09:01
rattaggi monica rattaggi        r37    5 Tue 09:56
blyth     simon blyth            r38  20: Mon 13:20
blyth     simon blyth            q90   3d Fri 12:21
(... lines omitted...)
rsl3pm1 (203)
.....................................................................
                                          finger command output sample
```

Here I just used the standard UNIX **finger** command, as it is available on any UNIX machine.

Note also that I just showed only few lines. Some systems might have hundreds of lines.

What we want is a more compact output format, which just shows the number of sessions each user has active, and a flag that shows if the inactivity time of a terminal session is less than an hour.

Also, we want to write a program that runs not only on UNIX, but also on WNT, W95, MAC (and I could continue) in a word, on any machine where NetRexx runs.

The full 'xfinger' code.

In the first lines we need some initialisation, like the program version, the author, and some constants, like the port for the finger daemon, and a **Carriage Return - Line Feed** sequence of characters, which are required by the simple fingerd protocol.

```
/* xfinger
 */
VERSION         = 'v1r000'
AUTHOR          = 'P.A.Marchesini, ETHZ'

DEFAULT_PORT    = int 79;
CRLF = '\x0D\x0A'
```

We now get the system we want to talk with. If the user doesn't give one, or he types **-h** or **--help** we give some help.

```
parse arg system
if system = '-h' | system = '--help' | system = '' then
  do
    parse source . . myname'.'
    say myname 'version' VERSION '(c)' AUTHOR
    say 'Purpose : sample implementation of a finger client.'
    say
    say 'java xfinger SYSTEM'
    say
    exit 1;
  end
```

Now comes the "real" fun. We define a socket port (25) and we define it on the **fingerd** PORT (27). Since we need to transfer data over the link, we have to define an INPUT (28) and OUTPUT (29) communication.

```
21 -- issue the client socket command
22 --
23 out = 0
24 j   = 0
25 s = Socket null;
26 do
27   s = Socket(system, DEFAULT_PORT);
28   sin   = DataInputStream(s.getInputStream());
29   sout  = PrintStream(s.getOutputStream());
30   line  = String
31   line  = crlf                      -- retrieve all entries
32   sout.println(line)                -- write msg
33   loop forever
34     line = sin.readLine();
35     if (line = null) then do
36       leave
37     end
```

```
38        j = j+1
39        out[j] = line
40      end
41  catch e1=IOException
42    say 'ERROR:' e1'.'
43  finally
44    do
45      if (s \= null) then s.close()
46    catch e2=IOException
47      say 'ERROR:' e2'.'
48    end
49  end
50  out[0] = j
+-----------------------------------------------------------------------+
```

Now comes a very important point:

If what you are looking for is just an equivalent of the UNIX(tm) ***finger*** *command, then you're already done.*

All you would need at this stage is to output the array **out[]** and, voila', you'd have your nice, working, finger client which runs on all the platforms we saw above, without recompiling!

But we want even more, so let's build a better output, as we discussed.

```
+-----------------------------------------------------------------------+
52  -- order the output, now
53  --
54  sessions = 0
55  active   = '.'
56  users = ''
57  loop i = 2 to out[0]                  -- skip the first line
58    parse out[i] userid . 35 quiet 40 .
59    if quiet = '' then
60      do
61        active[userid] = '*'
62      end
63    if users.wordpos(userid) = 0 then
64      do
65        users = users userid
66      end
67    sessions[userid] = sessions[userid] + 1
68  end
+-----------------------------------------------------------------------+
```

We define a list of users (initialised to the empty string (56)). We also assume that a user is inactive, and we initialize the active array to the inactive status (54). The first line is not interesting, so we loop over the lines starting from the second till the last one (57). We PARSE the line, getting the remote userid, and (after 35 characters) the activity flag (58).

If the flag is empty, than the user is active, so we set the active array to active ("*") for him (59-62). If it's the first time we encounter this user, we add him to the user list (63-66).

Finally, we increment the session counter for him (67).

We've now all the information we need. Let's print it on the screen.

```
+-----------------------------------------------------------------------+
70  -- display the result
71  --
```

```
| oline = ''
|72
| list = users
|73
| loop while list <> ''
|74
|   parse list user list
|75
|
|76
|   item  = user'('sessions[user]','active[user]')'                         |77
|   oline = oline||item.left(14)                                             |78
|   if oline.length() > 80 then                                              |79
|     do
|80
|       say oline
|81
|       oline = ''
|82
|     end
|83
| end
|84
| if oline <> '' then say oline
|85
|
|86
| exit 0
|87
+---------------------------------------------------------------------+
                                                            xfinger.nrx
```

Resources... [Download the source for the xfinger.nrx example]

We get the user list(73). We loop over it, analysing user by user (74-75). We generate an output line, and showing it on the screen when it's longer than 80 characters (77-84).

And finally that's a full output of the command we just created.

```
..................................
rsl3pm1(44) java xfinger shift3
fcot(1,.)      clarei(1,.)    blyth(11,.)    root(1,.)
palit(1,.)     marchesi(1,.) forconi(3,.)   shvorob(6,.)
tully(1,.)     tau(1,.)       braccini(1,.) xujg(2,.)
button(2,.)    filthaut(1,.) fisherp(2,.)   clare(2,.)
oulianov(2,.) pierim(1,.)    malgeril(1,.) gruenew(1,.)
fenyi(1,*)     barczyk(1,.)   graven(2,.)    dsciar(1,.)
passelev(1,.) choutko(2,.)   l3mc1(1,.)     clapoint(1,.)
lodovico(1,.) paus(2,.)      campanel(1,.) l3mc3(1,.)
despixv(1,.)   jessicah(1,.) dmigani(3,.)   lad(1,.)
................................
```

(NOTE: so few active people since it was taken at 2:00 AM 8-))

Real example no. 2

We now write a simple Infix to Polish notation converter, with the purpose of writing a program capable to understand expression of the kind:

```
1 + 5*4 + abs(7-6*2)
```

and write, hopefully, the correct result.

A complete discussion of the problem can be found in KRUSE, 1987, p. 455.

***** This section is:**

***** and will be available in next releases**

Translation from INFIX form to POLISH form.

```
+----------------------------------------------------------------------+
| -- method......: translate
| -- purpose.....: convert an infix tokenized string to a Polish
| --               Notation
| --
|   method translate(intk=Rexx) public static
|
|     -- initialization
|     --
|     valid_tokens = '+ - * / abs'
|     stk = ''                     -- empty stack  (work)
|     pol = ''                     -- output stack
|
|     loop until intk = ''
|       parse intk t intk
|       select
|         when t = '(' then
|           do
|             stk = t stk          -- push()
|           end
|         when t = ')' then
|           do
|             parse stk t stk
|             loop while t <> '('
|               pol = pol t        -- output
|               parse stk t stk    -- pop()
|             end
|           end
|         when valid_tokens.wordpos(t) <> 0 then
|           do
```

```
          loop forever
            if stk = '' then leave
            tk1 = stk.word(1)
            if tk1 = '(' then leave
            if priority(tk1) < priority(t) then leave
            if priority(tk1) = priority(t) & priority(t) = 6
               then leave
            parse stk x stk
            pol = pol x
          end
          stk = t stk
        end
      otherwise
        do
          pol = pol t
        end
    end
  end
  loop while stk <> ''
    parse stk x stk
    pol = pol x
  end
  pol = pol.space()
  return pol

+-----------------------------------------------------------------------+
                                          xstring.nrx(Method:translate)
```

Resources... [Download the complete source for the xstring.nrx library]

Evaluation of Postfix expressions.

This is the evaluation part.

***** This section is:**

***** and will be available in next releases**

```
+-----------------------------------------------------------------------+
| -- method......: evalrpn
| -- purpose.....: evaluates an RPN expression
```

```
--
method evalrpn(intk=Rexx,precision=Rexx) public static

  -- initialization
  --
  if precision = ''
    then precision = 9
  numeric digits precision
  stk = ''                              -- stack


  loop while intk <> ''
    parse intk tk intk
    select
      when 'abs'.wordpos(tk) <> 0 then
        do
          parse stk p1 stk
          select
            when tk = 'abs' then r = p1.abs()
            otherwise NOP
          end
          stk = r stk
        end
      when '+ * - /'.wordpos(tk) <> 0 then
        do
          parse stk p2 p1 stk
          select
            when tk = '+' then r = p1 + p2
            when tk = '-' then r = p1 - p2
            when tk = '*' then r = p1 * p2
            when tk = '/' then r = p1 / p2
            otherwise NOP
          end
          stk = r stk
        end
      otherwise
        do
          stk = tk stk
        end
    end
  end
```

```
      stk = stk.space()
      return stk

+--------------------------------------------------------------------+
                                           xstring.nrx(Method:evalrpn)
```

Resources... [Download the complete source for the xstring.nrx library]

Summary

Here is a resume' of what we have covered in this chapter:

```
Compiling and running a program (on any platform)
-------------------------------------------------
  java COM.ibm.netrexx.process.NetRexxC PROG
  java PROG
  - ex.: java COM.ibm.netrexx.process.NetRexxC hello
         java hello
```

***** This section is:**

***** and will be available in next releases**

File: nr_5.html.

Last update was done on 18 May 1998 21:47:35(GMT +2).

The NetRexx Tutorial
- Language Basics

Language Basics

Introduction

In this chapter we overview some of the NetRexx basics for syntax and structure. To avoid making it too boring, I have tried to make it as short as possible.

Comments

Any sequence of characters delimited by a **'/*'** and a **'*/'** is considered by NetRexx as a comment and will NOT be executed. Also, any sequence of characters following a double - character will be considered as comments (up to the end of line).

Comments can be nested.

```
/*  This is a valid comment */
-- Another comment
```

You are totally free to write the comments as you prefer, but here are some examples:

```
+-----------------------------------------------------------------+
| /**********************************************************/    |
| /*                                                        */    |
| /*   This is one type of comment                          */    |
| /*                                                        */    |
| /**********************************************************/    |
|                                                                 |
| /*                                                              |
|  * This is another type of comment                              |
|  */                                                             |
|                                                                 |
| -- Yet another set of comment                                   |
| -- lines                                                        |
| --                                                              |
+-----------------------------------------------------------------+
```

As a matter of taste I prefer the second style; it also requires less typing effort to add a new line.

- Starting a program with a comment is indeed good programming practice and you should say what the program does and the like. The following is an example of this. It is a bit lengthy, but all this can be built automatically with a program skeleton builder (see **rxtls** in later chapters).

```
/* Program   :  rxtlss
 * Subsystem :  rxt
 * Author    :  P.A.Marchesini (marchesi@shift3.cern.ch).
 * Created   :  4 Dec 1994 on marchesi@shift3.cern.ch
 * Info      :
 * Copyright :  none.
 *
 * Id     Info
 * ------ ----------------------------------------------------
 * v1r000 First release.
 * v1r010 Latest release (see rxtlss.HISTORY file for details)
 *
 */
```

prog2

Blank Lines

Blank lines are ignored. Enough said.

Assignments

We define as *assignment* the operation to store (assign) a value into a variable. The assignment operation is done with the = (equal) sign, as you can see from the following syntax diagram:

```
variable = expression
```

Naturally, what NetRexx does is the following: the *expression* is evaluated, and the result is assigned to the *variable*. Some examples:

```
test = 1
line = 'This is line'

sum = a + b + c
line = 'The sum is:' sum
```

ch0001.nrx

Resources... [Download the source for the ch0001.nrx example](http://www.netrexx.org/Tutorial/nr_6.html)

There are also other types of assignments, using the **parse** instruction, as we will see in later chapters.

Literal Strings

A literal string is a sequence of **any** characters delimited by a single quote character **'** or by a double quote **"**. A NULL string is a string with no (zero) characters in it. Here are some examples:

```
string = 'This is a test'
```

```
| string = "I'm happy to use quotes"                                  |02
| string = 'I"m even more happy now'                                 |03
| string = 'Enough of the "" quotes'                                |04
| string = ''                          /* a NULL string */            |05
+--------------------------------------------------------------------+
                                                      quoteexample.nrx
```

Resources... [Download the source for the quoteexample.nrx example]

NOTE:

- You are free to use single (') or double (") inverted commas. The only recommendation I would like to give is: Be consistent. Once you have adopted one or the other form, always use the same form Ñ at least on the same program Ñ as this is more agreeable for those reading it.
- As you have probably noticed, a double **""** or **"** allows you to put a SINGLE **"** or **'** in a string delimited by the given quote character.

Hexadecimal Strings

A hexadecimal string is a sequence of valid HEX characters (0-9, a-f, A-F), with a '\x' (or '\X' if you prefer).

```
+--------------------------------------------------------------------+
| num1 = '\x00\x01'                                                   |01
| crlf = '\x0D\x0A'   -- Carriage Return & Line Feed                 |02
+--------------------------------------------------------------------+
                                                                 prog6
```

Special Characters

There are few of them in NetRexx, and certain of them have a special meaning when outside a literal string. These are:

```
;  - the delimiter
-  - the continuation character
:  - the label identifier
(  - the start expression
)  - the end expression.
[  - array element (start).
]  - array element (end).
```

Delimiter Character

NetRexx does not need to be told that a statement is ended, as the End-of-Line character automatically implies this, and there is no need to type a **";"** at the end of a line. But if you need to put more than one clause on a line, then you MUST use the **";"** sign.

```
statement_1 ; statement_2 ; statement_3
```

In the following example, note that the three **loop** loops are equivalent:

```
+------------------------------------------------------------------------+
 /* delim_exa.nrx
   */
 loop i = 1 to 10                     -- no delimiter
   say i                              --
 end                                  --

 loop i = 1 to 10;                    -- delimiter
   say i;                             --
 end;                                 --

 loop i = 1 to 10; say i; end;        -- on only one line
 exit 0
+------------------------------------------------------------------------+
                                                              delim_exa.nrx
```

Resources... [Download the source for the delim_exa.nrx example]

Continuation Character

If your NetRexx statement is too long for one line, use the - character to signal to the interpreter that you wish to continue with the next line.

```
statement                           -
 continuation_of_statement          -
 again_continuation_of_statement    -
 termination_of_statement
```

Here is the usual example:

```
+------------------------------------------------------------------------+
 /* cont_exa.nrx
   */
 say 'Very long line'
 say 'Very'   -
     'long'   -
     'line.'
 exit 0
+------------------------------------------------------------------------+
                                                               cont_exa.nrx
```

Resources... [Download the source for the cont_exa.nrx example]

Variables and Constants

A **variable** is an object whose value may be changed during the execution of a NetRexx program. The **value** of a variable is a single character string that can contain **any** character. There are four groups of symbols:

- **constant**
- **simple**
- **arrays**

Constant symbols.

The symbol starts with a digit (0...9) or a period (.). Here are some valid **constant symbols**:

```
82
.92815
3.1415
```

Simple symbols.

The simple symbol does NOT start with a digit (0...9) or a period (.), and does NOT contain a period (.). Here are some valid **simple symbols**:

```
test
pi_Greek
is_it_ok?
```

- **NOTE1:** NetRexx is case insensitive: i.e. the symbols, such as **TEST**, **test**, and **Test** (I could go on, but I'm sure you understood what I mean), all refer to the SAME variable.
- **NOTE2:** An uninitialised variable is automatically trapped by NetRexx at compilation time.

Arrays.

The array is a simple symbol whose last character is a **[**. Here are some valid **arrays**:

```
list[]
a[]
info_test[]
```

As a convention, if indexed by a number the stem contains the same number of items as in its **stem.0** value. This is NOT done by the language itself, but as you will later see, it is useful to use this convention for arrays indexed by integers.

```
variable         value
```

```
--------          ----------------
list[0]           N                          ---+
list[1]           line 1 of list                |
list[2]           second line of list           |
(...)                                           |
list[N]           last line of stem list.    <-+
```

Resume'.

This table is a resume' of what we've seen so far concerning constants and variables. In the first column we see the definition, and in the others what it does and does not have.

```
             DOES                       DOES NOT                   EXAMPLE
             -------------------------  -------------------------  ----------
constant     start with '.' , 0-9       -                          2 , 3.9
simple       -                          start with '.' , 0-9       pippo
array        contain [ ]                -                          list[4]
                                                                   list[1,j]
```

Operations on Arrays.

As we have seen, **arrays** are a special category of variables. Consider the following small program:

```
+----------------------------------------------------------------------+
| -- arrayexa.nrx                                                      |01
| --
|02
| newlist = int[100]                                                   |03
| newlist[1] = 1
|04
| say newlist[1]        -- will print 1
|05
| say newlist[2]        -- will print 0
|06
|
|07
| list     = 'NULL'
|08
| list[2] = 'test'                                                     |09
| say list[1]           -- will print EMPTY
|10
| say list[2]           -- will print test
|11
|
|12
| exit 0
|13
+----------------------------------------------------------------------+
                                                          array_exa.nrx
```

Resources... [Download the source for the array_exa.nrx example]

NOTEs:

- **line 2:**

```
*** This section is:
```


```
*** and will be available in next releases
```

Special Variables

```
*** This section is:
```


```
*** and will be available in next releases
```

Outputting something with say

Use the instruction **say** to output something on your default output character stream (i.e. your screen). The format of the instruction is:

```
say expression
```

Unlike C language, in REXX you do NOT need the newline character ('\n') at the end of your expression; NetRexx automatically does it for you. Examples:

```
list = 'you and me';
total = 200
say 'The list is' list'.'  ->   The list is you and me.
say 'Total is:' total/2    ->   Total is: 100
```

Exiting a program.

Use the instruction **exit** to unconditionally leave a program, and (optionally) return a character string to the caller. The format is:

```
exit expression
```

Example(s):

```
exit 34

if rc <> 0 then
  do
    say 'Unrecoverable error.'
    exit 23
  end
```

As a convention, a program that ends correctly (i.e. with no error) should exit with 0; a non-zero exit code means there has been a problem.

```
exit 0      -> program ended OK
exit <> 0   -> problems
```

different error codes (or messages) might be helpful in understanding what has happened and why the program did not complete correctly.

Warning about Exit Status of UNIX Processes.

The Bourne shell puts the exit status of the previous command in the question mark (?) variable (the C shell uses the **status** variable instead). There is indeed a warning: this variable (**status** or **?**) is a 255 bit (1 byte) value. So if your NetRexx program exits with (for example)

```
exit 300
or:
exit(300)
```

you will get:

```
echo $?      -> 44          (BOURNE shell)
echo $status -> 44          (C shell)
```

This 'feature' should not be underestimated. A user once contacted me to say that his program was aborting in an 'undocumented way', as the $status code he was getting was not in the man page for the program. It took me some time to realize that the return code he was getting (253) was coming from an 'exit -3' instruction.

Getting the arguments from the shell (or input line).

Another important thing you will want to do is to get the arguments from the shell whenever your program is called. In fact, what you will need to do is call a program with 'something' entered on the same line on which you typed the command, and to use this 'something' inside the program. There are several ways with NetRexx to get the arguments used to call that particular program. The simplest is to use a **parse arg** instruction, as in:

```
parse arg variable_name
```

What **parse arg** *variable_name* tells NetRexx is the following: "get the parameters the program was called with, and put them in the variable (a string) called *variable_name*". Consider this simple example:

```
+----------------------------------------------------------------------+
| /* parrot.nrx
|01
|  * echoes back what you type on command line                         |02
|  */
```

```
 parse arg s1
 say 'you said "'s1'".'
 exit 0
+------------------------------------------------------------------+
                                                         parrot.nrx
```

Resources... [Download the source for the parrot.nrx example](#)

This program was called **parrot** for the very simple reason that it 'parrots' back to you whatever you type in in the command line.

```
.....................................................................
rsl3pm1 (401) java parrot toto bello
you said "toto bello".
rsl3pm1 (402) java parrot this is a long line
you said "this is a long line".
rsl3pm1 (404) java parrot `ls tu*`
you said "tu.tu".
rsl3pm1 (405)
.....................................................................
                                                          arg.example
```

Note that what follows the **parse arg**, is not necessarily a variable name: it can be any **parsing template**, as we will see in the chapter concerning string handling. This allows a great flexibility in parameter entering, such as in the following example:

```
+------------------------------------------------------------------+
| /* parsearg.nrx
|   * parses command line input with ONLY 2 fields
|   */
| parse arg infile outfile .
| say 'infile  = "'infile'".'
| say 'outfile = "'outfile'".'
| exit 0
+------------------------------------------------------------------+
                                                       parsearg.nrx
```

Resources... [Download the source for the parsearg.nrx example](#)

What we have told NetRexx is the following: get the input argument **arg**; put the first word in the variable 'infile' **infile**; put the second word in the variable 'outfile' **outfile**; forget about all the rest **"."**. To give you the feel of it, we try it out here:

```
.....................................................................
rsl3pm1 (412) java parsearg test out.TEST
infile  = "test".
outfile = "out.TEST".
rsl3pm1 (413) java parsearg test
infile  = "test".
outfile = "".
rsl3pm1 (414) java parsearg test output.test some other args
infile  = "test".
outfile = "output.test".
```

```
rsl3pm1 (415)
....................................................................
                                                          arg1.example
```

We will get back to parsing in a later chapter (when we'll deal with string operations).

Real Examples

Adding an item to an array (updating array[0])

If you use the convention of having **stem[0]** as the item count for your stem, you need to have a pointer that contains the number of items you have. Suppose that your array is called **list[ ]**. To save the various items in such an array, you will have to build a construct as in the following example:

```
i = 0
do loop
  (...)
  i = i+1
  list[i] = whatever_you_want
end
list[0] = i
```

Here is a better way of doing the same thing:

```
list = xarray()
do loop
  (...)
  list.ad_list whatever_you_want
end
```

We eliminate the need for the index variable **i**, which makes the program: a) easier to read, and b) less error prone since we 'might' for some reason overwrite the pointer variable. This approach is particularly useful for an output file: you build the various lines out output, and then, when you've finished the processing, you can write all the output (contained in the array **list[ ]**) in one go. The following program illustrates this approach. To repeat: in these examples are some new concepts you will find explained later on. You should not spend too much time right now on their details. What I want is to give you are real 'program-atoms' that you can put in your programs even when you have completely mastered the language. **NOTEs:**

- **line 1:** we define an object of the class xarray;
- **line 2:** we add an item;
- **line 3:** we add another one;
- **line 7:** we display the items we collected;

And this is what you will get running the above program:

```
....................................................................
rsl3pm1 (239) java xarray
Line 1
Line 2
Line 3
Line 4 (last)
```

```
rsl3pm1 (240)
.....................................................................
                                              Output of program xarray
```

This chapter's tricks.

Avoid the NEWLINE character.

At this point you might ask yourself: "But what if I do not want to have a NEWLINE?" In that case you cannot use **say**, but rather a small workaround. This is how to do it:

```
str = 'My test'
System.out.print(str'\x0D')
```

Chapter FAQ

QUESTION: *Can comments be nested?* Yes, comments can be nested, so you can happily write something like

```
/*
  (...)
  /* step 1.00
   * start procedure
   */
  (...)
  -- comment
  (...)
 */
```

This feature is useful if you want to comment out a whole piece of code (comments included) to easy you compilation tests.

NOTE: In JAVA comments can NOT be nested.

QUESTION: *How do I do Charin/Charout screen I/O?*

You use the "\-" at the end of string, like in this code atom:

```
say 'This will appear \-'
say 'as one line.'
```

which will print:

```
This will appear as one line.
```

on your terminal.

Summary

Here is a resume' of what we have seen in this chapter.

```
_ comments                          | /*    */
                                    | --
                                    | - ex.: /* this is a comment */
                                    |        -- and this another one
                                    |
_ delimiter character               | ;
                                    | - ex.: say '1' ; say '2'
                                    |
_ continuation character            | -
                                    | - ex.: say 'this is a' -
                                    |            'long line'
                                    |
_ arrays                            | variable[ ]
                                    | - ex.: list[ ]
                                    | - ex.: out[ ]
                                    |
_ reserved variable names           |
                                    |
```

***** This section is:**

***** and will be available in next releases**

File: nr_6.html.

Last update was done on 18 May 1998 21:47:37(GMT +2).

The NetRexx Tutorial
- Operations on Numbers

Operations on Numbers

Introduction

In this chapter we will analyse the basic arithmetic operations that you can perform on numbers. In NetRexx numbers are usually treated as strings of characters (containing digits and, eventually, a '.' sign and/or a '-' sign). This explains the possibility of having arbitrary precision arithmetic, independent of the H/W precision of your machine.

Arithmetic Operations

NetRexx handles the four basic arithmetic operations: Addition, Subtraction, Multiplication and Division. You have also other three special operators to perform Power Operation, Integer Division, and Remainder Division. To perform an arithmetic operation, you simply need to place the appropriate operator between the two terms, and assign what will be the result to a variable. Here is an example of this operation:

```
a = 4 + 5
```

When the Interpreter encounters such an expression, the terms on the right side are evaluated, and the variable (here 'a') will get the final result (which is, as you might suspect, '9'). The following table shows the operations that you can perform on numbers:

```
+           Add.
-           Subtract.
*           Multiply.
/           Divide.
%           Integer divide.
                   (i.e. divide and return the integer part)
//          Remainder.
                   (i.e. divide and return the reminder;
                    this is NOT modulo, as the result may
                    be negative)
**          Power.

-number     (as prefix) same as 0-number.
+number     (as prefix) same as 0+number.
```

Some additional examples:

```
a = 23           /* Assignment */
```

```
b = 1            /* Assignment */

c = a + b        /* Expression */
d = a ** c + 89  /* Expression */
```

Although I believe you may be able to imagine the result of **1+1**, I would like to present some small examples of arithmetic operations. The result is shown in the right column.

```
1+1                  ->  2
1+9                  ->  10

4*7                  ->  28
2**4                 ->  16

(1+2)/3              ->  1
1/3                  ->  0.333333333
4/3                  ->  1.33333333
5/3                  ->  1.66666667

1//3                 ->  1
4//3                 ->  1
5//3                 ->  2

1%3                  ->  0
4%3                  ->  1
5%3                  ->  1
```

The three ways to divide.

A special mention should be devoted to the 'three' divide operators that are used in NetRexx. The / operator performs the regular division. This produces the same result as you would get using the division key on your pocket calculator. If the result is not an integer number, you will get the integer part, a dot and as many digits as the precision is set to (see later in this chapter for considerations about precision). The % operator performs a division and returns ONLY the integer part of the result. Note that the result in NOT rounded (contrary to what I believed at the beginning of my REXX programming). It is simply truncated. The // operator again performs a division, but it returns the remainder. As you have seen in the table, this is NOT a MODULO operation, since the result might be negative. (As you will remember from school, the MODULO is a positive integer). At the risk of being pedantic, I propose a final four examples:

```
a =  13 / 2        a = 6.5
a =  13 % 2        a = 6
a =  13 // 2       a = 1
a = -13 // 2       a = -1
```

Operator Precedence

The operator precedence (or order of evaluation) controls the order in which operations are performed. NetRexx arithmetic uses the same rules you learned in primary school. This table resumes the operator precedence:

```
Precedence  Group                Operators
----------  -----------------    ---------

High        UNARY                + , -
```

```
            POWER               **
            MULTIPLY & DIVIDE   * , / , // , %
Low         ADD & SUBTRACT      + , -
```

As you can imagine, operators with highest precedence are evaluated first, down to the lowest ones.

If you are in doubt.

If you happen to be in doubt about operator precedence, (I sometimes am Ñ especially when dealing with different computer languages), you can use a simple trick: use parentheses. So do not be afraid to write:

```
value = 2 + ( 4 * 32 )
```

instead of the more terse:

```
value = 2 + 4 * 32
```

Of course, you should not use a lot of redundant parentheses inside a loop that is iterated 100 000 times in your program. The first expression in the above example is a little more CPU consuming, but in an average program it is perfectly all right, and saves time that could be lost with bugs.

Other operations on Numbers.

There are many operations you can perform on numbers apart from the ones we have just seen. These operations are performed by NetRexx built-in functions, i.e. functions that are provided by the language itself. You call on those functions in the following way:

```
result = argument.function()
```

as you can see from the example(s):

```
value = -9.abs()
say value          -> 9

max   = -9.max(7)
say max            -> 7
```

This is a table of the NetRexx built-in functions that deal with numbers.

```
number.abs()
  Returns the absolute value of number;

number.d2c()
  Converts the number from Decimal to Character;

number.d2x()
```

```
    Converts the number from Decimal to Hexadecimal;

  number.format()
    Performs a rounding and format over number;

  number.max(number1)
    Returns the largest number from a given list;

  number.min(number1)
    Returns the smallest number from a given list;

  number.sign()
    Returns the sign of a number;

  number.trunc()
    Returns the integer part of a number;
```

I again provide some examples: the right-hand column contains the results of the operations.

```
-2.abs()              ->  2
2.abs()               ->  2

12.min(1)             ->  1
12.min()              ->  12

1.max(42)             ->  42
12.max()              ->  12

-17.sign()            ->  -1
17.sign()             ->  1

n = 23.34
n.trunc(0)             ->  23
n.trunc()              ->  23
n.trunc(3)             ->  23.340
n.trunc(8)             ->  23.34000000

125.d2x()              ->  7F
71.d2c()               ->  G
```

Some of these instructions require a bit of more attention, and we will look at them in the paragraphs that follow.

The format() instruction.

Use the **format** instruction to round and format a number. The syntax of the instruction is:

```
out = format(number,before,after)
```

where **before** and **after** refers to characters before and after the decimal point.

```
number.format(before ,    after )
              ------     -----
                |           |
        (digits before)  (digits after)
                |           |
          -------- -------------
          NNNNNNNN.NNNNNNNNNNNN
                  |
           (decimal point)
```

Suppose that the value of n is "-3.1415". This is what we get for the **format()** instruction:

```
n.format(4,2)      ->    "  -3.14"
n.format(7,5)      ->    "      -3.14150"
n.format(2,3)      ->    "-3.142"
n.format()         ->    "-3.1415"
```

The xmath.random() instruction

As you would expect, the **xmath.random()** function returns a random number. "How random" strongly depends on the implementation of Java. In NetRexx you really get random values, while on VM/CMS you get 'pseudo-random' values. This means that, in the first case, whenever you start a program you get different values; on the contrary, in the second case, the values (although random) are always the same if you do not specify a different seed. The syntax of the instruction is, as we saw:

```
number = xmath.random(max_value)
```

You luckily do not need to modulo the result if you need random values within a certain interval Ñ the 'max_value' parameter will do it for you. A classical application of the random number generator is when you need (for example) to output a cookie message. If you have 150 cookie messages, you do not want to have random numbers greater than 150. All you need to specify, in order to be sure that you do not get values greater than 150, is:

```
ptr = xmath.random(150)
```

A **random(0)** will be accepted, but will generate something that is not really random (the question left to you being "why?"). This is how the **xmath.random()** function is implemented.

```
+------------------------------------------------------------------------+
| -- method......: random                                                |
| -- purpose.....:                                                       |
| --                                                                     |
|                                                                        |
|   method random(max=Rexx) public static;                               |
|     max = max.abs()                                                    |
|                                                                        |
|     n = Math.random() * max                                            |
|     n = n.trunc()                                                      |
|                                                                        |
|     return n                                                           |
|                                                                        |
|                                                                        |
|   method random() public static;                                       |
|     n = random(1000)                                                   |
|     return n                                                           |
|                                                                        |
|                                                                        |
+------------------------------------------------------------------------+
```

xmath.nrx(Method:random)	

Resources... [Download the complete source for the xmath.nrx library]

Comparative operators.

Now that you know how to perform the basic operations on two numbers, you might also want to compare them Ñ i.e. to look at which is larger or smaller, or check if they're equal. More formally, the comparative operators are used to compare two variables (or a variable and a constant) between them. The comparative operators return:

```
1  - if the result of the comparison is true
0  - otherwise
```

NetRexx has two sets of operators: the **normal** comparison and the **strict** comparison. The **strict** comparison is just what its name suggests Ñ two numbers must be **strictly** identical in order to pass the comparison.

```
NORMAL comparative operators:

     =                  True if terms are equal;
     \= , ^=            Not equal;
     >                  Greater than;
     <                  Less than;

     >< , <>            Greater than or less than
                          (same as NOT EQUAL)
     >= , ^< , \<       Greater than or equal to,
                          not less than;
     <= , ^> , \>      Less than or equal to,
                          not greater than;

STRICT comparative operators:

     ==                 True if the terms are strictly equal
                          (identical)
     \== , ^==          True if terms are strictly not
                          equal
     >>                 strictly greater than;
     <<                 strictly less than
     >>= , ^<< , \<<    strictly greater than or equal to,
                          strictly  not less than;
     <<= , ^<< , \>>    strictly less than or equal to,
                          strictly not greater than;

BOOLEAN operators:

     &                  AND;

     |                  Inclusive OR;

     &&                 Exclusive OR;

     ^  , \             LOGICAL NOT
```

We will see how to perform comparisons in the next chapter.

Controlling the precision.

The precision is the number of significant digits used in floating point computations. Roughly speaking, it is the number of digits you are expecting to have after a '.' sign in a floating point number. This table will (I hope) clarify the idea:

```
       value                      precision
       -------------------        ---------
1/3    .333333333                 9
1/3    .333333333333333333       18
1/3    .33333                     5
```

The precision of your arithmetic computations is controlled in NetRexx by the instruction:

```
Numeric Digits [expression]
```

In NetRexx, the default value for precision is 9. In this small program we look how the instructions dealing with precision work:

```
say 1/3                               --    0.333333333
                                      --      ---------
                                      --      9 digits
Numeric Digits 18
say 1/3                               --    0.333333333333333333
                                      --      ------------------
                                      --      18 digits
```

You might now ask: *"why not always run with high precision say, of 100 significant digits?"* The answer is simple: the higher the precision, the slower the program. So use higher precision only when you need it, otherwise keep to the standard one. To make this point even clearer, consider the following small program, which will allow you to measure the performance speed of your machine by changing the precision:

```
-- exercise the precision
parse arg prec .
say 'Running at precision "'prec'".'
numeric digits prec
t1 = timer()
loop i = 1 to 1000
  j = 1/i
  j = j
end
say t1.elapsed()
exit
```

numperf.nrx

Resources... [Download the source for the numperf.nrx example]

To run it, just type **java numperf NNN** where **NNN** is the precision you want Ñ as in the following screen dump:

```
...........................................................
rsl3pm1 (12) java numperf 5
It took 1.001 seconds.
rsl3pm1 (13)
...........................................................
                                                numperf example
```

The following table was built using my **HP Vectra Pentium 133MHz** machine.

```
+----------------------------------------------------+
| timing for 1000 divisions at NNN digits precision |
+----------------------------------------------------+
| NNN                         time                   |
| -----                 ----------                   |
|     5                  1.001   sec                 |
|     9                  2.642   sec                 |
|    18                  6.084   sec                 |
|    50                 37.181   sec                 |
+----------------------------------------------------+
                                         numperf table
```

These numbers will (as you can imagine) change for different machines. As a rule of thumb, the faster the machine for INTEGER operations, the smaller will be the time for big values of NNN. I again stress the fact that the FLOATING POINT capabilities of your machine are totally irrelevant for this computation: the numbers are strings, and the floating point engine of your computer is not used by the NetRexx interpreter.

A useful program: eval.

We now look at a program that will allow you to play a little with numbers. It is called **eval**. The basic idea is to have a small calculator that you can use to perform Arithmetic calculations from your command line.

```
+------------------------------------------------------------------------+
| -- eval
| --
| parse arg expr
| r = xstring.interpret(expr,24)
| say r
| exit
+------------------------------------------------------------------------+
                                                                  eval.nrx
```

Resources... [Download the source for the eval.nrx example]

You invoke it simply by typing:

java eval *expression*

Again, in order to give you the 'feeling', here is a dump of a sample session where I use **eval**.

```
.....................................................................
rsl3pm1 (282) java eval 2+89
91
rsl3pm1 (283) java eval '50**3 +760 -98'
125662 = 125,662 = 1.25662E+5
.....................................................................
                                                      Example of eval
```

Note for UNIX users: expressions such as:

```
1*2
```

are (unfortunately) interpreted by the shell. In fact, the shell will try to find, in your current directory, all the files that have filenames starting with 1 and ending with 2. As there normally are none, you will get a "No Match", and the answer will be "java: No Match", definitely NOT what you would have expected. To avoid this strange behaviour put the expression between quotes, as here:

```
'1*2'
```

or call the program without any argument. The program will then prompt you for an expression, and (now that there is no shell intervention) you can freely put in any character.

Other Mathematical functions with arbitrary precision.

```
*
* WARNING:
*   The so called SLAC arbitrary precision function package
*   will be implemented in xmath v2.000.
*
```

The other mathematical high-level functions (like **sin() cos()** , etc.) are available with the usage of an external package.

In the following table we summarise all the available functions. As you notice ALL the functions have an "_" character after the function name.

Note also that ALL those functions are arbitrary precision functions and are totally platform independent (i.e. you'll get the same result for the 400th decimal digit of sin(2) on an HP, SGI, PC, etc.).

```
   e()                     - returns the value of natural base e
   pi()                    - return the value of PI

   XtoY( x , y )           - x to the yth power
   ln( x )                 - log of x
   log10( x )
   logbase( x , y )        -
   sqrt( x )               - square root
   exp( x )
   fact( n )               - factorial of N

   sin( x , pr , mode )  - sine of x
   cos( x , pr , mode )  - cosine of x
   tan( x , pr , mode )  - tangent of ox
   sec( x , pr , mode )  - secant of x
   csc( x , pr , mode )  - cosecant of x
```

```
cot( x , pr , mode )  - cotangent of x

asin( x , pr , mode ) - arcsine of x
acos( x , pr , mode ) - arccosine of x
atan( x , pr , mode ) - arctangent of ox

sinh( x , pr )        - hyperbolic sine of x
cosh( x , pr )        - hyperbolic cosine of x
tanh( x , pr )        - hyperbolic tangent of ox

asinh( x , pr )       - hyperbolic arcsine of x
acosh( x , pr )       - hyperbolic arccosine of x
atanh( x , pr )       - hyperbolic arctangent of ox
```

NOTE: As previously stated those functions are arbitrary precision, and are NOT machine H/W dependent.

Real Examples

In order to provide some examples of the mathematical NetRexx functions, I think it better to present some 'real' algorithms that may prove to be useful even if you do not use NetRexx. These programs, although they present language features that it would be better to explore in the next chapter, are taken 'as-is' from the 'Collected Algorithms from the ACM' book. The only difference you might notice is that we have taken out all the 'GOTOs', replacing them with a more structured approach (after all, those algorithms were invented in 1962, well before even REXX was invented). What I would like to stress is the fact that NetRexx is very good for algorithm description. What might interest you are, de facto, only the functions. The rest of the program has been presented simply as an example of how to call the functions themselves.

Greatest Common Divisor (gcd).

The following code is a small example of a call to a routine that computes the gcd of two integer numbers. The format of the call is:

```
n = xmath.gcd(n1,n2)
```

```
+----------------------------------------------------------------------+
| parse arg n1 n2 .
| say xmath.gcd(n1,n2)
| exit 0
+----------------------------------------------------------------------+
                                                                 gcd.nrx
```

Resources... [Download the source for the gcd.nrx example](http://www.netrexx.org/Tutorial/nr_7.html)

```
+----------------------------------------------------------------------+
| -- method......: gcd
| -- purpose.....: find the greatest common divisor
| --
|   method gcd(a=int,b=int) public static
```

```
    if a = 0 then return b
    if b = 0 then return a
    r2 = a
    r1 = b
    loop forever
      rr = r2/r1
      g = rr.trunc()
      r = r2 - r1*g
      if r = 0 then return r1
      r2 = r1
      r1 = r
    end

+--------------------------------------------------------------------+
                                                xmath.nrx(Method:gcd)
```

Resources... [Download the complete source for the xmath.nrx library]

The **gcd()** function is a NetRexx function that (unlike the BUILT-IN functions) such as max(), min(), etc. are USER-WRITTEN.

Simultaneous Linear Equations Solution

The following piece of code shows how to call a routine (called **gauss**) that performs the solution of a system of linear equations with the Gauss Method.

```
+--------------------------------------------------------------------+
 -- gauss
 --
 n = 3
 a = rexx[n+1,n+1]
 y = rexx[n+1]
 c = rexx[n+1]

 a[1,1] = 13; a[1,2] = -8;  a[1,3] = -3; y[1] = 20
 a[2,1] = -8; a[2,2] = 10;  a[2,3] = -1; y[2] = -5
 a[3,1] = -3; a[3,2] = -1;  a[3,3] = 11; y[3] = 0

 rc = xmath.gauss(n,a,y,c)
 say 'RC:' rc'.'

 say 'Solution:'
 loop i = 1 to n
   say c[i].format(NULL,3)
```

```
  end
  exit
+----------------------------------------------------------------------+
                                                              gauss.nrx
```

Resources... [Download the source for the gauss.nrx example]

Here is the code itself. Of course, you can grab it and put it inside your program(s).

```
+----------------------------------------------------------------------+
| -- method......: gauss                                               |
| -- purpose.....:                                                     |
  --
    method gauss(n=int,a=Rexx[,],y=Rexx[],c=rexx[]) public static;    |
      b = rexx[n+1,n+1]                                                |
      w = rexx[n+1]
      error = 0
      loop i = 1 to n
        loop j = 1 to n
          b[i,j] = a[i,j]                                              |
        end
        w[i] = y[i]
      end
      loop i = 1 to n-1
        big = b[i,i].abs()                                             |
        l = i
        i1 = i+1
        loop j = i1 to n
          ab = b[j,i].abs()                                            |
          if ab > big then
            do
              big = ab
              l = j
            end
        end
        if big = 0
        then error = 1
        else
          do
            if l<>i then
              do
                loop j=1 to n
                  hold = b[l,j]
```

```
              b[l,j] = b[i,j]
              b[i,j] = hold
            end
            hold = w[l]
            w[l] = w[i]
            w[i] = hold
          end
        loop j = i1 to n
          t = b[j,i]/b[i,i]
          loop k = i1 to n
            b[j,k] = b[j,k] - t*b[i,k]
          end
          w[j] = w[j] - t*w[i]
        end
      end
    end
    if b[n,n] = 0 then error = 1
    else
      do
        c[n] = w[n]/b[n,n]
        i = n - 1
        loop until i = 0
          sum = 0
          loop j = i+1 to n
            sum = sum + b[i,j] * c[j]
          end
          c[i] = (w[i] - sum) / b[i,i]
          i = i-1
        end
      end
    return error

+-----------------------------------------------------------------------+
                                                 xmath.nrx(Method:gauss)
```

Resources... [Download the complete source for the xmath.nrx library]

Operations on HEX Numbers

In this section we will look at how to perform favourite operations on HEX quantities. A HEX number is treated by NetRexx as a string. This string is composed of numbers (0-9) and letters (A-F). Although I am sure you know what a HEX number looks like, here are some simple assignments:

```
hex1 = 'FEA078'
hex2 = 'CAFE'
hex3 = '1AB052'
```

As you will have noticed, I have defined these quantities as PURE strings. This makes the conversion work that we will need to do very much easier. But now what happens if you try to sum hex1 to hex2? As NetRexx understands ONLY decimal arithmetic, the operation is going to fail. The only way out is to build a small function that performs the HEX operation. This function will perform all the conversion work for us, both in the hex to decimal part and in the decimal to hex re conversion. The routine I propose is **hexop()** and you call it up using the following syntax:

`hex = ` **`hexop(`*`hex1 operation hex2`*`)`**

NOTE: the 'operation' must be put into quotes. Why? Because we want to avoid REXX interpreting it as an ARITHMETIC addition (remember that hex1 and hex2 are NOT hexadecimal quantities). This is the function itself and, as you can see, it is very short:

```
-- method......: hexop
-- purpose.....: execute an HEX operation
--
  method hexop(in=Rexx) public static
    parse in n1 op n2
    n1 = n1.x2d()
    n2 = n2.x2d()
    select
      when op = '+' then n3 = n1 + n2
      when op = '-' then n3 = n1 - n2
      when op = '/' then n3 = n1 / n2
      when op = '*' then n3 = n1 * n2
      otherwise
        do
          say 'Invalid operation.'
          exit 1
        end
    end
    n3 = n3.d2x()
    return n3
```

xmath.nrx(Method:hexop)

Resources... [Download the complete source for the xmath.nrx library]

As you will note from the code (apart from the parse and the interpret instruction, which we will cover later), we do a double translationÊÑ first from HEX to DECIMAL for the two terms (x2d), and then, once we have the result, back to HEX (d2x). I do not check whether the data (and the operation) are correct or not: this is left to the calling code (or to you, if you want to enhance it). Some examples:

```
say xmath.hexop('FFFF + 1A')      -> '10019'
say xmath.hexop('FFFE / 2' )      -> '7FFF'
```

Operations on Binary Numbers

Binary numbers are composed only of '0' or '1'. Again, these numbers will be NetRexx strings. At the risk of appearing very pedantic, here are some examples of binary quantities:

```
bin1 = '10010010'
bin2 = '100001111000'
```

The very same considerations for HEX quantities are to be found in relation to binary numbers. Since we cannot directly perform arithmetic on them, we are forced to use a function expressly made for the purpose. This function is similar to the **hexop()** we just saw (in fact, in accordance with my fancy, I have expressed this in its name, calling it: **binop()**). The only additional complication lies in the fact that you can convert to and from binaries starting only from HEX quantities. The syntax for the function is:

```
bin = binop(bin1 operation bin2)
```

The code is a small variation on **hexop**:

```
+------------------------------------------------------------------------+
| -- method......: binop
| -- purpose.....: execute a BIN operation
| --
|
|   method binop(in=Rexx) public static
|     parse in n1 op n2
|
|     n1 = n1.b2x.x2d()
|     n2 = n2.b2x.x2d()
|     select
|       when op = '+' then n3 = n1 + n2
|       when op = '-' then n3 = n1 - n2
|       when op = '/' then n3 = n1 / n2
|       when op = '*' then n3 = n1 * n2
|       otherwise
|         do
|           say 'Invalid operation.'
|           exit 1
|         end
|     end
|
```

```
|      n3 = n3.d2x.x2b()
       return n3
+-------------------------------------------------------------------+
                                             xmath.nrx(Method:binop)
```

Resources... [Download the complete source for the xmath.nrx library]

Again, no check is made to ascertain if the quantities are truly binary and the operation a valid one. Some examples:

```
  say xmath.bin_op('1010 + 10')     -> '1100'
  say xmath.bin_op('1110 / 10')     -> '0111'
```

Remark on HEX and BINARY operations

A conclusive remark: as you will have have noticed, in this last case (as in the one before that, for HEX quantities) the BINARY operations are CPU-intensive in NetRexx. To perform a single addition we do six conversions and two operations (without counting the function above). I have presented the two subroutines in order to show that 'it can be done', and in a rather easy way. However, as a rule you should remember that it is always a good idea to perform ALL the arithmetic operations in your programs as decimal operations, and perform conversions at the beginning (and end) of the program itself.

Tricks with numbers.

Put dots in long numbers.

It is usually a very difficult thing to read big numbers, if they're written as:

```
100345902
```

and it would be nice to display them in the form

```
100,345,902
```

The following **xmath** function will do the job.

```
+-------------------------------------------------------------------+
| -- method......: dotify
| -- purpose.....: put dots into a numeric string
| --
|   method dotify(n=Rexx) public static
|     if n.datatype('N') = 0 then return n
|     parse n a '.' b
|     if b <> '' then b = '.'b
|     c = ''
```

```
    loop for ((a.length() - 1) % 3)
      c3 = a.right(3)
      c = ','||c3||c
      a = a.left(a.length() - 3)
    end
    return a||c||b
+--------------------------------------------------------------------+
                                        xmath.nrx(Method:dotify)
```

Resources... [Download the complete source for the xmath.nrx library]

Convert numbers in Computer Units.

Another usual conversion is to take a number and express it in Computer Units (K (kilo), M (mega), G (giga), etc.)

```
n                     cu
----------          -----
       452     ->    452
      1025     ->     1K
   1000000     ->   976K    ( why ??? )
```

The following function will do this.

```
+--------------------------------------------------------------------+
| -- method......: n2cu
| -- purpose.....: convert n to Computer Units
| --
|   method n2cu(n=Rexx) public static
|     numeric digits 32         -- set high precision
|     list = 'K M G T P'        -- Kilo Mega Giga Tera Peta
|     base = 1
|     max  = 1024
|     unit = ''
|     loop forever
|       if n < max then
|         do
|           out = (n%base)||unit
|           leave
|         end
|       parse list unit list    -- get next unit, pls
|       base = max
|       max = max*1024
|     end
|     numeric digits 9
|     return out
|
```

```
+--------------------------------------------------------------------------+
                                                    xmath.nrx(Method:n2cu)
```

Resources... [Download the complete source for the xmath.nrx library]

Call example:

```
say 'File size is' xmath.n2cu(size)'.'
```

Convert seconds to hours.

That's my last favourite conversion routine. I use it to convert seconds to a more readable human format.

```
s                     h
------     -----------
  7272  ->     2:01:12
100000  -> 1d-03:46:40
```

```
+--------------------------------------------------------------------------+
| -- method......: s2h                                                     |
| -- purpose.....: convert seconds to hours (or days)                      |
| --
|
|   method s2h(s=Rexx) public static                                       |
|     h = s%3600
|
|     s = s//3600       -- modulo                                          |
|     m = s%60
|
|     s = s//60         -- modulo
|
|     if h > 24 then    -- express h in DAYSd-HH
|
|       do              -- if necessary
|
|         d = h%24
|
|         h = h//24
|
|         h = h.right(2,'0')                                               |
|         h = d'd-'h
|
|       end
|
|     m = m.right(2,'0')                                                   |
|     s = s.right(2,'0')                                                   |
|     out = h':'m':'s
|
|     return out
|
|
+--------------------------------------------------------------------------+
                                                     xmath.nrx(Method:s2h)
```

Resources... [Download the complete source for the xmath.nrx library]

Call example:

```
say 'Time elapsed' xmath.s2h(sec)'.'
```

Chapter FAQ

QUESTION: *How do I round-up a number?* As we saw, the '/' divide operator does a 'crude and simple' truncation on the result. If you need a real round up, then you should use the **format(NULL,0)** instruction, like in the following example:

```
rounded = n.format(NULL,0)
```

You can try out the following code to test yourself.

```
+----------------------------------------------------------------------+
| -- Round up example
| --
| parse arg n1 n2
| n3 = n1/n2
| say 'Result:' n3
| say 'Round :' n3.format(NULL,0)
| exit 0
+----------------------------------------------------------------------+
                                                             roundup.nrx
```

Resources... [Download the source for the roundup.nrx example]

Summary

We resume what we've seen so far in this chapter.

```
_ basic operations                 | + - * /
                                   | - ex.: a+b
                                   |
_ setting precision                | Numeric Digits NN
                                   | - ex.: Numeric Digits 20
                                   |
_ query precision                  | digits
                                   | - ex.: nn = digits
                                   |
```

***** This section is:**

***** and will be available in next releases**

File: nr_7.html.

Last update was done on 18 May 1998 21:47:38(GMT +2).

The NetRexx Tutorial
- Operations on Strings

Operations on Strings

Introduction

*As we already said, in NetRexx there is only ONE native data type: the **string**. We already saw how to define a string; now we will concentrate our attention on how to operate on the strings, starting with the simplest operations (such as concatenating two strings together) and ending with one of the most powerful features of NetRexx, the string parsing. This chapter unfortunately contains long reference sections. I hope you will not get too tired going through them.*

The string.

I remind you that we defined a **string** as "a sequence of characters" of arbitrary length and content. Strings are defined like this:

```
string = 'This is a string'
string_new = 'and this is another one'
```

You can use ' or " quotation marks to delimit a string when you define it.

String Concatenation

The first operation you might want to perform on a string (better on two or more strings) is to concatenate them, i.e. form a single string with a set of strings. NetRexx provides you with three ways of performing this:

`(blank)`	*Concatenate terms with one blank in between;*
`||`	*Concatenate without an intervening blank;*
`(abuttal)`	*Concatenate without an intervening blank;*

Concatenation without a blank might be forced by using the **||** operator. The same result can be obtained if a literal string and a symbol are abutted. This is the **abuttal** operator. Suppose you have a variable **p1** that contains the string **'my'** and a variable **p2** that contains the string **'simple test'**. Look at the concatenation:

```
simple
  say p1 p2     -> 'my simple test'

no blanks
  say p1||p2     -> 'mysimple test'

abuttal
  say 'my'p2     -> 'mysimple test'
```

The following additional examples might better clarify how concatenation works:

```
/* setting */                        /* s values */
s1 = 'Tyranno'
s2 = '-'
s3 = 'Saurus'

s = s1 s3                            s = 'Tyranno Saurus'

-- notice I put MANY spaces between s1 and s3: they
-- have no effect
s = s1       s3                      s = 'Tyranno Saurus'

s = s1||s3                           s = 'TyrannoSaurus'

s = s1||' '||s3                      s = 'Tyranno Saurus'

-- Here spaces count!
s = s1||'    '||s3                   s = 'Tyranno    Saurus'

s = s1 s2 s3                         s = 'Tyranno - Saurus'

s = s1||s2||s3                       s = 'Tyranno-Saurus'

s = s1      s2       s3              s = 'Tyranno - Saurus'

s = s1'-'s3                          s = 'Tyranno-Saurus'
```

Comparative operators.

The very same comparative operations that can be done with numbers can, of course, be done with strings. The comparative operators return:

```
1 if the result of the comparison is true

0 otherwise
```

NetRexx has two sets of operators: the **normal** comparison and the **strict** comparison. The **strict** comparison is really what its name suggests: two strings must be **strictly** identical in order to pass the comparison.

```
NORMAL comparative operators:

     =                  True if terms are equal;
     \= , ^=            Not equal;
     >                  Greater than;
     <                  Less than;

     >< , <>            Greater than or less than
                          (same as NOT EQUAL)
     >= , ^< , \<       Greater than or equal to,
                          not less than;
     <= , ^> , \>     Less than or equal to,
```

```
                              not greater than;
```

STRICT comparative operators:

```
     ==                 True if the terms are strictly equal
                           (identical)
     \== , ^==          True if terms are strictly not
                           equal
     >>                 strictly greater than;
     <<                 strictly less than
     >>= , ^<< , \<<    strictly greater than or equal to,
                           strictly  not less than;
     <<= , ^<< , \>>    strictly less than or equal to,
                           strictly not greater than;
```

BOOLEAN operators:

```
     &                  AND;

     |                  Inclusive OR;

     &&                 Exclusive OR;

     ^  , \             LOGICAL NOT
```

You will probably never need some of these operators, although it is good to know that they exist in order to avoid 'reinventing the wheel' when faced with a particular problem. The most important operators are definitely **= , ^= , < , >**; you will be using them for 99% of your comparisons.

A small program for checking comparisons.

We give a small example that shows the difference between the **strict** and the **normal** operators: the program we run is as follows:

```
+----------------------------------------------------------------------+
| -- strict test
| --
| str    = 'test'
| str[1] = 'test'
| str[2] = ' test'
| str[3] = 'test '
| say 'Comparing "'str'".'
| loop i = 1 to 3
|   normal = (str = str[i])
|   strict = (str == str[i])
|   say normal strict
| end
| exit 0
+----------------------------------------------------------------------+
                                                            strstrict.nrx
```

Resources... [Download the source for the strstrict.nrx example](http://www.netrexx.org/Tutorial/nr_8.html)

and the result is:

```
..........................................................................
rsl3pm1 (39) java strstrict
Comparing string "test".
with "test"      is normal: 1 ; strict: 1.
with " test"     is normal: 1 ; strict: 0.
with "test   "   is normal: 1 ; strict: 0.
rsl3pm1 (40)
..........................................................................
                                                                 strc1.out
```

Miscellaneous functions on strings.

Although this book is not a true reference, I would like to present some of the many built-in functions available in NetRexx. For a complete list, consult the NetRexx Reference. The purpose of including this list here is so that I can be sure that you at least know that some instructions exist. In fact, I have to admit that once I wrote myself a function in order to find out the last occurrence of a character in a string. A colleague later showed me that this function already existed (it is called **lastpos()**).

```
-----------------------------------------------------------
Standard NetRexx functions

information.abbrev(info,length)
        Check if 'info' is a valid abbreviation for the
        string 'information';

string.center(length,pad)
        Centers a string;

string1.compare(string2,pad)
        Compares 2 strings Ñ 0 is returned if the strings
        are identical, and if they are not, it returns the
        position of the first character not the
        same;

string.copies(n)
        Makes 'n' copies of the given string 'string';

string.delstr(n,length)
        Deletes the sub-string of 'string' that begins at the
        n-th character, for 'length' characters;

string.delword(n,length)
        Same as above, but now the integers 'n' and 'length'
        indicate words instead of characters, i.e. space
        delimited sub-strings;

new.insert(target,n,length,pad)
        Inserts a string ('new') into another ('target');

haystack.lastpos(needle,start)
        Returns the position of the last occurrence of the
        string 'needle' into another, 'haystack'; if the
        string is NOT found, 0 is returned; see also pos();

string.left(length[,pad])
        Returns the string 'length' characters with the
        left-most characters of 'string';

string.length()
        Returns the 'string' length;

string.lower([n[,length]])
        Returns a lower case copy of the string.
        Lowering will be performed from character n
        for length characters. If nothing
        is specified, lower() will lowercase the
        whole string, from the 1st character.
```

```
new.overlay(target,n,length,pad)
        Overlays the string 'new' onto the string 'target',
        starting at n-th character;

haystack.pos(needle,start)
        Returns the position of one string 'needle' inside
        another one (the 'haystack');

string.reverse()
        Returns the 'string' , swapped from end to start;

string.right(length,pad)
        Returns a string of length 'length' with the 'length'
        of right-most characters of a string 'string';

start.sequence(end)
        Returns a string of all one-byte character
        representations  starting from characters 'start'
        up to character 'end';
        It replaces REXX's xrange() function;

string.space(n,pad)
        Formats the blank-delimited words in string 'string'
        with 'n' 'pad' characters;

string.strip(option,char)
        Removes Leading, Trailing, or Both (Leading and
        Trailing) spaces from string 'string';

string.substr(n,length,pad)
        Returns the substring of string that begins at the
        'n'-th character;

string.subword(n,length)
        Returns the sub-string of string 'string' that starts
        at the 'n'-th word (for 'length' words: DEFAULT is
        up to the end of string);

string.translate(tableo,tablei,pad)
        Translates the characters in string 'string'; the
        characters to be translated are in 'tablei', the
        corresponding characters (into which the characters
        will be translated), are in 'tableo';

string.verify(reference,option,start)
        Verifies that the string 'string' is composed ONLY of
        characters from 'reference';

string.word(n)
        Returns the 'n'-th blank delimited word in string
        'string';

string.wordindex(n)
        Returns the character position of the 'n'-th word
        in string 'string';

string.wordlength(n)
        As above; but returning its length;

string.wordpos(phrase,start)
        Searches string 'string' for the first occurrence
        of the sequence of blank-delimited words in 'phrase';

string.words()
        Returns the number of words in string 'string';

string.upper()
        Returns the string uppercase;

string.lower()
        Returns the string converted lowercase;

-----------------------------------------------------------
```

You might now say: Thanks a lot for this list, but what are the most important functions, i.e. the most used ones I should remember? To make myself clearer, I have taken a sample of REXX programs written by a group of people and have tried to print out some statistics on the functions you just saw. This is the result:

```
-----------------------------------------------------
substr......: 361   19%       length......:  252  13%
wordpos.....: 214   11%       upper.......:  164   8%
right.......: 152    8%       space.......:  147   7%
insert......: 110    5%       words.......:  109   5%
strip.......:  74    3%       translate...:   70   3%
abbrev......:  58    3%       lastpos.....:   48   2%
copies......:  31    1%       pos.........:   30   1%

overlay.....:  23    1%       delword.....:   14   0%
reverse.....:   5    0%       verify......:    4   0%
subword.....:   1    0%       xrange......:    1   0%
lower.......:   1    0%       center......:    0   0%
wordindex...:   0    0%       delstr......:    0   0%
compare.....:   0    0%
-----------------------------------------------------
                           most used string functions
```

As you can see, at the top of the 'TOP-10' string functions is the **substr** instruction. Functions such as **compare()** never appeared. For comparison, the **parse** instruction (see next chapter) received 567 hits, whilst the **do** got 690. I've not included those instructions in the list simply because I wanted to look at only the string functions we've seen so far.

Some 'particular' string functions.

Some of the functions you have just seen require a bit more discussion. This will be taken care of in the section that follows.

translate().

The **translate** function is used Ñ as the name suggestsÊÑ to translate the characters that form a string, following a very simple rule: if a character is in a table (usually called TABLEI), it is translated into the corresponding character present in another table (usually called TABLEO). If a given character is not in the TABLEI, then it remains unchanged. The syntax of the function is:

```
trans = str.translate(tableo,tablei)
```

Some examples will better clarify:

```
'TEST'.translate('O','E')       -> 'TOST'
'CAB'.translate('***','ABC')    -> '***'
'(INFO)'.translate('  ','()')  -> ' INFO '
```

A often-made mistake is to invert the logic for TABLEO and TABLEI: I do this myself, and put TABLEO where TABLEI should be, and vice versa. To avoid this confusion, I suggest you always try to translate before, so that you can be sure that your tables are correctly placed. What's the use of **translate()**? A typical case is when you want to get rid of characters you do not wish to process. In this way your TABLEI will contain all the unwanted characters, and

TABLEO will just be an empty string. Another possible application is an ASCII to EBCDIC converter (or EBCDIC to ASCII).

Parsing.

The **parsing** feature of NetRexx is, in my opinion, one of the most useful and powerful features of the language and probably deserves a chapter to itself. By the term **parsing** we mean the splitting up of a selected strings into assigned variables, under the control of a **template**. The syntax of the instruction is the following:

```
parse variable template
```

The *variable* is the original string you want to split-up, whilst the *template* is the set of rules to be used to do this split-up (together with the variables that will hold the result).

```
              original_string
                    |
               template
                    |
     +---------+--+-------+-----(...)---+      PARSING
     |         |          |             |
     v         v          v             v
  string1   string2    string3       stringN
```

You might consider the **template** as a 'filter', or as a 'set of rules'. NetRexx 'reads' these rules before splitting up the original string into the targeted ones, and then uses the rules to complete the task. There are several ways to parse a string. In brief, you can parse a string

- into words;
- using literal patterns;
- using periods as place-holders;
- using unsigned numbers as positional patterns;
- using signed numbers as positional patterns;
- with variable patterns;

We will now analyse all possible cases for a particular 'flavour' for the parse instruction, the **parse var**.

Parsing into words.

This is probably the most simple case: the variable is split into words defined by the variable(s) that follow the one we want to parse.

```
--------------------------------------------------------
string = 'Very Simple String'

parse string word1 word2 word3
        |
        +--->  word1 'Very'
        +--->  word2 'Simple'
        +--->  word3 'String'
```

```
str = 'This simple string, I hope, is parsed.'

parse str p1 p2 rest
       |
       +--->  p1    'This'
       +--->  p2    'simple'
       +--->  rest  'string, I hope, is parsed.'

str = 'Short string'

parse str p1 p2 rest
       |
       +--->  p1    'Short'
       +--->  p2    'string'
       +--->  rest  '' (NULL)

----------------------------------------------------------
                                       parsing into words
```

As you can see, the template is simply a set of variables, which will hold the result after the split by word has been performed. Each variable holds a word. A word is a set of characters divided by a SPACE (' ').

Parsing with literal patterns.

In this case NetRexx will scan the data string to find a sequence that matches the value of the literal. Literals are expressed as quoted strings. The literals DO NOT appear in the data that is parsed.

```
----------------------------------------------------------

str = 'Here I am.'

parse str p1 'I' p2
       |
       +--->  p1    'Here'
       +--->  p2    ' am.'

str = 'This simple string, I hope, is parsed.'

parse str p1 ',' p2 ',' p3
       |
       +--->  p1    'This simple string'
       +--->  p2    ' I hope'
       +--->  p3    ' is parsed.'

parse str p1 'simple' p2 ',' p3 'is' p4'.'
       |
       +--->  p1    'This'
       +--->  p2    'string'
       +--->  p3    ' I hope,'
       +--->  p3    ' parsed'

----------------------------------------------------------
                            parsing with  literal patterns
```

I stress the fact that the characters (or strings) that you use to build your literal patterns DO NOT appear in the final parsed result.

Parsing using periods as place holder.

The symbol **'.'** *(single dot) acts as a place holder in a template. It can be regarded as a "dummy variable", since its behaviour is exactly the same as a variable, except that the data is not stored anywhere. Use it when you 'really don't care' about some portions of a string.*

```
--------------------------------------------------------
str = 'This simple string, I hope, is parsed.'

parse str . p1 . . p2 .
       |
       +--->  p1    'simple'
       +--->  p2    'hope,'

--------------------------------------------------------
                   parsing using periods as place holder
```

As you can see, the terms **This** , **string,** , **I** , and is **is parsed.** have simply disappeared. It is a common construct to put the '.' at the end of a parsing instruction, simply to avoid the extra arguments that would pollute the last valid argument in the parsing itself. You should keep an eye on the '.' as the **/dev/null** for parsing. It can eat a word (if in the middle of a pattern) or even all the remaining part of a string, if the '.' is the last term.

parsing using unsigned numbers.

If you put unsigned numbers in a pattern, NetRexx will treat them as references to a particular character column in the input.

```
--------------------------------------------------------
str = 'This simple string, I hope, is parsed.'

parse str p1 10 p2 20 p3
       |
       +--->  p1    'This simp'
       +--->  p2    'le string,'
       +--->  p3    ' I hope, is parsed.'

str = TEST

parse str 1 p1 1 p2 1 p3
       |
       +--->  p1    'TEST'
       +--->  p2    'TEST'
       +--->  p3    'TEST'


--------------------------------------------------------
                          parsing using unsigned numbers
```

As you can see, the variable **p1** holds the characters from the original **str** string from the first to the ninth column. The variable **p2** holds the characters from the 10th column to the 19th. The variable **p3** holds the rest of the input. Note that the space is treated as is any other character. In the second example we see an interesting feature: we can restart from a given position when this is defined by an unsigned integer.

Parsing using signed numbers.

Signed numbers can be used in a template to indicate a displacement relative to the character position at which the last match occurred.

```
--------------------------------------------------------
str = 'ABCDEFGHILM'
```

```
parse str 3 p1 +4 p2
       |
       +---> p1    'DEFG'
       +---> p2    'HILM'

parse str 3 p1 +4 p2 6 p3
       |
       +---> p1    'DEFG'
       +---> p2    'HILM'
       +---> p3    'GHILM'

--------------------------------------------------------
                            parsing using signed numbers
```

Let us look at the first example: the first '3' tells the interpreter *'Position yourself at the 3rd character of "str".'* (this is "D"). Then 'p1 +4' instructs it to *'Put in "p1" the characters that follow, until you have reached the 4th character from where you were'* (this will build "DEFG"). Then we see "p2" which tells it to: *'Put all the rest in 'p2'*. So that 'p2' comes to be "HILM".

Parsing with variable patterns.

(Don't worry, this is the last case!) Using '(' ')' to delimit a variable in a template will instruct NetRexx to use ***the value*** *of that variable as a pattern.*

```
--------------------------------------------------------

delim = ','
str = 'This simple string, I hope, is parsed.'

parse str p1 (delim) p2 (delim) p3
       |
       +---> p1    'This simple string'
       +---> p2    ' I hope'
       +---> p3    ' is parsed.'

--------------------------------------------------------
                          parsing with variable patterns
```

This is probably the most complex case, since the pattern is variable.

Parsing with ALL methods intermixed.

Of course you will ask yourself: *"I've seen all those methods for parsing a string, but can I intermix them?"*. The answer is Ñ as you can imagine, since I asked this question rhetorically Ñ "Yes!". Your template can intermix all the methods we've seen so far, and it can became extremely complicated. You can write:

```
parse test 1 flag +1 info tape . '-' rest 80 comment
```

Strings & Parsing in the real life.

Implement a stack or a queue using a string.

A **stack** is an example of abstract data type (see KRUSE, 1987, pg. 150).

Usually the implementation of a **stack** is done using arrays, which require particular attention for conditions like **empty-stack full-stack**, etc.

If we make the assumption that you're dealing with numeric quantities (or with space delimited alphanumeric quantities), the implementation of a stack (or a queue) is extremely easy and elegant using a simple string.

This is how you do it:

```
(...)
stack = "                -- empty stack
(...)
stack = n stack           -- push() n into the stack
(...)
parse stack m stack       -- pop() m from the stack
(...)
entries = stack.words() -- count stack items
(...)
```

To be even more clear, let's follow the example:

```
op                        stack
--                        -----
stack = "                "
stack = 1 stack           1
stack = 2 stack           2 1
stack = 3 stack           3 2 1
parse stack m stack       2 1        m = 1
stack = 4 stack           4 2 1
parse stack n stack       2 1        n = 1
```

Parsing a list of words.

You will often find yourself with a string that contains a list of items (words). If you need to process all the items from this list, here is a simple trick for doing it. The basic idea is the following:

```
do while list <> "
  parse list item list
  (...)
  processing over 'item'
  (...)
end
```

the variable **list** is parsed with itself, and what we obtain is only its first word, keeping what remains. In fact, we are just 'eating-up' **list** word by word, in each iteration. This small piece of code illustrates the trick:

```
+----------------------------------------------------------------------+
| -- pex1.nrx
| --
| list = 'MARTIN DAVID BOB PETER JEFF'
| i = 0
| loop while list <> ''
```

```
06      parse list item list
07      i = i+1
08      say i.right(2,'0') item.left(10) list
09    end
10    exit 0
+--------------------------------------------------------------------+
                                                              pex1.nrx
```

Resources... [Download the source for the pex1.nrx example]

NOTEs:

- **line 2**: we define the list. Note that the procedure that follows will eat-up all the **list** variable, so that you need to save it if you plan on using it later;
- **line 5**: this is the real parsing phase;

Here is what you get when you run it.

```
..............................................................
01 MARTIN     DAVID BOB PETER JEFF
02 DAVID      BOB PETER JEFF
03 BOB        PETER JEFF
04 PETER      JEFF
05 JEFF
..............................................................
                                                    parseex1.out
```

Sorting.

In the NetRexx language there are no built-in sort functions.

sorting a string

The following program atom **str_sort.regproto** does a sort over a string. Even if this is not a built-in function, you call it as if it were:

```
sorted = xstring.sort(string , 'R' )
```

where **string** is our unsorted string, and **'R'** is an optional parameter to signify a reverse sorting. The code is:

```
+--------------------------------------------------------------------+
| -- method......: sort
| -- purpose.....: Sort a string
| --               A = Ascending: A B C D ...
| --               R = Reverse:   ... D C B A
| --
```

```
  method sort(stri=Rexx,mode=Rexx) public static
    if mode <> 'R' then mode = ''
    ws = stri.Words()
    incr = ws%2
    loop while incr > 0
      loop i = incr+1 for ws
         j = i-incr
         loop while j > 0
            k = j+incr
            wj = stri.Word(j)
            wk = stri.Word(k)
            if mode = 'R'
              then do ; If wj >= wk Then Leave ; end;
              else do ; If wj <  wk  Then Leave ; end;
            stri = stri.Subword(1,j-1) wk      -
                   stri.Subword(j+1,k-j-1) wj -
                   stri.Subword(k+1)
            j = j-incr
         End
      End
      incr = incr%2
    End
    stri = stri.space()
    Return stri
```

xstring.nrx(Method:sort)

Resources... Download the complete source for the xstring.nrx library

A sample program that calls such a routine is:

```
-- composers.nrx
--
composers = 'Bach Vivaldi Verdi Mozart Beethoven Monteverdi'

say 'Unsorted:' composers'.'
say 'Sorted..:' xstring.sort(composers,'A')'.'
say 'Sorted.R:' xstring.sort(composers,'R')'.'
exit 0
```

composers.nrx

Resources... Download the source for the composers.nrx example

and here is a sample output:

```
.....................................................................
rsl3pm1 (110) java composers
Unsorted: Bach Vivaldi Verdi Mozart Beethoven Monteverdi.
Sorted..: Bach Beethoven Monteverdi Mozart Verdi Vivaldi.
Sorted.R: Vivaldi Verdi Mozart Monteverdi Beethoven Bach.
rsl3pm1 (111)
.....................................................................
                                                              eso1.out
```

Other string manipulation examples

A simple "censure"

The following code is a simple implementation of a "censor" over a string. Suppose that you totally want to get rid of a string inside another string, or replace it with 'XXX' characters (like real censors do). The small method described above might help you.

```
 -- method......: censure
 -- purpose.....: get totally rid of a string sequence
 --               inside a string
 --

   method censure(s1=Rexx,s2=Rexx,ch=Rexx) public static
     -- initialization
     os = ''
     repl = ''
     if ch <> '' then
       do
         n = s2.length()
         repl = ch.copies(n)
       end


     -- do the job: this is really easy with parse ()
     loop while s1 <> ''
       parse s1 p1(s2)s1
       if s1 <> ''
         then os = os||p1||repl
         else os = os||p1
     end


     -- all done
     return os


   method censure(s1=Rexx,s2=Rexx) public static
     return censure(s1,s2,")

```

xstring.nrx(Method:censure)

Resources... [Download the complete source for the xstring.nrx library]

You should look at the way it is implemented: the string is parsed, till it is exausted, using:

```
parse string (search) string
```

where **search** is a value determined at runtime.

An animated status line.

Some programs take a long time to run, so that the person sitting in front of the terminal might ask "What ARE they doing?". So it is often nice to show the user 'where' the program is in the processing. For example, if a program has to process 300 files, and each file takes one or more seconds to process, you might want to use the routine that follows, in order to keep the person sitting at the terminal informed as to how many files the program has done, and how many there are yet to go. The following routine shows:

```
1. a 'rotating' symbol                  : (- \ | / -)
2. a number of 'done' item              : nnnn/NNNN
3. a graphic scale of 'done' items      : [****.....]
4. a numeric percent                    : nnn%
5. an additional information message    : string
```

The routine that is really of interest to you is called **info_display**. In this example, between the various displays we really do nothing (just a sleep instruction). This 'sleep' should be replaced by your computation intensive/time expensive part of the code.

```
+----------------------------------------------------------------------+
| -- method......: display                                             |
| -- purpose.....:                                                     |
| --
|
|   method display(i1=Rexx,i2=Rexx,rest=Rexx) public                   |
|     pt = dinfop//4 +1
|
|     f1 = '-\\|/'.substr(pt,1)                                        |
|     dinfop = dinfop+1                                                |
|     n1 = i1/i2*20
|
|     n2 = i1/i2*100
|
|     n1 = n1.format(3,0)                                              |
|     n2 = n2.format(3,0)                                              |
|     cu = '.'.copies(20)                                              |
|     cu = cu.overlay('*',1,n1,'*')                                    |
|     s1 = i1.right(4,'0')                                             |
|     str = f1 s1||'/'||i2.right(4,'0') '['cu'] -' rest                |
|     System.out.print(str'\x0D')                                      |
|
|
+----------------------------------------------------------------------+
                                               xstring.nrx(Method:display)
```

Resources... [Download the complete source for the xstring.nrx library]

Of course, you cannot see the motion in the figure, but you can use your imagination. You should simply try it on a real terminal, and you will get, on the very same line, something that 'moves' and shows (more or less) this:

```
.....................................................................
rsl3pm1 (80) display

\ 0001/0010 [**..................]  10%  |
(...)                                     | ALWAYS
| 0005/0010 [**********.........]  50%  | ON
/ 0006/0010 [************........]  60%  | THE
(...)                                     | SAME
- 0010/0010 [********************] 100%  | LINE

rsl3pm1 (81)
.....................................................................
                                                        display example
```

A hashing function.

I will not discuss in detail the concepts of hashing. I leave this to more specialised literature [KRUSE, LEUNG , TONDO ; 1991]. I will simply note that hashing is used to perform fast searches in databases, and hashing functions are used to index a hashing table. The basic idea of a **hashing table** is to allow a set of keys to be mapped into the same location as that of an array by means of an index function. For the moment we are not interested in implementing a full hashing table algorithm, so we will will concentrate on the hashing function itself. We need an algorithm that takes a key (a string) and builds a number. The algorithm must be quick to compute and should have an even distribution of the keys that occur across the range of indices. The following function **hash** can be used for hashing keys of alphanumeric characters into an integer of the range:

```
0 ... hash_size
```

You call the function issuing:

```
nn = hash(key)
```

```
+----------------------------------------------------------------------+
| -- method......: hash
| -- purpose.....:
| --
|
|   method hash(str=Rexx) public static
|     hash_size = 1023
|
|     t = 0                      -- zero total
|
|     l = str.length()           -- str length
|     loop while str <> ''       -- loop over all CHARS
|
|       parse str ch +1 str      --  get one
|
|       t = t+ch.c2d()           --  add to total
|
|     end                        --
|
|     out = (t*l)//hash_size     -- fold it to SIZE
|     return out
|
|
```

```
+-------------------------------------------------------------------+
                                          xstring.nrx(Method:hash)
```

Resources... [Download the complete source for the xstring.nrx library](#)

The algorithm shown is rather fast, and produces a relatively even distribution. The basic idea is in the loop that adds-up the decimal value of each character. I then multiply this value with the original length of the string, and modulo for the hash table size.

Converting from/to BASE64 (MIME).

The small programs that we analyse in this section are merely two small examples of how you can implement a BASE-64 converter. You can find more info on the Sun Implementation for a BASE64 Decoder/Undecoder methods at the URL:

http://www.java.no/javaBIN/docs/api/sun.misc.BASE64Decoder.html
http://www.java.no/javaBIN/docs/api/sun.misc.BASE64Encoder.html

Keep in mind that the MIME protocol (see RFC 1341 and 1342) is a mechanism by which you can send binary files by mail. The basic idea is the following: you take a set of bytes, you split by chunks of 6 bits each, you build 4 new bytes and you map this new quantity in base 64 (2**6 = 64). Suppose you want to translate the string "Thi" to base 64. Here is the procedure:

```
 1. Original string:
    'Thi'

 2. Translated in HEX:
    '54 68 69'

 3. translated in BINARY:
    '01010100 01101000 01101001'

 4. ditto (group by 6):
    '010101 000110 100001 101001'

 5. Add '00' in front of each 6 bits:
    '00010101 00000110 00100001 00101001'

 6. New quantities (in HEX):
    '15 06 21 29'

 7. Convert to Base 64:
    'VGhp'
```

The two following programs will convert one (a2m) from a generic string to a BASE-64 string, and the opposite for the other (m2a). Look at the listing for **a2m**. From line 16 to line 21 I put into comments the steps which I described above for the conversion (note how each step is an instruction). The whole algorithm is based on the **parse** and the **translate** function.

```
+-------------------------------------------------------------------+
16 | -- method......: a2m
17 | -- purpose.....: Convert a string from ASCII to MIME --
18 |
19 |   method a2m(str=Rexx) public static
20 |     b64 = '\x00'.sequence('\X3F')
```

```
    e64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" || -
          "abcdefghijklmnopqrstuvwxyz" || -
          "0123456789+/"

    out = ''

    loop while str <> ''

      parse str bl +3 str                          /* 1 */

      bit = c2x(bl).x2b()                          /* 2 , 3 */
      parse bit p1 +6 p2 +6 p3 +6 p4               /* 4 */

      bitn = '00'p1'00'p2'00'p3'00'p4              /* 5 */
      bln = x2c(bitn.b2x)                          /* 6 */
      base = bln.translate(e64,b64)                /* 7 */
      if base.length()<>4 then
        do

          app = '='.copies(4-base.length())
          base = base||app

        end

      out = out||base

    end

    return out

+--------------------------------------------------------------------------+
                                                 xstring.nrx(Method:a2m)
```

Resources... [Download the complete source for the xstring.nrx library]

The opposite of **a2m** is **m2a**:

```
+--------------------------------------------------------------------------+
  -- method......: m2a
  -- purpose.....: Convert a string from MIME to ASCII
  --

  method m2a(str=Rexx) public static
    b64 = '\x00'.sequence('\x3F')
    e64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" || -
          "abcdefghijklmnopqrstuvwxyz" || -
          "0123456789+/"

    out = ''

    loop while str <> ''

      parse str bl +4 str

      base = bl.translate(b64,e64)
      basex = c2x(base)

      bit = basex.x2b()
      parse bit 3 p1 9 11 p2 17 19 p3 25 27 p4 33

      bitn = p1||p2||p3||p4
      new = x2c(bitn.b2x())
      out = out||new

    end

    return out
```

```
|63
+----------------------------------------------------------------------+
                                                xstring.nrx(Method:m2a)
```

Resources... [Download the complete source for the xstring.nrx library]

Those programs could be used as building blocks for a real MIME packer/unpacker routine. Note that you will need quite a bit of work to make them really useful: what is missing is a proper handling of line splitting in the output file (in **a2m**).

Tricks with strings

TRICK: *Avoid multiple substr() calls with just one parse.* If you find yourself using more than one **substr()** function in a raw, you should probably consider rewriting your code using a more appropriate **parse** function. Suppose you have to split a time stamp in its components.

```
YYMMDDhhmmss          (timestamp)
| | | | | |
| | | | | +--------- second
| | | | +----------- minute
| | | +------------- hour
| | +--------------- day
| +----------------- month
+------------------- year
```

The first and most obvious approach is the following:

```
year  = substr(timestamp,1,2)
month = substr(timestamp,3,2)
(...)
```

And so on. The alternative using parse is:

```
parse var timestamp year +2 month +2 day +2 ,
                    hour +2 minute +2 second +2
```

The gain (both in terms of execution speed and coding) is clear: you use one instruction instead of six. Your code is also easier to modify (and to adapt to different formats of time-stamps). **TRICK:** *Use the parse with '.' to avoid the need for issuing a space() afterwards.* The title of this trick is a cryptology trick in itself. "How's that?" Simple. Suppose you need to parse lines of this format:

```
node=rsl3pm1
os=AIX
```

Depending on what the left term of the '=' sign is (we will call it the **key**), you will need to perform certain actions. What you can do is something along these lines:

```
parse var line key '=' attributes
if key == 'node' then  (...)
if key == 'os' then (...)
```

This works well until there are no extra spaces between the key and the '=' sign. But this is precisely what will happen if someone modifies the file containing these lines, as we have seen. You must be 100% sure that someone will write:

```
node = rsl3pm1
os   = AIX
```

Now the value of **key** will be: "node " and "os ", and this is not exactly what we expect. The first solution that will came to mind is the following (at least it was the first that came to my mind before learning this trick):

```
parse var line key '=' attributes
key = space(key)
if key == 'node' then  (...)
if key == 'os' then (...)
```

The trick (finally we come to it), is to use a '.' in the parse, as here:

```
parse var line key . '=' attributes
if key == 'node' then  (...)
if key == 'os' then (...)
```

This will avoid any **space()** instruction, acting as a 'space-eater'. **TRICK:** *Avoid unexpected results from a missing wordpos().* This particular trick I learned from Eric Thomas, the author of LISTSERV(tm) (probably the most popular Mailing List Server Software). I offer a concrete example: suppose you want to write a program that translates a given TCP/IP port number in its "human" meaning , i.e. a program that tells you that port 21 is FTP, port 23 is TELNET, etc. You will write two lists , one containing the port numbers, the other the 'human meaning'. These lists will then be:

```
portl    = '21  23     37'
servicel = 'ftp telnet time'
```

Note that those two lists are "ordered": 21 is the port number for FTP, 23 for TELNET, and so on , i.e. the nth element of the list **portl** corresponds to the nth element of the list **servicel**. The existence of this one-to-one correspondence is the basis of our discussion. Suppose that the port number for which we want to know the 'human meaning' is contained in the variable **port**. The obvious way to find out its meaning is, first, to identify the position in the string **portl** of the variable **port**, and second, use this number to extract (using the function **word()** the corresponding value in the list **servicel**). Each of these words translates into a sentence:

```
service = servicel.word(portl.wordpos(port))
```

This code is correct, but 'buggy'; what happens if you enter a port number that is not in **portl**? The result of **wordpos()** will be 0, and a **word** with a second argument zero will cause a buggy "ftp" answer. We could check that **port** is in **portl** before doing the **wordpos()**, but there is a simpler solution:

```
service = ('unknown' servicel).word(portl.wordpos(port) + 1)
```

The trick is simple: we add a term in front of **servicel** (the **'unknown'** term) and we add 'plus 1' to **wordpos()**. In this way we can be sure that we have covered the case when **port** is not in **portl**. The code is now correct, and can handle unexpected errors. I provide the full final code as a resume':

```
+----------------------------------------------------------------------+
| -- portn.nrx
| --
| parse arg port .
| portl    = '21  23     37'
| servicel = 'ftp telnet time'
| service = ('unknown '||servicel).word(portl.wordpos(port)+1)
| say service
| exit 0
+----------------------------------------------------------------------+
                                                              portn.nrx
```

Resources... [Download the source for the portn.nrx example]

Of course there are many more services (look to */etc/services* if you want to see them. Note also that this is NOT the way to find out the service name from the port number; rather, see the chapter on sockets in order to discover how to obtain it from the system itself.

Chapter FAQ

QUESTION: *How do I know the program's name at running time?* This is a real FAQ. Suppose that you have written (or created, to make your work more important) a program called **toto**. How does **toto** know its name? You could put the information inside a variable in **toto** but that is UGLY, and whenever you rename the program, you will need to remember to change that variable. The solution is the **parse source** instruction Ñ do

```
parse SOURCE . . myname .
```

SMALL ADDENDUM for UNIX users. If you place the program **toto** in a directory in your PATH (for example, **/usr/local/bin**) and you execute it, you will notice an interesting effect: myname is no longer **toto**, but **/usr/local/bin/toto**. This might be interesting, since you're now capable of ascertaining the directory from which your program was called, but the question then becomes how to eliminate the (probably unwanted) **/usr/local/bin**? You do it by coding:

```
myname = myname.substr(myname.lastpos('/') + 1)
```

QUESTION: *Can I put the character '00'X in a string?* Yes. The only thing you need to remember is to make the byte a HEX constant, as here:

```
string = 'this is a' '\x00' 'test'
string = '\x00\x00\x00'
```

As a rule of thumb, you can put any character you like in a string; the only thing you should remember is that you might have problems if you try a **say** of this string. **QUESTION:** *How do I display strings containing control characters?* You can use the **c2x()** instruction, in order to see the string in HEX. A more elegant way is to translate all the non-printable characters to a '.' (or to any other character you prefer). This small program shows you how to do it:

```
+------------------------------------------------------------------------+
| -- nodisp
| --
| str = 'This is a \x03\x09\xFE test.'
| tablei = '\x00'.sequence('\x1F')||'\x80'.sequence('\xFF')
| tableo = '.'.copies(tablei.length())
| say str.translate(tableo,tablei)
| exit 0
+------------------------------------------------------------------------+
                                                                nodisp.nrx
```

Resources... [Download the source for the nodisp.nrx example]

Note how I build the **tablei**: I use **sequence()** over all the unprintable characters (from '00'x to '1F'x, and from '80'x till 'FF'x). **tableo** is simply a sequence of '.' (for the same length of **tablei**). That is all I need. Note, however, that this will only work for ASCII systems: EBCDIC systems will require a different **tablei**.

Summary

We resume some of the concepts we have encountered in this chapter.

```
_ concatenate a string             | ||  or  abuttal
  (with no spaces)                 | - ex.: s1||s2
                                   | - ex.: n1'%'
                                   |
_ concatenate a string             | blank
  (with spaces)                    | - ex.: s1 s2
                                   |
```

***** This section is:**

***** and will be available in next releases**

File: nr_8.html.

Last update was done on 18 May 1998 21:47:40(GMT +2).

The NetRexx Tutorial
- Control Structures

Control Structures

Introduction.

No algorithmic language would be complete without instructions that allow the execution of statements depending on certain conditions for performing iterations and selections. NetRexx has many such instructions for allowing program flow control. Probably the most important is the ***do...end*** *construct.*

Statement Block.

A **statement block** is a sequence of statements enclosed by a **do (...) end**. A statement block looks like this:

```
do
   statement_1
   statement_2
   (...)
   statement_N
end
```

NetRexx executes these statements in sequence Ñ from the first to the last. Syntactically, a block of statements is accepted in place of any single statement.

if/then/else.

The **if/then/else** construct is used to conditionally execute an instruction or a group of instructions. The **if/then/else** construct can also be used to select between two alternatives.

```
if expression
  then instruction
  else instruction
```

The **expression** is evaluated, and MUST result in '0' or '1'. Thus, as you can imagine:

```
if expression
  then instruction   if expression results to 1
  else instruction   if expression results to 0
```

NOP

It is usually difficult to do 'nothing'. However, the **nop** instruction was created for just such a purpose: it is a dummy instruction.

NOP

It is useful as target for a **then** or **else** clause:

```
--------------------------------------------------------
   if a = 3
     then NOP
     else say 'a is NOT 3.'
--------------------------------------------------------
                                          example of NOP
```

loop for (with a repetitor)

The **loop** instruction is used (as we have already seen), to group a set of instructions, and to execute (optionally) more than once. In its easier case, the **loop for** looks suspiciously like the C-language **for** statement. Let us consider a first case:

```
loop for expression
  statement_1
  statement_2
  (...)
  statement_N
end
```

In this case, **expression** - an expression that evaluates a number - tells NetRexx 'how many times to execute the loop'. Here is an example:

```
--------------------------------------------------------
/* this statement will be executed 3 times */
loop for 3
  say 'Hello'
end
--------------------------------------------------------
                                            do N example
```

Will print on your screen:

```
   Hello
   Hello
   Hello
```

```
--------------------------------------------------------
list = 'MARTIN JOE PAULA'
/* this statement will be executed 3 times */
loop for list.words()
  parse list name list
  say 'Hi!' name
end
--------------------------------------------------------
                                     second do N example
```

Will print on your screen:

```
Hi MARTIN
Hi JOE
Hi PAULA
```

Of course, you can use a variable (which we will regard as an index) to run the iteration. This is a 'controlled repetitive loop'. A more complex case is the following:

```
loop name = expr1 to expr2
  statement_1
  statement_2
  (...)
  statement_N
end
```

Examples:

```
---------------------------------------------------------
loop i = 1 to 5
  say i
end
---------------------------------------------------------
                                             loop example
```

Will print on your screen:

```
1
2
3
4
5
```

```
---------------------------------------------------------
cols   = 2
rows   = 3
loop i = 1 to cols
  loop j = 1 to rows
    say j
  end
end
---------------------------------------------------------
                                     loop with 2 indices
```

Will print on your screen:

```
1
2
3
1
2
3
```

In the above examples, we always incremented by a positive quantity (+1). What about when your increment is NOT +1? The solution is again a **do**, but now with a **by** statement. Our **do** loop will then look like:

```
loop varname = expr1 to expr2 by expr3
```

```
  statement_1
  statement_2
  (...)
  statement_N
end
```

And here are some examples:

```
----------------------------------------------------------
loop i = 2 to -1 by -1
  say i
end
----------------------------------------------------------
                                                by example
```

Will print on your screen:

```
   2
   1
   0
  -1
```

```
----------------------------------------------------------
x1 = 2.1
x2 = 2.5
increment = .1
loop x = x1 to x2 by increment
  say x
end
----------------------------------------------------------
                                                by example
```

Will print on your screen:

```
   2.1
   2.2
   2.3
   2.4
   2.5
```

You can even add a repetition counter, which sets a limit to the number of iterations if the loop is not terminated by other conditions. Our **loop** loop will then look like the following:

```
loop varname = expr1 to expr2 by expr3 for expr4
  statement_1
  statement_2
  (...)
  statement_N
end
```

Example:

```
----------------------------------------------------------
y_start = .9
y_end = 2.7
loop y = y_start to y_end by .9 for 2
  say y
end
----------------------------------------------------------
                                               for example
```

Will print on your screen:

```
.9
1.8
```

loop/while/until.

The **while** and **until** constructs commonly found in other programming languages are also available in NetRexx, as a condition to the ubiquitous **loop** statement. Here is how to build a simple **while** loop:

```
loop while expression
  statement_1
  statement_2
  (...)
  statement_N
end
```

And here is how to build a simple **until** loop:

```
loop until expression
  statement_1
  statement_2
  (...)
  statement_N
end
```

Consider the example:

```
----------------------------------------------------------
i = 1
loop while i < 7
  say i '\-'
  i = i+1
end
----------------------------------------------------------
                                             while example
```

---> The previous code will print: **1 2 3 4 5 6**

```
----------------------------------------------------------
i = 1
loop until i > 6
  say i '\-'
  i = i+1
end
----------------------------------------------------------
                                                     until
```

---> Will print: **1 2 3 4 5 6**

do resume.

A nice NetRexx feature is that you can combine the **loop** in its **repetitive** form with the **loop** in its **conditional** form (i.e. the while/until construct we just considered). This can lead to constructs that look like:

```
---------------------------------------------------------
loop i = 1 to 10 while i < 6
  say i '\-'
end
---------------------------------------------------------
                                         combined example
```

---> This code will print: **1 2 3 4 5**. There is a nice 'side effect' to this feature, and that is the possibility of building a while/until loop without incrementing (or decrementing) the control variable yourself. Consider the case we just looked at:

```
---------------------------------------------------------
i = 1.0
loop while i < 3
  say i '\-'
  i = i+.5
end
---------------------------------------------------------
                                         do while example
```

---> This code will produce: **1.0 1.5 2.0 2.5** We need to define the start value **i = 1.0**, and define the step increment **i = i+.5**. All this can be avoided with the following construct:

```
---------------------------------------------------------
loop i = 1.0 by .5 while i < 3
  say i '\-'
end
---------------------------------------------------------
                                      do by while example
```

---> Will print: **1.0 1.5 2.0 2.5** This code is much more compact. A resume' of what we have seen so far on the **do** instruction:

```
-----------------------------------------------------------------------

loop repetitor conditional
     --------- -----------
          |          |
          |          +----------< _ WHILE expr_w
          |                       _ UNTIL expr_u
          |
          +------------< _ var = expr_i TO expr_t BY expt_b FOR expr_f
                         _ expr_r
                         _ FOREVER

   instruction_1
   instruction_2
   (...)
   instruction_N
end

-----------------------------------------------------------------------
```

select.

The **select** instruction is used to execute one of several alternative instructions. The format is:

```
select
  when expression_1 then instruction_1
  when expression_2 then instruction_2
  when expression_3 then instruction_3
  (...)
  otherwise instruction_N
end
```

What NetRexx does is evaluate the expressions after the **when**. If the result is '1', then what follows the corresponding **then** is executed (this can be anything Ñ a single instruction, a set of instructions inside a **do ... end** clause, etc.). Upon return, the control will pass directly to the **end** instruction. If none of the **when** expressions result in a '1', then the **otherwise** instruction is executed. NOTE: the **otherwise** clause is NOT mandatory, but if none of the **when** expressions results in a '1', and the **otherwise** is not present, you will get a 'SYNTAX error'. It is thus wise to ALWAYS add an **otherwise** clause at the end of a **select**, usually with a **NOP** instruction.

```
----------------------------------------------------------
(...)
/* this will print a flag corresponding to the */
/* inactivity time of a terminal:              */

/* the table is the following                  */
/* hour    0...1...2...3...4...5...6...7...8   */
/* flag    ****;;;;::::::::.................   */

/* where 'hour' is since how many hours the    */
/* terminal is inactive, and flag is the       */
/* flag we want to display                     */

/*   inactive: time (in hours)   a terminal */
/*             has been inactive            */
select
  when inactive < 1 then flag = '*'
  when inactive < 2 then flag = ';'
  when inactive < 4 then flag = ':'
  otherwise flag = '.'
end
(...)
----------------------------------------------------------
                                            select example
```

iterate.

Use the **iterate** instruction to alter the flow of control within a repetitive **do** loop (i.e. any **do** construct which is NOT a plain **do**). The syntax is:

```
do (expression)
  statement_1
  (...)
  statement_N
  (condition) iterate [name]
  statement_N+1
  (...)
  statement_M
end
```

If program flow reaches the **iterate** instruction, the control is passed back to the **do** instruction, so that the statements **statement_N+1,...statement_M** are NOT executed. Here is an example:

```
----------------------------------------------------------
loop i = 1 to 5
```

```
  say '* \-'
  if i = 3 then iterate
  say i '\-'
end
---------------------------------------------------------
                                         iterate example
```

---> This will print: *** 1 * 2 * * 4 * 5** The **iterate** instruction also supports a 'name' following it, and **name** (if present) must be the variable name of a current active loop. Consider this following code atom:

```
---------------------------------------------------------
num = 7
loop i = 1 to num
  line = ''
  loop j = 1 to num
    if i = j then
      do
        say line
        iterate i
      end
    line = line j
  end
end
---------------------------------------------------------
                                      iterate example II
```

This code will print:

```
 1
 1 2
 1 2 3
 1 2 3 4
 1 2 3 4 5
 1 2 3 4 5 6
```

leave.

Use the **leave** instruction to exit immediately from a **do** loop. The syntax is:

```
loop (expression)
  statement_1
  (...)
  statement_N
  (condition) leave [name]
  statement_N+1
  (...)
  statement_M
end
```

The flow of control is passed to the instruction that FOLLOWS the corresponding **end** in the **loop** loop. Here is an example:

```
---------------------------------------------------------
loop i = 1 to 5
  say '* \-'
  if i = 3 then leave
  say i '\-'
end
---------------------------------------------------------
```

---> The above code will produce the output: *** 1 * 2 *** You should note that **leave** is similar, in a certain sense, to the **iterate** instruction: like it, **leave** 'breaks' the normal flow of control in the **do** loop. Pictorially:

```
loop           <------+
  (...)               |
  (...)          (back to beginning)
  (...)               |
      iterate  -------+
      leave    -------+
  (...)               |
  (...)          (jump past the end)
  (...)               |
end                   |
               <------+
```

Real Examples.

As usual, we now present some 'real-life' examples.

Simulating the 'foreach' instruction.

As you may have noticed, the **foreach** instruction does not exist in NetRexx. And if you are a shell programmer, you may well also be without it. However, here is a trick for simulating it with a minimum of effort:

```
--------------------------------------------------------
loop while list ^= ''       | -> foreach item (list)
  parse list item list      |
  (...)                     |
end                         |    end
--------------------------------------------------------
                                         foreach example
```

The only thing you need to remember is that the **list** variable, at the end of the do loop, will be NULL; remember to save it if you plan to use it later.

Reading a 'stanza' file.

Configuration files are usually divided in the UNIX terminology into 'stanzas'. A 'stanza' is a uniquely identified portion of the file that contains the parameters for a specified entity. VM programmers may identify a 'stanza' as a single entry in a NAMES file: an identifier marks the start of a stanza, and a set of parameters follows, until a new stanza (or an End_of_File) is reached. Let us look at a 'stanza' example:

```
+-------------------------------------------------------------------+
| # comment line                                                    |
| node: rsl3pm1              #  first stanza                        |
|   machine: rs6000          #  defines node                        |
|   vendor: IBM              #    rsl3pm1                           |
|   location: b32r035        #                                      |
|                                                                   |
| node: sgl3pm1              #  second one                          |
|   machine: Indigo2         #  defines node                        |
|   vendor: SGI              #     sgl3pm1                          |
```

```
|   location: b11r023       #                                 |
|                                                             |
|  node: hpl3sn05                                             |
|    machine: 730/50                                          |
|    vendor: H/P                                              |
|    location: b71r233                                        |
+-------------------------------------------------------------+
                                         Source file: test.stanza
```

You should note that:

- the character **#** is used as a comment. If a line starts with a **#**, it is ignored, and if a line contains a **#**, all what follows it is also ignored;
- blank lines are ignored.

The following program is composed of a small call to a routine that does the job of:

- reading the configuration file that contains all the stanzas;
- finding out the one we are looking for;
- setting the output variable to the required values for the selected stanza.

As you can see, the function is a good example of utilisation of the **do, leave, iterate** instructions.

```
+----------------------------------------------------------------------+
| -- readst.nrx
| --
| parse arg nodeid .
|
| --
| --
|
| -- read the file
| --
| infid = xFile('test.stanza')
| rc = infid.rd_file()
| if rc <> 0 then
|   do
|     say 'problem reading "'infid.name'".'
|     exit 1
|   end
|
| output = ''
| found  = 0
| loop i = 1 to infid.line[0]
|   if infid.line[i] = '' then iterate
|   parse infid.line[i] key rest '#' .
|   if key = '#' then iterate
|   if key = 'node:' then
```

```
24
25    do
26      if found then leave
27      if rest = nodeid then
28        do
29          found = 1
30          iterate
31        end
32    end
33  if found = 0 then iterate
34  parse infid.line[i] line '#' .
35  output = output line
36 end
37 out = output.space()
38 if out = ''
39  then say 'Not found.'
40  else say output.space()
41
42 exit 0
+--------------------------------------------------------------------+
                                                          readst.nrx
```

Resources... [Download the source for the readst.nrx example]

NOTEs:

- **line 16:** we read the configuration file containing ALL the stanzas;
- **line 21:** we ignore empty lines;
- **line 23:** we ignore comment lines as well;
- **line 24:** check if this keyword identifies a new stanza;
- **line 26:** if we have already found the stanza we wanted, there is no need to continue further;
- **line 27:** if this is the stanza we wanted, remember that we found it, and iterate;
- **line 33:** up to now we have not found the stanza, so iterate;

Run this program and here is the result you will get:

```
....................................................................
rsl3pm1 (182) java readst sgl3pm1
machine: Indigo2 vendor: SGI location: b11r023
rsl3pm1 (183) java readst rsl3pm1
machine: rs6000 vendor: IBM location: b32r035
rsl3pm1 (184)
....................................................................
                                                       readst.output
```

Expanding a list.

The following problem might appear totally 'academic'. It did to me until I encountered the following problem. A directory contained a set of files (more than 20 000), each identified by a number (as filename). To make the problem clearer, my directory contained these files:

```
  10000      10001      10002       10003
  10004      10005      10006       10007
  (...)
  33002      33003      33004       33005
```

The user needed to perform operations on a subset of the files Ñ for example:

```
    10000 10981 10982 10983 21900 21901
or: 30291 30292
or: 67234 67235 67236 67237 77889 88974 88975
```

The user had to start from N and continue until item M, or from item J for K files. There was no easy solution with UNIX standard wild-cards. And the only solution was to write the items one by one. The small program (and routine) that follows is a possible solution to the problem Ñ it expands a pattern according to a very simple syntax:

```
  first-last
  first.how_many
```

The expansion is then of the type:

```
  10020-10022   -> 10020 10021 10022
  30452.4       -> 30452 30453 30454 30455
```

The program will accept any combination of items containing '.' or '-', or simple single items. The program is really very simple:

```
+------------------------------------------------------------------+
| parse arg teststr                                                 
| say expandlist(teststr)                                           
| exit 0                                                            
+------------------------------------------------------------------+
                                                        explist.nrx
```

Resources... [Download the source for the explist.nrx example]

And of course requires this small function: (I present it separately so that you can quickly put it inside a bigger program if you like it).

```
+----------------------------------------------------------------------+
| -- method......: listexpand
| -- purpose.....:
| --
|   method listexpand(il=Rexx) public static
|     ol = ''
|     loop while il <> ''
```

```
      parse il it il
      if it.pos('.') <> 0 then
        do
          parse it f'.'n
          loop i = f to f+n-1
            if ol.pos(i) <> 0 then iterate i
            ol = ol i
          end
          iterate
        end
      if it.pos('-') <> 0 then
        do
          parse it f'-'l
          loop i = f to l
            if ol.pos(i) <> 0 then iterate i
            ol = ol i
          end
          iterate
        end
       if ol.pos(it) <> 0 then iterate
       ol = ol it
    end
    Return ol

+----------------------------------------------------------------------+
                                        xstring.nrx(Method:listexpand)
```

Resources... [Download the complete source for the xstring.nrx library]

Here is what you can use it for:

```
...................................................................
rsl3pm1 (9) explist 2000 3045.3 7002-7003
2000 3045 3046 3047 7002 7003

rsl3pm1 (11) echo `explist 20000 30890-30900`
20000 30890 30891 30892 30893 30894
30895 30896 30897 30898 30899 30900

rsl3pm1 (12)  ls -la `explist 20000 30890-30900`
(...)

rsl3pm1 (13) cat `explist 20000.7 30890-30900` > toto
(...)
...................................................................
                                                          explist.out
```

Operation on arrays.

It is sometimes usefull to convert information from an array, to a string, and viceversa.

```
+----------------------------------------------------------------------+
| -- method......: a2s
| -- purpose.....: converts a Rexx array to a string
| --
|
|  method a2s(a=Rexx) public static
|    a = a
|    out = ''
|    loop i = 1 to a[0]
|      out = out a[i]
|    end
|    return out
|
+----------------------------------------------------------------------+
                                                  xstring.nrx(Method:a2s)
```

Resources... [Download the complete source for the xstring.nrx library]

```
+----------------------------------------------------------------------+
| -- method......: s2a
| -- purpose.....: converts a string to an array
| --
|
|  method s2a(str=Rexx,a=Rexx) public static
|    a = a
|    i = 0
|    loop while str <> ''
|      parse str nn str
|      i = i+1
|      a[i] = nn
|    end
|    a[0] = i
|
+----------------------------------------------------------------------+
                                                  xstring.nrx(Method:s2a)
```

Resources... [Download the complete source for the xstring.nrx library]

The following example will show the utilization of such functions.

```
+----------------------------------------------------------------------+
| -- simple test of a2s and s2a
| --
```

```
 -- convert a string to an array --
 b = rexx(")
 xstring.s2a('52 45 66 3 4',b)
 loop i = 1 to b[0]
   say i ':' b[i]
 end

 -- convert an array to a string --
 c = rexx(")
 c[0] = 3
 c[1] = 'This is a test'
 c[2] = 'another el.'
 c[3] = 'LAST ONE.'

 s = xstring.a2s(c)
 say s

 exit 0
+----------------------------------------------------------------------+
                                                            tarray.nrx
```

Resources... [Download the source for the tarray.nrx example]

Chapter FAQ

***** This section is:**

***** and will be available in next releases**

Chapter Summary

A resume' of some of the concepts we've encountered in this chapter:

```
_ block of instructions          | do (...) end
                                 | - ex.: do
                                 |          instructions
                                 |          instructions
                                 |        end
```

```
_ 'for' loop                   |  loop for n=n1 to n2 (...) end
                               | - ex.: loop i = 1 to 6
                               |          instructions
                               |          instructions
                               |        end
                               |
_ 'while' loop                 |  loop while expr (...) end
                               | - ex.: loop while i < 6
                               |          instructions
                               |          instructions
                               |        end
                               |
```

***** This section is:**

***** and will be available in next releases**

File: nr_9.html.

Last update was done on 18 May 1998 21:47:43(GMT +2).

The NetRexx Tutorial
- Classes and Objects in NetRexx

Classes and Objects in NetRexx

*As we already said, NetRexx, like its cousin Java, is an object-oriented (OOP) language . The term **object-oriented** has become so widely used that we need to give it a more concrete meaning.*

This section assumes no knowledge of object-oriented concepts.

At the end of this section, I hope that you'll get the feeling of how OOP can be "fun".

Some basic ideas.

The Object Oriented Programming basic ideas are simple ones. Unfortunately, OOP has developed some special terminology, and many introductory works become totally incomprehensible to people encountering the subject food the first time.

OOP has four key concepts. You can remember them from the acronym "A PIE": think about the big pie that software vendors are sharing in selling us their OOP products. The components are:

```
A - Abstraction
P - Polymorphism
I - Inheritance
E - Encapsulation
```

In the following part of the chapter, we will consider, as an example, the OOP representation of a 3 dimensional vector.

A 3d vector, we will see, can be defined in a computer using three numbers (this is the ABSTRACTION). A whole series of operations can be performed on a 3d vector (like inverting it, summing with other vectors, etc), making sure that we never corrupt the values of it (this is the ENCAPSULATION part). Using the concepts we used to define the 3d vector, we can build a 4d vector, keeping some of the functions we used to encapsulate the 3d vector (and this is the INHERITANCE part). Indeed, some functions (like the sum) must be overridden by the new 4d vector functions (to take account of the 4th dimension), and that' all for the POLYMORPHISM.

Resuming it in few lines definitely looks hard, but (you'll see) there is nothing more.

A vector class

In this section we develop a simple example class, that we will call **vector3d**, that , as you can easily guess, will represent a geometric object in a three-dimensional space.

A vector, quoting Feynman, **is three numbers. In order to represent step in space, say from the origin to some particular point P whose location is (*x, y, z*), we really need three numbers, but we are going to invent a single mathematical symbol, r. (...) It is not a single number, it represents *three* numbers: *x*, *y* and *z*.** (FEYNMAN, 1963).

In those words Feynman has, de facto, extracted out the essential characteristics that we need to consider in order to represent a vector on a computer. This process is called *abstraction*.

Translating the above words in the NetRexx language, we get:

```
class vector3d public
  properties public
    xc           -- x component
    yc           -- y component
    zc           -- z component
```

The important thing to note is that we did **not** define a real vector **r**. We just defined **how** we define a vector, i.e. with 3 quantities xc, yc, and zc.

The lines above contain two new keywords: **class** and **properties**.

The **class** keyword must be followed by the name of the class that we are defining. **NOTE:** this name MUST be the filename of the file we are writing: i.e. **vector3d.nrx**.

After the **properties** keyword we define the so called **data-members**, which are, de-facto, variable names.

Methods

There is a number of things that we can do with vectors: we can compute their magnitude (or module), we can inverse them. We can also execute operations with two vectors, like adding two vectors, computing their scalar product, check if they are equal, etc.

For each of those operations we then define a **method** which is, if you like, a sort of *function* that belongs to a class and that we perform over an object (belonging to this class).

```
-- method x()
-- will return the x component of the 3d vector
method x() public
  -- the code will go here

-- method inverse()
-- will inverse a 3d vector
method inverse() public
  -- the actual code will go here

-- method mag()
-- will return the magnitude of a vector
method mag() public
  -- the actual code will go here
-- etc. etc.
```

Why we use the term **method**, and not just **function** or **procedure** ? The reason is just historical [VAN DER LINDEN, 1997] and goes back to Smalltalk-72. For you, just remember that a **method** is just a **function that belongs to a class**.

With the definition of the methods, we then have completed the class definition.

Resuming, if we want to capture the class of 3d vectors, (at least partially) in NetRexx code, we will write:

```
class vector3d public
  properties public
    xc            -- x component
    yc            -- y component
    zc            -- z component

  method inverse() public
    xc = -xc
    yc = -yc
    zc = -zc

  method mag() public
    mag = Math.sqrt(xc*xc + xy*xy + xz*xz)
    return mag
```

When you define a class, you need to specify:

```
 +------------------------------------------+
 |                                          |
 | CLASS                                    |
 |                                          |
 |          +----------------------------+  |
 |          | PROPERTIES                 |  |
 |          |  (storage definitions)     |  |
 |          |                            |  |
 |          +----------------------------+  |
 |          | METHODS                    |  |
 |          |  (operations on the        |  |
 |          |   PROPERTIES)              |  |
 |          |                            |  |
 |          +----------------------------+  |
 |                                          |
 +------------------------------------------+
```

Some "real" vectors

The Objects are **instances of a Class**. So far we have defined **how** and **what** we can do to define and use a vector, but we need a "real" one, to try out the class definitions, and use it. We need an **instance** of the class.

By defining the **vector3d** class in NetRexx, we have now created a new data type. To have a REAL vector3d you then write:

```
    v = vector3d()
```

Here you just told NetRexx: "please, treat the variable **v** as a **vector3d**: as I told you in the class definition, this variable will have 3 components associated to it, and I will be capable to perform operations like **inverse** , **mag()** etc. to it".

As you probably realize, all this procedure made you create "something" that is NOT a string. Infact, as I said, NetRexx has ONLY one NATIVE data type (the **string**) but you can create your own data types, and **vector3d** is just one example.

NOTE for Java Programmers: Note that this definition is a bit different of what you would do in Java. If you had to write the very same code in Java you would do:

```
vector3d v;
v = new vector3d();
```

In NetRexx, the dynamical definition of the object is done automatically for you (saving one line of typing).

Initialising the Vector values

Now that we have a real **vector3d** object **v**, we can use its data fields and initialise it to some values.

We do it like this:

```
-- v is a vector3d object
v = vector3d()
-- initialise the vector 2 , 3 , 1
v.xc = 2
v.yc = 3
v.zc = 1
```

Memory Model

Consider the following definitions, were we define two vectors **v1** and **v2**:

```
v1 = vector3d( 1 , 3 , 0)
v2 = vector3d( 0 , 1 , 1)
```

It is important to consider how NetRexx defines those objects (and the class methods) in your computer's memory.

```
   Objects:                     Code:
   +------+                     +-------------------+
   |xc=1  |                     | vector3d()        |
v1:|yc=3  |========>===========>| x()               |
   |xc=0  |         =           |  y()              |
   |      |         =           |  z()              |
   +------+         =           |  mag()            |
                    =           |  phi()            |
   +------+         =           |                   |
   |xc=0  |         =           |                   |
v2:|yc=1  |========>=           +-------------------+
   |xc=1  |
   |      |
   +------+
```

We can see that **an object is an instance of a class** (which is a new, user defined, type).

Each object (the vectors **v1** and **v2** in our example) has its own data.

On the contrary, only ONE copy of the code for a class is shared by all the objects (that we now know we can call **instances of the class**).

Using vector3d Methods.

So far, we just defined the vector **v**, but we have done nothing with it.

To access **vector3d** methods, we use the very same syntax we used to access the data of the object.

```
v = vector3d()
-- Initialise values
(...)
m = v.mag()      -- compute vector' mag
p = v.phi()      -- compute vector's PHI
```

In classical non-OO languages (FORTRAN, REXX, Pascal, etc.) the above call would have been written like:

```
m = mag(v)
p = phi(v)
```

while, in NetRexx, we wrote:

```
m = v.mag()
p = v.phi()
```

The difference is not just cosmetics: we are stressing the fact that the "center" of our attention is the **v** object, not the action that we are performing (the computation of the MAG or of PHI).

We see that properties and methods are considered at the same logical level (even if, as memory is concerned, treated in different ways).

So:

```
v.xc = 2       -- means:
               -- assign 2 to the xc component of v

m = v.mag()   -- means:
               -- apply method "mag()" to v, and store
               -- the result in "m"
```

Initialising a vector: the constructor.

If we look closer to the instruction we used to create a vector:

```
v = vector3d()
```

we notice the usage of the parentheses **()** after the **vector3d**. This looks really like a method call. Infact, we are calling a special method, called **constructor**, which is used to perform all the initialisations that are needed to prepare the new object.

The constructor is a "special" method, that's why it MUST have the same name of the class. So, since our class is called vector3d, to define the vector3d constructor method we'll write:

```
class vector3d
      ========      <-----------------+
                                      |
                                      |
-- constructor                        |
  method vector3d( ... ) public       |
         ========   <-----------------+
                                      |
                                      |
                                      |
                +-------------------- -----------+
                |I MUST use the same name for    |
                |the CLASS and for the CONSTRUCTOR |
                +--------------------------------+
```

Our first constructor will then look like:

```
-- method......: vector3d
-- purpose.....: constructor
--
  method vector3d(x=Rexx,y=Rexx,z=Rexx) public
    this.xc = x
    this.yc = y
    this.zc = z
```

In order to use the constructor for our vector initialization, we'll then write:

```
  v = vector3d(2,3,1)
```

which is exactly the same as writing, when we had not defined the constructor:

```
  v = vector3d()        -- ditto like
  v.xc = 2              --
  v.yc = 3              --   v = vector3d(2,3,1)
  v.zc = 1              --
```

Defining more than one constructor.

You'll find that having just one constructor method is usually not enough. Even in our simple class, it would be nice if it was possible to write something like:

```
  zero = vector3d()      -- define a vector 0 0 0

  v = vector3d(3,2,1)
  z = vector3d(v)        -- define a vector like v,
                         -- i.e. 3 2 1

  unary = vector3d(1)    -- define a vector 1 1 1
```

You can do this in NetRexx writing "additional" methods, with the same name, but with different arguments. In our example, we'll write:

```
-- overloaded constructors
--
method vector3d() public
  this(0,0,0)

method vector3d(x=Rexx) public
  this(x,x,x)

method vector3d(v1=vector3d) public
  this(v1.x,v1.y,v1.z)
```

What we just achieved is an operation of "method overloading", i.e. define a method with the same name, but different arguments.

Undefined constructor

So far we have defined 4 constructor methods, which are (just to summarise):

```
method vector3d(x=Rexx,y=Rexx,z=Rexx) public

method vector3d() public

method vector3d(x=Rexx) public

method vector3d(v1=vector3d) public
```

This tells NetRexx that there are 4 ways to define a new vector. What happens if you try to write:

```
a = vector3d(1,2)
```

Simple: NetRexx does not know how to treat this case, so you'll get a very nasty message saying:

```
4 +++ a = vector3d(1,2)
  +++     ^^^^^^^^
  +++ Error: cannot find constructor 'vector3d.vector3d(byte,byte)'
```

which means: "I do not know how to deal with this special case of vector3d followed by 2 arguments."

The main() method

The **main()** method is a special one. It is the method that will automatically be called if you invoke a class directly from the command line.

Recall the **parrot** program:

```
+------------------------------------------------------------------------+
| /* parrot.nrx
```

```
|
|  * echoes back what you type on command line
|  */
|
| parse arg s1
|
| say 'you said "'s1'".'
|
| exit 0
|
+--------------------------------------------------------------------------+
                                                                   parrot.nrx
```

Resources... [Download the source for the parrot.nrx example]

If you want to write the very same code using a class, you'll do:

```
+--------------------------------------------------------------------------+
| -- This class implements a class version of parrot.nrx
| --
|
| class parrotc public
|
|
|   method main(arguments=String[]) public static
|     parse Rexx(arguments) s1
|     say 'you said "'s1'".'
|
|     exit 0
|
+--------------------------------------------------------------------------+
                                                                  parrotc.nrx
```

Resources... [Download the source for the parrotc.nrx example]

The two programs are perfectly equivalent (although the first one is definitely less typing). Infact, what NetRexx does is to translate the 1st one into "something" that looks like the 2nd one.

The main() method is very useful if you want to test a class. You will just put the class test cases, and run it typing **java PROGNAME**.

Putting all those pieces together

This is probably the most important section we've seen so far, since we finally apply in reality what we've been doing till now.

We have a file, called **vector3d.nrx**, that contains all the properties and methods used by the **vector3d** class. We compile it, and obtain a **vector3d.class** class file.

We can now edit a file that exercises the 3d vectors. The easiest one can be something like:

```
+--------------------------------------------------------------------------+
| -- tvec3ds.nrx
```

```
 a = vector3d(1,1,1)     -- define a vector
 say 'Vector "a" components:' a.components()'.'

 a.inverse()             -- inverse it
 say 'Vector "a.inverse()" is' a.components()'.'
 exit 0

+------------------------------------------------------------------------+
                                                              tvec3ds.nrx
```

Resources... [Download the source for the tvec3ds.nrx example]

As you can see, we do very little: just define a vector3d **a**, display his components, invert it, and check that all was OK.

We compile **tvec3ds.nrx**. NetRexx will grab the **vector3d** class definition at compile time, so it will know how a **vector3d** looks like. We end up with with a **tvec3ds.class**, which we can run as usual.

Resuming:

```
-- compile
[1]> java COM.ibm.netrexx.process.NetRexxC vector3d.nrx
[2]> java COM.ibm.netrexx.process.NetRexxC tvec3ds.nrx
-- run
[3]> java tvec3ds
```

To visually resume what we did, here's a picture:

```
   +---------+             +---------+
   |         |             |         |
   |SOURCE   |             |SOURCE   |
   |         |             |         |
   +---------+             +---------+
   vector3d.nrx           tvec3ds.nrx
       =                       =
       =                       =
 (java IBM...  vector3d)   (java IBM...  tvec3ds)
       =   [1]                 =    [2]
       =                       =
   +---------+              +---------+
   |         |              |         |
   |JAVA CODE|==============|JAVA CODE|
   |         |              |         |
   +---------+              +---------+
   vector3d.class           tvec3ds.class
                                 =
                                 =
                                 =
                            (java  tvec3ds)
                                 =
                                   [3]
```

The following program illustrates all what we've implemented in the vector3d class.

```
+------------------------------------------------------------------------+
 -- tvec3d.nrx
 -- exercise the 3dim vector class
 --
 a = vector3d(1)
 b = vector3d(3,4,3)
 c = vector3d()
 d = vector3d(b)
 e = vector3d()
 f = vector3d()

 say 'Vector "a" components:' a.components()'.'
 say 'Vector "b" components:' b.components()'.'
 say 'Vector "c" components:' c.components()'.'
 say 'Vector "d" components:' d.components()'.'
 say 'Say "a.mag()" is: 'a.mag()'.'

 e.zero()
 e.add(a)
 e.add(b)
 say 'Vector "a+b" is' e.components()'.'
 e.inverse()
 say 'Vector "e.inverse()" is' e.components()'.'

 e = vector3d.add(a,b)
 say 'Vector "a+b" is' e.components()'.'

 f = vector3d.greater(a,b)
 say 'Vector "greater(a,b)" is' f.components()'.'

 -- let's play with an array of vectors
 --
 k = 200
 v = vector3d[k]
 v[1] = vector3d(1,1,1)
 v[2] = vector3d(2,2,1)
 v[3] = vector3d(0,2,0)
 e.zero()
 loop i = 1 to 3
   say 'vector "v['i']" is' v[i].components()'.'
   e.add(v[i])
 end
 say 'Vector "INTEGRAL" is' e.components()'.'

 exit 0
+------------------------------------------------------------------------+
                                                                tvec3d.nrx
```

Resources... [Download the source for the tvec3d.nrx example]

***** This section is:**

***** and will be available in next releases**

Static Properties and Methods

Subclasses and Inheritance

The **vector3d** class we defined is very good for classical physics. But, for relativistic studies, we need also to add another dimension: **t**.

This means that we need a new class, which we'll call **vectorLo** (as an abbreviation for vectorLorentz: a vector in the 4 dimension space).

Extending a Class

NetRexx allows you to use the code we already wrote for the 3 dimension vector class, defining **vectorLo** as an extension (or subclass) of **vector3d**

We do this as:

```
class vectorLo public extends vector3d
   properties public
     (...)

   method (...)
```

The **extends** keyword tells NetRexx that the newly created vectorLo class is a subclass of vector3d. As such it INHERITS the variables and methods declared as public in that class.

That's where the real point is: we do not have to define again the method x(), in order to get the x component of a Lorentz vector, we just use the method we inherited from the 3 dimensional **vector3d** class.

Some methods, of course, need to be overloaded, like in the case of:

```
-- method......: components
-- purpose.....: prints the components
--
  method components() public returns string
    return '('this.xc','this.yc','this.zc','this.tc')'
```

to take into account the new dimension.

The Lorenz's vector implementation will be:

```
+------------------------------------------------------------------------+
-- This class implements a Vector in a 4 dimentional space
--
class vectorLo public extends vector3d
  properties public
    tc

-- method......: vectorLo
-- purpose.....: constructor
--
  method vectorLo(x=Rexx,y=Rexx,z=Rexx,t=Rexx) public
    super(x,y,z)
    this.tc = t

  method vectorLo() public
    this(0,0,0,0)

  method vectorLo(x=Rexx) public
    this(x,x,x,x)

  method vectorLo(v1=vectorLo) public
    this(v1.xc,v1.yc,v1.zc,v1.tc)

-- method......: components
-- purpose.....: prints the components
--
  method components() public returns string
    return '('this.xc','this.yc','this.zc','tc')'

-- method......: main
-- purpose.....: runs the test case
--
  method main(args=String[]) public static
    args=args

    a = vectorLo(1,1,1,1)
    b = a

    say 'Vector "a" components:' a.components()'.'
    say 'Vector "b" components:' b.components()'.'
    say b.mag()

    exit 0
+------------------------------------------------------------------------+
                                                              vectorLo.nrx
```

Resources... [Download the source for the vectorLo.nrx example]

Class Hierarchy

Just to clear out the terminology we speak about superclasses and sublasses, saying:

```
+---------------+
```

```
          | vector3d       |
          +----------------+
       +-(...is a superclass of...)<--+
       |                              |
       |                              |
       |                              |
       +->(...is a subclass of...)----+
          +----------------+
          | vector10       |
          +----------------+
```

or, if you prefer:

```
                         +--------+
                         |KEYWORD |
                         +--------+
                             |
                             V
 class vectorLo public extends vector3d
          |                       |
          V                       V
      +---------+             +-----------+
      |SUBCLASS |             |SUPERCLASS |
      |INHERITS |             |           |
      +---------+             +-----------+
```

Check if an object belongs to a subclass

It is sometimes useful to check if we have a particular subclass, within a superclass, and perform this check at runtime.

Java programmers might use the **instanceof** operator; in NetRexx you just do:

object **<=** *class_name*

using the **<=** operator.

So, for example, we might have:

```
   class vector3d public
      (...)
   class vectorLo public extends vector3d
      (...)
   class vectorHEP public extends vectorLo
      (...)

   v1 = vectorLo()
   if v1 <= vectorLo
```

Another way to test for a class match, as suggested by Mike Cowlishaw in a recent thread is:

```
   if OBJECT.getclass.getname == 'CLASS' then
```

I resume the above discussion in the following code:

```
+----------------------------------------------------------------------+
| -- tvecLo1.nrx
| --
|
| a = vectorLo(1,2,1,1)   -- a Lorentz vector
| b = vector3d(1,1,1)     -- a 3d vector
|
| -- check if a and b are Lorentz vectors
| --
| if a <= vectorLo
|   then say 'a is a Lor vec'
| if b <= vectorLo
|   then say 'b is a Lor vec'
|
| -- get in another way
| --
| say 'a is a:' a.getclass.getname
| say 'b is a:' b.getclass.getname
|
| exit 0
|
+----------------------------------------------------------------------+
                                                              tvecLo1.nrx
```

Resources... [Download the source for the tvecLo1.nrx example]

First case study: A better approach to vectors.

I presented the example of the 3 dimensional and 4 dimensional vector classes mainly for "educational" purposes. We saw infact a "minor" problem which is the need to write again some methods for the 4 dimensional vector class, because we need to take into account the fact that we have an extra dimension (remember the mag() method). So, if we have to deal with 5 dimensional vectors, we'll need to rewrite AGAIN those methods.

There MUST be a better approach; the idea is to write a class which has NO notion of the space dimension, and use that to build the 3d, 4d, 5d, etc. vectors.

This class will be called **xvector.nrx**.

The xvector class: a generic vector.

The xvector class will implement a N-dimensional vector. We are then forced to use arrays to hold the numerical values.

The **mag()** method will look like:

```
+----------------------------------------------------------------------+
```

```
| -- method......: mag
| -- purpose.....: vector's elements mag
| --
|
|   method mag() public
|     sum = 0
|
|     loop i = 1 to this.dimension
|       sum = sum + this.element[i]*this.element[i]
|     end
|
|     sum = Math.sqrt(sum)
|     return sum
|
|
+----------------------------------------------------------------------+
                                              xvector.nrx(Method:mag)
```

Resources... [Download the complete source for the xvector.nrx library]

```
+----------------------------------------------------------------------+
| -- method......: add
| -- purpose.....: adds a vector to another
| --
|
|   method add(v1=xvector,v2=xvector) public static returns xvector
|     v3 = xvector('0')
|     v3.dimension = v1.dimension
|     loop i = 1 to v1.dimension
|       v3.element[i] = v1.element[i] + v2.element[i]
|     end
|
|     return v3
|
|
+----------------------------------------------------------------------+
                                              xvector.nrx(Method:add)
```

Resources... [Download the complete source for the xvector.nrx library]

A revisited 3d vector.

Look now how simple is to build our 3 dimension vector class: we just extend the xvector class and override the constructor, to allow writing:

```
v = xvector3d(1,2,3)
```

I'll call the new 3d vector class **xvector3d**, to avoid confusion with the **vector3d** one we studied in the previous sections.

```
+----------------------------------------------------------------------+
| -- This class implements a Vector in a 3 dimensional space
| -- extending the xvector class
| --
|
| class xvector3d public extends xvector
```

```
 -- method......: vectorLo
 -- purpose.....: constructor
 --
   method xvector3d(x=Rexx,y=Rexx,z=Rexx) public
     super(x','y','z)

   method xvector3d() public
     this('0','0','0')

   method xvector3d(x=Rexx) public
     this(x,x,x)

   method xvector3d(v1=xvector3d) public
     this(v1.element[1],v1.element[2],v1.element[3])

 -- method......: main
 -- purpose.....: runs the test case
 --
   method main(args=String[]) public static
     args=args

     a = xvector3d(1,1,1)
     b = a


     say 'Vector "a" components:' a.display()'.'
     say a.mag()

     say 'Vector "b" components:' b.display()'.'


     exit 0
+--------------------------------------------------------------------+
                                                          xvector3d.nrx
```

Resources... [Download the source for the xvector3d.nrx example]

Second case study: the command line class cmdline.

After having dealt with vectors, which might not be interesting for you, if you're not a physicist or an engineer, let's start with some real objects that you are dealing with everyday.

The command line

The command line is one of those objects. With **command line** I mean all what you enter after a program's name on the command line (shell or DOS prompt).

```
prompt> java my_command arguments -options
                         ------------------
                                 |
                          +--------------+
                          | COMMAND LINE |
                          +--------------+
```

A **command line** is usually divided in

- arguments
- options (introduced with a "-" sign)

Just take a UNIX book and you'll find hundreds, if not thousands of examples. I give you a really small sample:

```
command             options      arguments
----------------    -------      ---------

ls -la test tost    -l -a        test tost
df -k /usr          -k           /usr
cat test            NONE         test
tar -cvf o.tar *    -c -v        *
                    -f tar
```

The operations that we do, when analysing a command line in a in a program are (in random order):

- check that the user enters the right number of arguments;
- initialise options to a default value;
- check that the options are valid;
- check that an option requiring an argument has a valid one;

Additional requirements

Since we want to be clever, we add also some requirements:

We want that the arguments and options can be intermixed: this means that:

```
myprog -t -o test.file input_arg
myprog -to test.file input_arg
myprog input_arg -o test.file -t
```

MUST be perfectly equivalent from the user's point of view. (note that this is not always true in UNIX!).

Also, we want to be capable to query, at any time in the program, the value of an option, in order to write something like:

```
il = cmdline()
(...)
if il.option('TRACE')
  then say 'Tracing is active'
(...)
```

Option pre-setting.

In the actual implementation, we need indeed an additional information, which is "how to pass the options and their default value when we create the cmdline?".

A way is to use a string that holds, separated by a delimiter, the value of :

```
- the symbol of the option (like r, t, o, etc.);
- a parameter indicating if it's a flag or a variable;
- the NAME of the option
```

```
- the default value
```

We will call this string the **rules definition**, since we use those rules to define the options.

Example:

```
't/FLA/TRACE/0'
```

we define an option (-t) which is a flag, known in our program as 'TRACE' and defaulted to 0

```
'o/VAR/OUTFID/test.output'
```

we define an option (-o) which defines a variable, known in our program as 'OUTFID' and defaulted to test.output

```
't/FLA/TRACE/0 o/VAR/OUTFID/test.output'
```

our rules definition is now to have two options, the same as above

Cmdline class overview

The **cmdline** constructor will accept two arguments: the first one being a **rexx string** containing the line entered by the user; the second one being again a **rexx string**, containing the rules in the format we defined. This allows us to already prepare all the options and all the arguments.

```
+----------------------+
|USER'S INPUT          |
|like: file1 -t -o test|
+----------------------+
                 |
cl = cmdline( inputline , rules )
 |                          |
 |                  +------------------------------+
 |                  | PROGRAMMER'S RULES           |
 |                  | like:  't/FLA/TRACE/0' -     |
 |                  |        'o/VAR/OFID/test.out' |
 |                  +------------------------------+
 |
 +-------------------------------------+
 | This object is now aware of the     |
 | options as entered by the user      |
 | allowing something like:            |
 | if \cl.option('TRACE') then ...     |
 +-------------------------------------+
```

The class will look like:

```
class cmdline
  properties private
    options
    argument
  (...)
  method cmdline(rexx,rexx) public
  method option(rexx) public
  method verify(rexx) public
  method optiondump() public
  (...)
```

Cmdline class implementation.

I show now how some of the class methods are implemented.

By far the most complex is the **cmdline** constructor. We need infact to analyse the command line, as entered by the user (**instr**) and parse the options as defined by the programmer (**rules**).

The first step is to check the **rules**, set the valid options and set the default option values.

```
+----------------------------------------------------------------------+
 -- method......: cmdline
 -- purpose.....: constructor
 --
   method cmdline(instr=Rexx,rules=Rexx) public

     -- initial setup
     --
     olist  = ''          -- option_list
     oinfo  = ''          -- option info
     outstr = ''          -- that's the string that holds all BUT the
                          -- options; we'll return this

     -- set the defaults
     --
     loop for rules.words()
       parse rules rule rules
       parse rule opt'/'info
       olist = olist opt
       oinfo[opt] = info
       parse  info kin'/'nam'/'def
       select
         when kin = 'FLA' then
           do
             value[nam] = def
           end
         when kin = 'VAR' then
           do
             def = def.translate(' ','$')
             value[nam] = def
           end
         otherwise
           do
```

```
          say '(parse_UXO) Internal error.'
          say '(parse_UXO) kin was "'kin'".'
          say '(parse_UXO) Aborted.'
          exit 901
        end
      end
    end

    -- get the options as entered
    --
    loop while instr <> ''
      parse instr var instr
      if var.left(1,1) <> '-' then
        do
          outstr = outstr var
          Iterate
        end
      svar = var
      var = var.substr(2,1)
      if olist.wordpos(var) = 0 then
        do
          say 'Invalid option "'var'" selected.'
          say 'Valid options are "'olist.space()'".'
          say 'Program aborted.'
          exit 902
        end
      info = oinfo[var]
      parse info kin'/'nam'/'def
      select
        when kin = 'FLA' then
          do
            if def = '0'
              then def = '1'
              else def = '0'
            value[nam] = def
          end
        when kin = 'VAR' then
          do
            def = def.translate(' ','$')
            cho = ''
            loop for def.words()
              parse instr tt instr
              if tt = '' then
                do
                  say 'Invalid argument for option "'var'".'
                  say 'Should be a' def.words() 'words string.'
                  say 'Like default "'def'".'
```

```
                  say 'Program Aborted.'
                  exit 903
                end
              cho = cho tt
            end
            value[nam] = cho.space()
          end
        otherwise NOP
      end
      -- here I deal with the case when one enters
      -- -tf instead of -t -f
      --
      if svar.length() <> 2 then
        do
          ll = svar.length() - 2
          oo = svar.substr(3,ll)
          instr = '-'oo instr
        end
    end
    argumentlist = outstr.space()

+----------------------------------------------------------------------+
                                              xstring.nrx(Method:cmdline)
```

Resources... [Download the complete source for the xstring.nrx library]

```
+----------------------------------------------------------------------+
  -- method......: option
  -- purpose.....:
  --
  method option(in=Rexx) public
    out = value[in]
    return out

+----------------------------------------------------------------------+
                                               xstring.nrx(Method:option)
```

Resources... [Download the complete source for the xstring.nrx library]

Additional examples

This two additional examples should clarify what we did.

```
+----------------------------------------------------------------------+
  -- test for the cmdline class
  --
```

```
 -- we allow 2 options:
 --    -t  (TRACE)  flag default to 0
 --    -o  (OUTFID) variable defaulted to test.out
 --
 parse arg argsl

 cl = cmdline(argsl,'t/FLA/TRACE/0'          -
                    'o/VAR/OUTFID/test.out')
 say 'The arguments are:' cl.arguments()'.'
 if cl.option('TRACE')
   then say 'Tracing is ON'
   else say 'Tracing is OFF'
 say 'The output file is:' cl.option('OUTFID')

 exit 0
+------------------------------------------------------------------------+
                                                                tcl1.nrx
```

Resources... [Download the source for the tcl1.nrx example]

```
+------------------------------------------------------------------------+
 -- another test
 --
 class tcl2
   properties public

   method tcl2() public

   method main(ar=String[]) public static
     argsl = xstring.a2s(ar)

     -- test for the cmdline class
     --
     -- we allow 2 options:
     --    -r  (REPLACE)   flag default to 0
     --    -T  (TESTLEVEL) variable defaulted to 0
     --
     cl = cmdline(argsl,'r/FLA/REPLACE/0'          -
                        'T/VAR/TESTLEVEL/0')
     say 'The arguments are:' cl.arguments()'.'
     if cl.option('REPLACE')
       then say 'Replace is ON'
       else say 'Replace is OFF'
     say 'The testlevel is:' cl.option('TESTLEVEL')

     exit 0
```

```
|25
+-----------------------------------------------------------------------+
                                                                 tcl2.nrx
```

Resources... [Download the source for the tcl2.nrx example]

This chapter's tricks.

Getting the arguments from main()

As we have seen, the arguments in the **main()** method are passed as an array of **string[]**.

This is clearly different from the approach we saw in Chapter 2 about the argument passing from the command line, where **arg** was returning a simple NetRexx string.

To get the arguments in the "right" way (i.e. the way you have been used to) you need to code an extra line:

```
   method main(args=String[]) public static
     arg = Rexx(args)     -- ADD THIS LINE
     parse arg p1 p2 .    -- THIS as usual
     --
```

The line:

```
  arg = Rexx(args)
```

instruct NetRexx to "translate" the array of string **args** into a single NetRexx variable **string**.

```
   args[0]    -+---( Rexx() )--> arg
   args[1]    -+
   (...)
   args[n]    -+
```

```
*** This section is:
```


```
*** and will be available in next releases
```

Chapter Summary

```
*** This section is:
```


```
*** and will be available in next releases
```

File: nr_11.html.

Last update was done on 18 May 1998 21:47:45(GMT +2).

The NetRexx Tutorial
- More on NetRexx Classes

More on NetRexx Classes

Introduction

In this chapter we'll look at some "details" we intentionally left uncovered in the previous discussion.

Basic Concepts

Patterns and Pattern Design

Pattern Design is used to sketch a solution to some particular Object Oriented problem. It has probably already happened to you (as it did to me) to think: "I've already solved this problem (or a similar one) in the past." Then you rush to your code and try to find the solution again. If I'm allowed to make such comparison, then, "Design Patterns" stand to "Object Oriented Programming" as "Algorithms" stand to "Procedural Programming". Even further, Gamma, Helm, Johnson and Vlissides text stands to "Design Patterns" as Knuth's stands to "Algorithms".

The key issue is to make your software reusable. Using Design Patterns, you not only make it such, but you also reuse other's people efforts to find the right solution.

Usage of Abstract Classes

A Simple (?) problem

Let us consider a class hierarchy for a simple problem: we consider the "universe" of 2D rectangular objects, where we'll find Rectangles and Squares. A Venn diagram representing our "universe" might be useful:

```
+-----------------------------------------------+ U
|                                               | N
|  RECTANGLE                                    | I
|                                               | V
|                  +---------------+            | E
|                  |               |            | R
|                  |     SQUARE    |            | S
|                  |               |            | E
|                  +---------------+            |
+-----------------------------------------------+

                                     Venn diagram of the
                              "universe" class RECTANGLE
                                with a subclass (SQUARE)
```

Making an Object Model

Recalling what we saw in the previous section, we can try to implement the above diagram using NetRexx. The first thing I'd think of is to make a **Rectangle** class, and have **Square** defined as a subclass of **Rectangle**.

Let's make an Object Model for this diagram:

```
+--------------+
|              |
| RECTANGLE    |
|              |<-----( RE1 )
+--------------+
      |
      *
     * *
    *****
      |
+--------------+
|              |
| SQUARE       |
|              |<-----( SQ1 )
+--------------+
```

Note that, in our diagram:

- classes are represented by squared boxes;
- the triangular symbol connects two classes, and represents the "is-a" relationship. It points ALWAYS to the superclass.
- The "<---( )" represents an object, and it points to the class it belongs to.

So, from our picture we can say phrases like: *the class "SQUARE" "is-a-subclass-of" the class "RECTANGLE"*, or *the object "SQ1" is an instance of the class "SQUARE"*.

Implementing it in NetRexx

The actual implementation is trivial: so just look at the code.

```
+------------------------------------------------------------------------+
| -- abex1.nrx
| -- Implements Rectangles and Squares
| --
| class abex1 public
|   properties public
|
|   method main(args=String[]) public static
|     args = args
|
|     RE1 = _Rectangle(1,2)
|     say RE1.area()
```

```
   SQ1 = _Square(2)
   say SQ1.area()

   exit 0

class _Rectangle
  properties public
    length
    width
  method _Rectangle(l=Rexx,w=Rexx) public
    length = l
    width = w
  method area public
    return this.length*this.width
  method set_width(w=Rexx) public
    this.width = w
  method set_length(l=Rexx) public
    this.length = l
  method perimeter public
    return 2*(this.length+this.width)

class _Square extends _Rectangle
  method _Square(s=Rexx) public
    super(s,s)
  method area public
    return this.length*this.length
  method perimeter public
    return 4*this.length
+----------------------------------------------------------------------+
                                                              abex1.nrx
```

Resources... [Download the source for the abex1.nrx example]

Critics to the above implementation

There is a series of problems with the above implementation; I analyse them in order of increasing importance.

- To compute the perimeter of a Square, we need to do 4*width. It should be more logical to do 4*side.
- We use 2 variables to store a SQUARE's side, since width and length are always equal. This means a waste of storage.
- There is no protection for somebody writing:

  ```
  SQ1 = _Square(3)
  SQ1.setlength(4)
  ```

  which is, in my opinion, REALLY a bad thing: we allow people to make squares with different sides.

Using Abstract Class

To correctly represent the Venn Diagram, we MUST use three classes. The universe class will be an "abstract" class, that we can call 2DSHAPE.

Let's revise our Object Model:

```
                  +---------------+
                  |               |
                  |  2DSHAPE      |
                  |               |
                  +---------------+
                         |
                         *
                        * *
            +---------*****-----------------+
            |                               |
+---------------+                     +---------------+
|               |                     |               |
|  SQUARE       |                     |  RECTANGLE    |
|               |<--( SQ1 )           |               |<--( RE1 )
+---------------+                     +---------------+
```

Implementation

In order to create an abstract class (i.e. a class that contains at least an abstract method), we use the keyword **abstract** (note that in C++ the keyword **virtual** is used).

That's how you'd implement in NetRexx:

```
-- abex2.nrx
-- abstract class example
--
class abex2 public

  method main(args=String[]) public static
    args = args
    R1 = _Rectangle(2,3)
    say R1.area()
    S1 = _Square(3)
    say S1.area()
    say 'You defined' _2Dshape.nobjects 'shapes.'
    exit 0

class _2Dshape abstract
  properties public static
    nobjects = 0
  method _2dShape() public
    nobjects = nobjects+1
  method area public returns Rexx abstract
  method perimeter public returns Rexx abstract

```

```
class _Rectangle extends _2Dshape
  properties private
    length
    width
  method _Rectangle(l=Rexx,w=Rexx) public
    super()
    length = l
    width = w
  method area public
    return length*width
  method perimeter public
    return 2*length*width

class _Square extends _2Dshape
  properties private
    side
  method _Square(s=Rexx) public
    super()
    side = s
  method area public
    return side*side
  method perimeter public
    return 4*side

                                                              abex2.nrx
```

Resources... [Download the source for the abex2.nrx example]

Interfaces

***** This section is:**

***** and will be available in next releases**

Dynamical Interfaces

Sample code

The interface part will look as follows:

```
-- runnable.nrx
--
class runnable interface
  method run() public
```

runnable.nrx

Resources... [Download the source for the runnable.nrx example]

```
+--------------------------------------------------------------------+
 -- dyna2.nrx
 --
 class dyna2 public

   method main(args=String[]) public static
     arg = Rexx(args)
     do
       r = runnable;
       un = Class.forName(arg);
       r = runnable un.newInstance()
       r.run()
     catch e= Exception
       say e
     end
     exit 0

 class test1 implements runnable
   method run public
     say 'Hello from class TEST1'

 class test2 implements runnable
   method run public
     say 'Hello from class TEST2'

+--------------------------------------------------------------------+
```

dyna2.nrx

Resources... [Download the source for the dyna2.nrx example]

```
+--------------------------------------------------------------------+
 -- dyna3.nrx
 --
 class dyna3 public

   method main(args=String[]) public static
     arg = Rexx(args)
     loop forever
       say 'Enter Class name (A,B,C) or quit'
       parse ask.upper() name
       if name = 'QUIT' then leave
```

```
      do
        r = runnable;
        un = Class.forName(name);
        r = runnable un.newInstance()
        r.run()
      catch e= Exception
        say e
      end
      say 'There are' A.n 'instances for A.'
      say 'There are' B.n 'instances for B.'
      say 'There are' C.n 'instances for C.'
    end
    say 'End.'
    exit 0

 -- class A
 --
 class A implements runnable
   properties static
     n = 0
   method A public
     n = n+1
   method run public
     say 'Hello from class A'

 -- class B
 --
 class B implements runnable
   properties static
     n = 0
   method B public
     n = n+1
   method run public
     say 'Hello from class B'

 -- class C
 --
 class C implements runnable
   properties static
     n = 0
   method C public
     n = n+1
   method run public
     say 'Hello from class C'
```

dyna3.nrx

Resources... [Download the source for the dyna3.nrx example]

This is what we get running **dyna3**:

```
..............................................................
Enter Class name (A,B,C) or quit
A
Hello from class A
There are 1 instances for A.
There are 0 instances for B.
There are 0 instances for C.
Enter Class name (A,B,C) or quit
A
Hello from class A
There are 2 instances for A.
There are 0 instances for B.
There are 0 instances for C.
Enter Class name (A,B,C) or quit
A
Hello from class A
There are 3 instances for A.
There are 0 instances for B.
There are 0 instances for C.
Enter Class name (A,B,C) or quit
B
Hello from class B
There are 3 instances for A.
There are 1 instances for B.
There are 0 instances for C.
Enter Class name (A,B,C) or quit
C
Hello from class C
There are 3 instances for A.
There are 1 instances for B.
There are 1 instances for C.
Enter Class name (A,B,C) or quit
B
Hello from class B
There are 3 instances for A.
There are 2 instances for B.
There are 1 instances for C.
Enter Class name (A,B,C) or quit
quit
End.
..............................................................
```

***** This section is:**

***** and will be available in next releases**

Patterns

The Singleton

The idea of **Singleton** is simple: we want to make sure that a class has ONLY one instance, and we want to provide a global point of access to it.

The structure is (GAMMA, 96, p. 127)

```
+-----------------------------+
```

```
| Singleton                     |
+-------------------------------+          +-----------------------\
|    static Instance()    *-------------->  | return uniqueInstance|
|    (...)                      |          +-----------------------+
|    SingletonOperation()       |
|    GetSingletonData()         |
|                               |
|                               |
|                               |
+-------------------------------+
|    static UniqueInstance      |
|    (...)                      |
|    singletonData              |
|                               |
|                               |
+-------------------------------+
```

NetRexx Implementation of the Singleton

The NetRexx implementation of the Singleton Pattern might look like:

```
-- Singleton.nrx
--   NetRexx Implementation of Singleton
--   see GAMMA, 1996, p.127
--
class Singleton public

  properties private static
    _instance = Singleton NULL

  method Singleton() private

  method Instance() returns Singleton public static
    if _instance = NULL then
      do
        _instance = Singleton()
        return _instance
      end
    return _instance
```

Singleton.nrx

Resources... [Download the source for the Singleton.nrx example]

Let's look at it closely. The first "uncommon" feature we find is:

```
method Singleton() private
```

i.e. the constructor is declared as **private.** Clients will not be capable to access it with a normal:

```
s = Singleton()
```

Instead, they're forced to use the **Instance()** member function, declared as static.

This means that the clients will need to write:

```
s = Singleton.Instance()
```

in order to get the unique Singleton's instance.

```
*** This section is:
```


```
*** and will be available in next releases
```

An history class.

Description of the problem

It is sometimes interesting to record the actions that an user enters when dealing with an interactive program. This is, for example, the case of the **history** command in an UNIX shell.

First approach.

When I dealt for the first time with an implementation of an history command, my solution was to define a history buffer (with his length):

```
properties public static
  cmdbuf = Rexx('')
  cmdbufl = 20
```

and 2 methods to save/dump the history:

```
+----------------------------------------------------------------------+
| -- method......: historyd
| -- purpose.....: display the history
| --
|   method historyd(cur=Rexx) public static
|     if cur < cmdbufl
|       then st = 1
|       else st = cur-cmdbufl
|     loop i = st to cur-1
|       say i.right(5) cmdbuf[i]
|     end
|
|
```

```
+--------------------------------------------------------------------------+
                                              xshell1.nrx(Method:historyd)
```

Resources... [Download the complete source for the xshell1.nrx library]

```
+--------------------------------------------------------------------------+
| -- method......: history
| -- purpose.....: history
| --
|
|  method history(a=Rexx,n=Rexx) public static
|    if a <> '' then
|
|      do
|
|        cmdbufl = a
|
|      end
|
|    else
|
|      do
|
|        historyd(n)
|      end
|
|
+--------------------------------------------------------------------------+
                                               xshell1.nrx(Method:history)
```

Resources... [Download the complete source for the xshell1.nrx library]

In the main loop, I was calling saving the entered command in the buffer

```
cmdbuf[cmdno] = todo
cmdno = cmdno+1
```

The history class

The commands are saved in the history buffer inside a circular buffer

```
+--------------------------------------------------------------------------+
| -- method......: save
| -- purpose.....:
| --
|
|  method save(entry=Rexx) public
|    k = lastrec // maxrec
|    if record[k] <> NULL then
|
|      do
|
|        if entry = record[k]
|
|          then return
|
|      end
|
|    lastrec = lastrec+1
|    k = lastrec // maxrec
|    record[k] = entry
```

```
+-----------------------------------------------------------------------+
                                              history.nrx(Method:save)
```

Resources... [Download the complete source for the history.nrx library]

```
+-----------------------------------------------------------------------+
| -- method......: dump
| -- purpose.....:
| --
|
|   method dump(n=Rexx) public
|     first = lastrec - n + 1
|
|     loop i=first to lastrec
|       k = i// maxrec
|
|       if record[k] = NULL then iterate
|       if record[k] = ''   then iterate
|       say i.right(5) record[k]
|     end
|
|
+-----------------------------------------------------------------------+
                                              history.nrx(Method:dump)
```

Resources... [Download the complete source for the history.nrx library]

```
+-----------------------------------------------------------------------+
| -- method......: retrieve
| -- purpose.....:
| --
|
|   method retrieve(n=Rexx) public returns Rexx
|     if n < lastrec - maxrec then return ''
|     if n > lastrec then return ''
|     k = n// maxrec
|
|     return record[k]
|
+-----------------------------------------------------------------------+
                                          history.nrx(Method:retrieve)
```

Resources... [Download the complete source for the history.nrx library]

```
his = history(100)

loop
  -- get user input
  his.save(USER_INPUT)

end
```

Additional sources of information

You can find additional information about patterns at:

http://st-www.cs.uiuc.edu/users/patterns/

with some tutorial information at:

http://www.enteract.com/~bradapp/docs/patterns-intro.html
http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/

Chapter Summary

***** This section is:**

***** and will be available in next releases**

File: nr_12.html.

Last update was done on 18 May 1998 21:47:47(GMT +2).

The NetRexx Tutorial
- Operations on files

Operations on files

Introduction

File I/O has always been a 'problem' in REXX. In fact, REXX was born as a platform-independent language. There were no (at least not in the first specifications of the language) direct ways to access informations stored into a file. This led to (quoting some examples) EXECIO in VM/CMS, CMS/PIPES in later VM/CMS, STREAMS in OS/2, etc.

In NetRexx we use the file access routines that Java provides. In this chapter we will discuss how to access file information using NetRexx. You will learn how to:

- *read/write a file in one single operation;*
- *use 'pseudo' RECFM F files on UNIX;*
- *use indexed file and get a random record out of 1 000 000 within a millisecond.*

We will perform this task using an OOP approach.

Thanks to Massimiliano (Max) Marsiglietti, a NetRexx porting of the functions ***STREAM, CHARIN, CHAROUT*** *etc. has been done. Those routines will be discussed in the latest section of this chapter.*

Basic Concepts

The Stream I/O model

In NetRexx (like in Java) all the I/O mechanism is built on **Streams**. The idea of Streams is similar (but not related) to the STREAMS in the Unix Kernel.

The Stream class gives you a mechanism "to get from" or "to put in" data into "somewhere". This "somewhere" can be a file, a screen or keyboard, the network, an audio device, etc. Your program will not have to care about the "details" in the implementation. It will just **read()** or **write()** to the Input or Output Stream.

```
                           ______________
                          (              )
  read()   <---(bytes)---(  Input  Stream  )
                          (______________)

                           ______________
                          (              )
```

```
write()  ----(bytes)-->( Output Stream )
                       (_____________)
```

Read/Write in Blocks

Before going into the details concerning the file I/O operations, let me clarify a point which is, in my opinion, extremely important.

I have always found the approach in which one tries to "confine" the operations on files inside subroutines more elegant, and easier to port and maintain. This is in contrast with the usual practice of intermixing file operations with the normal program flow. Let me make this clearer. The 'standard' approach is the following:

```
-- "standard" approach to file
-- Read/Write
open INPUT_FILE
open OUTPUT_FILE

do while THERE_ARE_RECORDS
  read INPUT_FILE
  process RECORD
  write OUTPUT_FILE
end

close INPUT_FILE
close OUTPUT_FILE
```

what I prefer is the following:

```
-- "alternative" and "preferred"     *** BETTER ***
-- approach to FILE I/O

-- opening/reading/closing
-- of the input file is done inside the rd_file
-- method
read_file INPUT_FILE

do for RECORDS
  process RECORD
end

-- opening/writing/closing
-- of the output file is done inside the wr_file
-- method
write_file OUTPUT_FILE
```

All the 'dirty' jobs (checking file existence, opening, closing, transferring data to and from an array, etc.) are reduced by this approach to ONLY two methods (the **read_file** and the **write_file**). There are cases where (as we shall see) there is no choice other than to take the first approach, but these are rare. I have personally used the second in 95% of programs and, again, it is easier to read, easier to port, simpler to maintain. You might ask yourself why I stress the benefits of code porting. An example is the code on VM/CMS written some years ago. In the early versions of Rexx there were NO instructions for file I/O, so someone was obliged to use the "infamous" EXECIO instruction. If all the instructions are 'confined' in two subroutines (now we call'm methods) , the changes are minimal when the code is ported. Otherwise you will need to change it in hundreds of places (if the program is big). And the more changes you make, the more bugs that can slip in. Summary: use simple methods for file I/O operations as much as you can; some of those methods you can see later in this chapter.

Checking file existence

The first thing you might want to do on a file is to check that it really exists. You can use the **xfile** built-in function 'state':

```
rc = xfile.state(file_id)
```

Alternatively, you could use the HEP/VM function fexist (file exist):

```
rc = xfile.fexist(file_id)
```

The output variable (rc in this example) has the following meaning for both functions:

```
   rc   = 0  : file does NOT exist
   rc   = 1  : file exists
```

Here a small example of the function:

```
 -- (...)
 if \xfile.state('xstring.nrx')
   then say 'File does not exist.'

 if \xfile.fexist('/usr/local/bin/tcsh')
   then say 'TCSH not present.'

 exit 0
```

fexa1.nrx

Resources... [Download the source for the fexa1.nrx example]

The implementation of those functions is trivial:

```
 -- method......: state
 -- purpose.....: check file existence
 --
   method state(fid=Rexx) public static
     in = File(fid)
     fl = in.exists()
     return fl
```

```
+--------------------------------------------------------------------+
                                            xfile.nrx(Method:state)
```

Resources... [Download the complete source for the xfile.nrx library]

Basic File operations in Java.

This entire section should be regarded as "reference" only. You will find, in fact, the basic I/O operations that you perform on a file. In my humble opinion it is nice to know these functions exist, but it is much better to use higher level subroutines that do all the work for you. Therefore, you should skip this section if you are not really interested in the detail.

Java classes for File access

- java.io.File
- java.io.FileDescriptor
- java.io.RandomAccessFile
- java.io.InputStream
- java.io.OutputStream
- java.io.PrintStream

```
*** This section is:
```


```
*** and will be available in next releases
```

Writing an extension to the Java File class

In the previous chapter we showed the advantages of the OOP. We now even more clarify those advantages building our own extensions to the java.lang.Object class **File**.

This new class (that we'll call **xfile**) will allow us to:

- use an intermediate array to buffer READ or WRITE operations
- perform appends to existing files
- simulate fixed length files (RECFM F)
- allow the building of an index for fast random access record retrieval

Reading and Writing a whole file.

It is sometimes desirable to read or write an ENTIRE file (i.e. from the first to the last line) with a single operation. This approach has the obvious advantage of giving 'somebody else' all the bother of opening, read/write and closing a file. That 'somebody' is merely the code that performs the function. The only drawback to such an operation is that, especially if the file is big, it uses a lot of system resources. Therefore, as a rule of thumb, use the ENTIRE file approach only for files < 1MB in size when you already know you are using ALL the records.

Implementation of read() and write() in xfile.

***** This section is:**

***** and will be available in next releases**

```
+---------------------------------------------------------------------+
| -- method......: read
| -- purpose.....: read a full file into an array
| --
|
|   method read() public
|     rc = 0
|
|     do
|
|       in = DataInputStream(FileInputStream(File(name)))
|     catch er=ioException
|       rc = 3
|
|       return rc
|
|     end
|
|
|     i = 0
|
|     loop while in.available <> 0
|       i = i+1
|
|       line[i] = in.readLine
|     catch er=ioException
|       say 'Problem reading file "'name'".'
|       say 'Message is "'er'".'
|       rc = 1
|
|       return rc
|
|     end
|
|     line[0] = i
|
|     lines = line[0]
|
|     return rc
|
|
|
+---------------------------------------------------------------------+
                                                  xfile.nrx(Method:read)
```

Resources... [Download the complete source for the xfile.nrx library](http://www.netrexx.org/Tutorial/nr_13.html)

```
+------------------------------------------------------------------------+
| -- method......: write
| -- purpose.....: ARRAY -> disk file operation
| --
|
|  method write() public
|    rc = 0
|
|    do
|
|      out = PrintStream(FileOutputStream(File(name)))
|    catch er=ioException
|      say 'Problem opening file "'name'".'
|      say 'Message is "'er'".'
|      rc = 3
|
|      return rc
|
|    end
|
|
|
|    loop i = 1 to line[0]
|
|      linew = line[i]
|
|      if recfm = 'F' then           -- is recfm = F ?
|
|        do                          -- Yup, insert the right amount
|          linew = linew.left(lrecl) --      of spaces (or truncate
|        end                         --      if necessary)
|
|      out.println(linew)
|    end
|
|
|
|    -- we're done. but do not forget to close
|
|    -- and flush the printstream
|    --
|
|    out.close()
|    if out.checkError() then
|      do
|
|        say 'ERROR in writing "'name'".'
|        rc = 1
|
|      end
|
|    return rc
|
|
|
+------------------------------------------------------------------------+
                                                    xfile.nrx(Method:write)
```

Resources... [Download the complete source for the xfile.nrx library](http://www.netrexx.org/Tutorial/nr_13.html)

How to use the new methods.

You use the methods in the following way:

```
infid = xfile('test.input')

rc = infid.read()
--   ----- ------
 |      |     |
 |      |     +-------------< read operation
 |      +-------------------< file object
 +--------------------------< ==0 : OK
```

```
                                  <>0 : problem

   (...)

   oufid = xfile('test.output')

   rc = oufid.write()
   --    ----- ------
    |       |    |
    |       |    +------------< read operation
    |       +-----------------< file object
    +-------------------------< ==0 : OK
                                <>0 : problem
```

Example of reading of an entire file.

If you need to read an entire file and put its contents into the ARRAY variable, you use the **.read()** method. Let's follow a complete example. Suppose your input file is **test.data**, and it looks like:

```
+------------------------------------------------------+
| data info 1                                          |
| data info 2                                          |
| (...)                                                |
| data info N                                          |
+------------------------------------------------------+
file: test.data
```

You read the ENTIRE file by calling

```
+-------------------------------------------------------------------+
| (...)                                                             |
| infid = xfile('test.file')                                        |
| (...)                                                             |
| rc = infid.read()                                                 |
| if rc <> 0 then      /* action on READ fail */                    |
| (...)                                                             |
+-------------------------------------------------------------------+
                                                  Example: read a file
```

AFTER the call, if rc was == 0, you get the values

```
  infid.line[0]   :   N
  infid.line[1]   :   'data info 1'
  infid.line[2]   :   'data info 2'
  (...)
  infid.line[N]   :   'data info N'
```

You can now process the lines with a loop, such as

```
+-------------------------------------------------------------------+
| (...)                                                             |
| loop i = 1 to infid.line[0]                                       |
|   parse infid.line[i] (...)                                       |
|   (...)                                                           |
| end                                                               |
| (...)                                                             |
+-------------------------------------------------------------------+
                                         Example: post read processing
```

Writing a whole file

Now consider the opposite situation, where we accumulate information into an ARRAY and we want to write a file with it (for example `test.output`).

```
+------------------------------------------------------------------+
| (...)                                                            |
| oufid = xfile('test.output')                                     |
| (...)                                                            |
| loop i = 1 to 30                                                 |
|   (...)                                                          |
|   oufid.addline('Output line' i)                                 |
| end                                                              |
| (...)                                                            |
| rc = oufid.write()                                               |
| if rc <> 0 then      /* action on WRITE fail */                  |
| (...)                                                            |
+------------------------------------------------------------------+
                                              Example: read a file
```

Read/Write access to a file (line by line)

Reading a file line by line

It is sometimes more desirable to read a file line by line and perform certain tasks within the reading loop. A typical case is when the input file is REALLY big ; for example, a 200MB tape or database. Another instance is when you really do not need to read all the records of the file, but only certain selected ones ; for example, all the accounting cards for a certain user. The logic is the following:

```
open(file)
do while NOT(EOF)
  record = readline(file)
  --
  -- processing
  --
end
close(file)
```

The following code is an example of this approach. You will notice that it is far more expensive in terms of instructions and complexity than the **read()** example.

```
*** This section is:
```


```
*** and will be available in next releases
```

Writing a file line-by-line

It is also interesting to consider writing a file line-by-line. This is again a case where the file being produced is big, or where you do not want to store it inside an ARRAY variable. The logic is

```
open(file)
do for all records
  --
  -- processing
  --
  writeline(file,record)
end
close(file)
```

Here is a complete example:

```
*** This section is:
```


```
*** and will be available in next releases
```

Read/Write access to a fixed-format record file

Unlike the VM/CMS and MVS systems, UNIX and Windows systems have no concept of RECORDS in files, so there is not much point in referring to LRECL and RECFM. However, using the **xfile** class you can access for read and for write a 'pseudo' fixed length file such as you are used on VM or on MVS. The advantage of these files is that you can access them on a record basis and use the record number as the index.

Suppose, for example, you have a TAPE database containing 300 000 records. To access the 283 954th one, where the records are all of the same length, you simply need to position yourself at the 283.954*RECL byte, and operate over a RECL quantity. And that is what the following functions will do. A 'pseudo' RECFM F LRECL 16 file will appear like this on your system:

```
.........    ---------------------------------------------------
record 01    t  h  i  s     i  s     a     t  e  s  t
  (hex)      74 68 69 73 20 69 73 20 61 20 74 65 73 74 20 20 0A
record 02    a  n  o  t  h  e  r     l  i  n  e
  (hex)      61 6E 6F 74 68 65 72 20 6C 69 6E 65 20 20 20 20 0A
record 03    l  a  s  t     o  n  e
  (hex)      6C 61 73 74 20 6F 6E 65 20 20 20 20 20 20 20 20 0A
.........    ---------------------------------------------------
```

This file does have three records of 16 (actually 17 with the '0A'x character) characters, so it occupies 51 bytes of disk space. Note that the '0A'x character is not mandatory. You could rewrite the routines presented herein in order to avoid it. I prefer having it so that I can look at the produced files with an editor or a browser. The format of the function to access a RECFM F file is the following:

```
fid = xfile('test.FIXED')
fid.options('recfm=F,lrecl=80')
```

```
-- write record 120
fid.recwrite(120,'Test')

-- read record 133
parse fid.recread(133) rc line
```

The methods are the following:

```
+------------------------------------------------------------------------+
| -- method......: recio
| -- purpose.....: RANDOM access file record read
| --
   method recio(oper=Rexx,recno=Rexx,out=Rexx) public
     rc = 0


     -- checks & initialization
     --
     oper = oper.upper()
     if recfm <> 'F' then
       do
         rc = 10
         return rc 'ERROR: not a RECFM=F file.'
       end
     raff = File(name)
     size = int raff.length()
     skip = (recno-1)*(lrecl+1)
     skip = int skip
     if size <= skip then
       do
         rc = 11
         return rc 'ERROR: past file end.'
       end


     -- access as a Random File
     -- and skip till the beginning of record
     do
       raf = RandomAccessFile(name,"rw")
     catch er=ioException
       say 'Problem opening file "'name'".'
       say 'Message is "'er'".'
       rc = 3
       return rc
     end
     do
       raf.skipBytes(skip)
     catch er=ioException
       rc = 4
       return rc
     end
```

```
    if oper = 'READ' then
      do
        do
          line = raf.readLine()
        catch er=ioException
          say 'Problem reading file "'name'".'
          say 'Message is "'er'".'
          rc = 3
          return rc
        end
        return rc line
      end

    -- is it a WRITE operation?
    --
    if oper = 'WRITE' then
      do
        do
          linew = out.left(lrecl)
          buf = linew'\x0A'
          raf.writebytes(buf)
        catch er=ioException
          say 'Problem reading file "'name'".'
          say 'Message is "'er'".'
          rc = 3
          return rc
        end
        return 0
      end
    return 11

+-----------------------------------------------------------------------+
                                               xfile.nrx(Method:recio)
```

Resources... [Download the complete source for the xfile.nrx library]

```
+-----------------------------------------------------------------------+
  -- method......: recwrite
  -- purpose.....: RANDOM access file record write
  --
   method recwrite(recno=Rexx,rec=Rexx) public
     out = recio('WRITE',recno,rec)
     return out

+-----------------------------------------------------------------------+
                                            xfile.nrx(Method:recwrite)
```

Resources... [Download the complete source for the xfile.nrx library]

The following program makes use of the above methods, showing all the possibilities:

```
+--------------------------------------------------------------------+
01 -- test the xfile fixed record feature
02 --
03 parse arg what
04 what = what
05
06 fname = 'test.FIXED'
07 fid = xfile(fname)
08 fid.options('recfm=F,lrecl=16')
09
10 say 'Accessing file "'fid.name'".'
11 fid.addline('this is a test')
12 fid.addline('another line')
13 fid.addline('last one')
14
15 rc = fid.write()
16 say 'RC:' rc' writing "'fid.name'".'
17
18 /* access a record
19   */
20 say fid.recread(2)
21 say fid.recwrite(2,'New line 2')
22 say fid.recread(2)
23
24 exit
+--------------------------------------------------------------------+
                                                              tfix.nrx
```

Resources... [Download the source for the tfix.nrx example]

Some explication: In line '08' we write a file, RECFM F LRECL 16 ,using the contents of the stem **list.**. The file will look like this:

```
.........    ---------------------------------------------------
record 01    t  h  i  s     i  s     a     t  e  s  t
  (hex)      74 68 69 73 20 69 73 20 61 20 74 65 73 74 20 20 0A
record 02    a  n  o  t  h  e  r     l  i  n  e
  (hex)      61 6E 6F 74 68 65 72 20 6C 69 6E 65 20 20 20 20 0A
record 03    l  a  s  t     o  n  e
  (hex)      6C 61 73 74 20 6F 6E 65 20 20 20 20 20 20 20 20 0A
.........    ---------------------------------------------------
```

In line '11', we 'zap' the contents of the record 2, so our file will look like this:

```
.........    ---------------------------------------------------
record 01    t  h  i  s     i  s     a     t  e  s  t
  (hex)      74 68 69 73 20 69 73 20 61 20 74 65 73 74 20 20 0A
record 02    N  e  w     l  i  n  e     2
  (hex)      4E 65 77 20 6C 69 6E 65 20 32 20 20 20 20 20 20 0A
```

```
record 03    l  a  s  t     o  n  e
  (hex)      6C 61 73 74 20 6F 6E 65 20 20 20 20 20 20 20 20 0A
.........    --------------------------------------------------
```

In line '14' we read the record we just zapped in an indexed way Ñ i.e. we access JUST the 2nd record of the file. If you run it, this is what you get:

```
..................................................................
rsl3pm1 (521) java tfix
0 another line
0
0 New line 2
rsl3pm1 (522)
..................................................................
                                                          rwf.out
```

You can look at the file with your preferred editor, and check that it's really like I said.

Indexed files

What we discussed about RECFM F files is also true for RECFM V files. On VM/CMS and MVS systems, you can say: "get the record NNN of this file", and you get it in a really fast way. In UNIX, this is not possible. If you want the NNNth record of a file, and the file is NOT fixed length, you MUST read all the file till line NNNth (in the assumption that a record corresponds to a line). In this chapter we will analyse a method for overcoming this limitation, so you can at least partially have the benefits of a RECFM V file on VM/CMS. We will write a routine that (without you doing anything) will build an index file, and use it when you access the file itself. The idea is the following: Whenever you build a variable record length file (ex. **test**), an index table for it is built automatically, containing for each record the displacement (in bytes) from the beginning of the file itself. As the table is RECFM F, it is easy to find the NNNth record, and, from its contents, to identify the REAL contents of record NNN. Pictorially:

```
  +-------+        +-------------+
1 |       |        |RRRRRRRR     |
2 |       |        |RRRRRRRRRRR  |
3 |ptr r3 |------->|RRRRRRRRRR   |
4 |       |        |R            |
5 |       |        |RRRRRRRRRRR  |
  |  (...)|        |   (...)     |
N |       |        |RRRRRRR      |
  +-------+        +-------------+
  test.IDX         test
  (RECFM F)        (RECFM V)
```

When should you use Indexed files?

The kind of applications that are well suited for indexed files are those where you read many times, RANDOMLY, a big file that you produce or refresh infrequently. An example is the 'phone book' of a company with hundreds of thousand of records, hashed in some particular form. Another example is a tape database, where the Volume ID of the tape is de-facto the key to accessing the file.

pro and cons for Indexed files.

Clearly, if the file is big, the indexed method makes a search for a random record as much as order of magnitudes faster. The drawback is that EVERY time you change the file, you need to refresh the index and make sure that no access to the file is made while the index is being built. In addition, the index itself uses space Ñ a 1 million record file requires an index file as large as 8MB. Of course if your records are big, this will be just a small percentage of the total disk space, but if the records are small, you risk the index file becoming bigger than the file itself.

The 'rw_filev' routine.

The 'rw_filev' routine is the 'kernel' of our discussion. It has three subfunctions: one for writing a file and, at the same time the file's index, a second for building an index for an existing file, and a third for reading a random record.

***** This section is:**

***** and will be available in next releases**

Handling of binary files.

It is sometimes useful to handle binary files.

The xfile routines

The two **xfile** routines **readbuf** and **writebuf** help you dealing with byte quantities.

```
                readbuf()
 ( FILE )      ----------->    fid.buffer
                               byte[size]
               <-----------
                writebuf()
```

```
+--------------------------------------------------------------------------+
| -- method......: readbuf
| -- purpose.....: read an entire file into a buffer
| --
|
|   method readbuf() public
|     rc = 0
|
|     do
|
|       fd   = File(name)
|
|       size = int fd.length()
|       off  = int 0
|
|       fis  = FileInputStream(fd)
|       in   = DataInputStream(fis)
|       buf  = byte[size]
|
```

```
        oprc = in.read(buf,off,size)
      catch er=ioException
        rc = 3
        say '(readbuf) ERROR:' er'.'
        return rc
      end
      if oprc = size
        then rc = 0
        else rc = 1
      buffer = buf
      return rc

+----------------------------------------------------------------------+
                                            xfile.nrx(Method:readbuf)
```

Resources... [Download the complete source for the xfile.nrx library]

The key instruction is:

```
oprc = in.read(buf,off,size)
```

where we read from the input stream **size** bytes, and we place them in a **byte** array called **buffer**.

```
+----------------------------------------------------------------------+
|  -- method......: writebuf
|  -- purpose.....: write an entire buffer onto a file
|  --
|    method writebuf() public
|      rc = 0
|      do
|        fd   = File(name)
|        size = int buffer.length
|        off  = int 0
|        fos  = FileOutputStream(fd)
|        out  = DataoutputStream(fos)
|        out.write(buffer,off,size)
|        out.flush()
|        oprc = out.size()
|        out.close()
|      catch er=ioException
|        rc = 3
|        say '(writebuf) ERROR:' er'.'
|        return rc
|      end
|      if oprc = size
|        then rc = 0
|        else rc = 1
```

```
    return rc

+--------------------------------------------------------------------+
                                              xfile.nrx(Method:writebuf)
```

Resources... [Download the complete source for the xfile.nrx library]

Examples

Let's look to some real examples.

```
+--------------------------------------------------------------------+
 -- test WRITE buffer
 --

 -- init a buffer, please
 --
 buf = byte[126]
 loop i = 1 to buf.length-1
   buf[i] = i
 end

 -- declare the output file
 --
 fn = 'twf.out'
 of = xfile(fn)

 -- point to the buffer space
 --
 of.buffer = buf

 -- OK, do the write
 --
 rc = of.writebuf()
 say 'Write of "'fn'" got RC:' rc'.'

 exit
+--------------------------------------------------------------------+
                                                             twb.nrx
```

Resources... [Download the source for the twb.nrx example]

This is how your output file will look like, looking it using **hedit** (see next section).

```
.......................................................................
rsl3pm1 (1) > java hedit twf.out
Welcome to hedit.
d0000000 - 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F  "................"
d0000016 - 1011 1213 1415 1617 1819 1A1B 1C1D 1E1F  "................"
d0000032 - 2021 2223 2425 2627 2829 2A2B 2C2D 2E2F  ".!"#$%&'()*+,-./"
d0000048 - 3031 3233 3435 3637 3839 3A3B 3C3D 3E3F  "0123456789:;<=>?"
d0000064 - 4041 4243 4445 4647 4849 4A4B 4C4D 4E4F  ".ABCDEFGHIJKLMNO"
d0000080 - 5051 5253 5455 5657 5859 5A5B 5C5D 5E5F  "PQRSTUVWXYZ[\]^_"
d0000096 - 6061 6263 6465 6667 6869 6A6B 6C6D 6E6F  "`abcdefghijklmno"
d0000112 - 7071 7273 7475 7677 7879 7A7B 7C7D 0000  "pqrstuvwxyz.|..."
<<< EOF >>>
cmd -> quit
All done.
rsl3pm1 (2) >
.......................................................................
```

Case study: hedit, a file dump/edit in HEX

Let's look at a program that allows us to dump and edit binary (and even text files) in HEX digits. The program, called **hedit** is available on the WEB source page for the tutorial.

The program does:

```
  - read the full file in storage
  - display the first "page" worth of dump
  - wait for commands
```

Some relevant code

The reading of the input file is issued with a simple call to the **readbuf** method.

```
fid = xfile(fn)
rc = fid.readbuf()
```

We now can use the array:

```
fid.buffer
```

to get the byte information of the file contents. Again, remember that:

```
fid.buffer.length                 -- buffer's length

fid.buffer[0]                     -- BUFFER
(...)                             --
fid.buffer[fid.buffer.length-1]  --
```

The method **linedis** is used to prepare the line that needs to be displayed.

```
+------------------------------------------------------------------------+
| -- method......: linedis                                               |78
| -- purpose.....: prepare a line                                        |79
| --
```

```
method linedis(bs=rexx,buf=byte[]) public static
  obh = ''
  obc = ''

  -- this logic is not perfect, might require rewriting
  -- get 2 bytes a time, in HEX and in CHAR
  --
  loop j = 0 to 14 by 2
    p1 = bs*16+j
    p2 = bs*16+j+1
    if p1 > buf.length - 1     -- past end of buffer
      then c1 = '00'           --
      else c1 = rexx buf[p1]
    if p2 > buf.length - 1
      then c2 = '00'
      else c2 = rexx buf[p2]
    obh = obh||c1.d2x(2)||c2.d2x(2)' '
    if c1 > 32 & c1 < 127      -- only char we can see
      then c1 = c1.d2c()       -- please
      else c1 = '.'
    if c2 > 32 & c2 < 127      -- ditto
      then c2 = c2.d2c()       --
      else c2 = '.'
    obc = obc||c1||c2
  end

  -- that's the full line
  --
  ptr = bs*16
  if dtyp = 'D'
    then ptr = 'd'ptr.right(7,'0')
    else ptr = 'd'ptr.right(7,'0')
  l = ptr '-' obh '"'obc'"'
  return l

+------------------------------------------------------------------+
                                          hedit.nrx(Method:linedis)
```

Resources... [Download the complete source for the hedit.nrx library]()

The **change** routine is used to perform a change over a subsequent set of bytes. You perform a change typing:

```
change START byte_string
```

like:

```
change 5 CAFE000067
```

```
+------------------------------------------------------------------------+
 -- method......: change
 -- purpose.....: change a set of bytes
 --
   method change(bs=rexx,up=rexx,buf=byte[]) public static
     -- some checks
     --
     if bs < 0 | bs > buf.length-1 then
       do
         say 'Invalid start byte.'
         return
       end
     list = up
     i = bs
     loop while list <> ''
       parse list nb +2 list
       say nb
       buf[i] = nb.x2d(2)
       i = i+1
     end
+------------------------------------------------------------------------+
                                              hedit.nrx(Method:change)
```

Resources... [Download the complete source for the hedit.nrx library]

The actual saving is performed by the method **save**, and the real kernel code is:

```
ofid = xfile(ofn)      -- define OUTPUT file
ofid.buffer = buf      -- point to buffer
rc = ofid.writebuf()   -- WRITE it!
```

```
+------------------------------------------------------------------------+
 -- method......: save
 -- purpose.....: saves a buffer
 --
   method save(sargs=rexx,buf=byte[]) public static
     parse sargs ofn .
```

```
    -- check if we have a filename and if it is not
    -- already there
    --
    if ofn = '' then
      do
        say 'Missing filename.'
        return
      end
    if xfile.fexist(ofn) then
      do
        say 'File "'ofn'" already exists. OK to overwrite?\-'
        parse ask.upper() answ
        if answ <> 'Y' then return
      end


    -- OK, go head
    --
    ofid = xfile(ofn)
    ofid.buffer = buf
    rc = ofid.writebuf()
    if rc = 0
      then say 'Buffer written OK to "'ofn'".'
      else say 'Problems writing "'ofn'".'

+-----------------------------------------------------------------------+
                                                   hedit.nrx(Method:save)
```

Resources... [Download the complete source for the hedit.nrx library]

Sample session

```
.......................................................................
rsl3pm01 (1) > java hedit rwf.out
Welcome to hedit.
d0000000 - 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F  "................"
d0000016 - 1011 1213 1415 1617 1819 1A1B 1C1D 1E1F  "................"
d0000032 - 2021 2223 2425 2627 2829 2A2B 2C2D 2E2F  ".!"#$%&'()*+,-./"
d0000048 - 3031 3233 3435 3637 3839 3A3B 3C3D 3E3F  "0123456789:;<=>?"
d0000064 - 4041 4243 4445 4647 4849 4A4B 4C4D 4E4F  ".ABCDEFGHIJKLMNO"
d0000080 - 5051 5253 5455 5657 5859 5A5B 5C5D 5E5F  "PQRSTUVWXYZ[\]^_"
d0000096 - 6061 6263 6465 6667 6869 6A6B 6C6D 6E6F  "`abcdefghijklmno"
d0000112 - 7071 7273 7475 7677 7879 7A7B 7C7D 0000  "pqrstuvwxyz.|..."
<<< EOF >>>

cmd ->help
Available commands are:
DOWN                 - move down one page.
UP                   - move up one page.
QUIT                 - exit program.
VERSION              - show program version.
GO nnnn              - go to location NNNN (DECimal).
TOP                  - go to top.
SAVE fn              - save buffer as "fn".
CHANGE start hexstr - change bytes from "start" with "hexstr".
```

```
cmd ->CHANGE 0 CAFEBABECAFEBABE
d0000000 - CAFE BABE CAFE BABE 0809 0A0B 0C0D 0E0F  "................"
d0000016 - 1011 1213 1415 1617 1819 1A1B 1C1D 1E1F  "................"
d0000032 - 2021 2223 2425 2627 2829 2A2B 2C2D 2E2F  ".!"#$%&'()*+,-./"
d0000048 - 3031 3233 3435 3637 3839 3A3B 3C3D 3E3F  "0123456789:;<=>?"
d0000064 - 4041 4243 4445 4647 4849 4A4B 4C4D 4E4F  ".ABCDEFGHIJKLMNO"
d0000080 - 5051 5253 5455 5657 5859 5A5B 5C5D 5E5F  "PQRSTUVWXYZ[\]^_"
d0000096 - 6061 6263 6465 6667 6869 6A6B 6C6D 6E6F  "`abcdefghijklmno"
d0000112 - 7071 7273 7475 7677 7879 7A7B 7C7D 0000  "pqrstuvwxyz.|..."
<<< EOF >>>
cmd -> quit

......................................................................
```

The rxfile package.

Availability

The **rxfile** package is available directly from the author, at the following URL:

http://www.geocities.com/SiliconValley/Park/4218/RXFILE.HTML
for rxfile
and
http://www.geocities.com/SiliconValley/Park/4218/
Marsiglietti's home page

***** This section is:**

***** and will be available in next releases**

Summary

A resume' of what we have seen in this chapter:

***** This section is:**

***** and will be available in next releases**

File: nr_13.html.

Last update was done on 18 May 1998 21:47:48(GMT +2).

The NetRexx Tutorial
- Threads

Threads

Introduction

*All modern operating systems are **multi-tasking**. This means than more than one program can concurrently run on the system at the same time. At least, this is how the user(s) perceive it: the operating system is responsible to allocate CPU cycles to the various processes, giving the impression that every process has, by itself, an entire CPU available.*

*In a **multi-threaded** system, you can divide each process into several components. These components are called **threads** or **light weight processes**.*

In this chapter we will analyse how we can have multiple threads running within our programs.

Definition of a Thread.

A **thread** is a component of a process. A **thread** is synonim of **light weight process**. Each thread executes a sequential set of instructions. The result of several threads running in parallel is a **concurrent** process.

```
| concurrent |   ---------+------->   thread 1
| process    |            |              >  Instruction 1.1
                          |              >  Instruction 1.2
                          |                 (...)
                          |              >  Instruction 1.n
                          |
                          +------->   thread 2
                          |              >  Instruction 2.1
                          |              >  Instruction 2.2
                          |                 (...)
                          |              >  Instruction 2.n
                          (...)
                          |
                          +------->   thread N
                                         >  Instruction N.1
                                         >  Instruction N.2
                                            (...)
                                         >  Instruction N.n
```

When you need to use Threads.

As we saw, **threads** allow to run multiple instances of the same process on your machine. But, you may ask, what's the real interest in doing this, if my machine has just one CPU? Aren't those processes going to compete for this unique resource?

I/O limited processes.

While it is true that CPU tied processes will benefit from a multiprocessor H/W environment, it is also true that, on many OS (notably UNIX and Windows/NT) the I/O subsystem is usually decoupled from the main CPU, so you can imagine to split your program in 2 parts: one which deals with the I/O, and one that deals with the CPU intensive work. A natural example is when you load a WEB containing pictures using Netscape. The text is immediately retrieved and the pictures are loaded while you can read, scroll, and do any other operation on the page itself (even if still incomplete). In principle, any picture retrieval can be a separate thread.

Daemons

A **daemon** is a process that runs on your system and acts as a **server**. As we will analyse in the next chapter, a **daemon** waits on a socket port for work to do. When it receives a request from a **client**, he dispatches the request. If the daemon is **single-threaded** he will not be capable to accept and serve other requests, till he has not finished the one is serving. Using threads, you'll be capable to concurrently serve many requests.

```
SERVER                SERVER THREAD
(...)
loop forever
  wait request
  dispatch request
       +------->      start thread
                      execute request
                      answer client
                      end thread
  (...)
end
```

monitoring

Another application of **threads** is monitoring of certain process. Some applications might hung (for a network problem, for example). You might want to put an external timeout to such occurrences.

Threads for UNIX users.

If you are a C (or C++) programmer working on UNIX platforms, and you want to create a process running in parallel with your main process, you would write something like:

```
+------------------------------------------------------------------------+
| /* example in Regina UNIX REXX */
| (...)
|
| /* issue the fork */
```

```
 i = fork()

 if i > 0 then
   do
     /* This is the parent process
      */
     say '(parent) Waiting.'
     rc = waitpid(i)
     say '(parent) Wait rc:' rc'.'
   end
 else
   do
     /* This is the children
      */
     'sleep 1'
     say '(child) Starting. Going to sleep.'
     'sleep 2'
     say '(child) Ending now.'
   end
 exit 0
+------------------------------------------------------------------------+
                                                              forkex1.rex
```

In NetRexx, like in Java, the approach is totally different. The above example will be written like:

```
+------------------------------------------------------------------------+
 --        package:  thrt1
 --        version:  1.000 beta
 --           date:  02 APR 1998
 --         author:  P.A.Marchesini
 --      copyright:  (c) P.A.MArchesini, 1998
 --   latest vers.:  http://wwwcn.cern.ch/news/netrexx
 --
 --
 class thrt0
   properties public

 -- method......: main
 -- purpose.....: timeout test
 --
   method main(args=String[]) public static
     arg = rexx(args)
     arg = arg
```

```
    say 'MAIN starts now.'
    child = thrt0handler()
    child.start()
    child.join()
    say 'MAIN ends'
    exit 0

 -- method......: thrt0handler
 -- purpose.....:
 --
 class thrt0handler extends Thread
   properties private

   method thrt0handler()

   method run() public
     say 'CHILD starts.'
     do
       sleep(2000)
     catch e = interruptedException
       say 'Got: "'e'".'
     end
     say 'CHILD ends.'
+-----------------------------------------------------------------------+
                                                               thrt0.nrx
```

Resources... [Download the source for the thrto.nrx example]

Thread API

A first practical example.

It is always a good practice to put a timeout on certain commands that you might issue inside your program. Infact, especially in a networked environment, a lot of things might "go wrong", and the program itself might hung forever.

The following example will show how to implement a timeout on a command that you issue from the command line.

```
+-----------------------------------------------------------------------+
 --       package:  thrt1
 --       version:  1.000 beta
 --          date:  02 APR 1998
 --        author:  P.A.Marchesini
 --     copyright:  (c) P.A.MArchesini, 1998
```

```
--  latest vers.:  http://wwwcn.cern.ch/news/netrexx
--
--
class thrt1
  properties public

-- method......: main
-- purpose.....: timeout test
--
  method main(args=String[]) public static
    arg = rexx(args)

    parse arg timeout command
    if timeout = " | command = " then
      do
        say 'Missing arguments.'
        say 'usage  : java thrt1 TIMEOUT_IN_SEC COMMAND'
        say 'example: java thrt1 5 sleep 6'

        exit 1

      end

    timeout = timeout*1000


    say 'MAIN starts now.'

    child = thrt1handler(command)
    child.start()
    child.join(timeout)
    if child.isAlive()
      then

        do

          say 'Children still alive. Killing it now.'
          child.stop()
          if child.isAlive()
            then say 'ERROR: stop() did not work.'
            else say 'OK: child killed.'

        end

      else say 'Children finished before timeout.'


    say 'MAIN ends'

    exit 0


-- method......: thrt1handler
-- purpose.....:
--
class thrt1handler extends Thread
  properties private
    command


  method thrt1handler(cmd=rexx)
    command = cmd


  method run() public
    say 'CHILD starts "'command'".'
```

```
     out = xexec(command)
     out = out
     say 'CHILD ends "'command'".'
+--------------------------------------------------------------------------+
                                                                  thrt1.nrx
```

Download the source for the thrt1.nrx example

You can try out the code typing:

```
# no timeout shown here
$ java thrt1 5 sleep 4
# timeout shown here
$ java thrt1 5 sleep 6
```

***** This section is:**

***** and will be available in next releases**

File: nr_14.html.

Last update was done on 18 May 1998 21:47:50(GMT +2).

The NetRexx Tutorial
- Socket and Networking

Socket and Networking

Introduction

```
*** This section is:
```


```
*** and will be available in next releases
```

Basic Concepts

The socket

Common Operations

Getting your HOST name.

One of the first things you will want to do, is to determine your machine name, i.e. doing in NetRexx what you normally get on your shell typing **hostname**.

You need to use the class **InetAddress**, in order to gather your current HOST name, with a call like:

```
host = InetAddress.getlocalHost()
```

The following **xsock** function will accomplish the job, striping out the (probably) unwanted address in numeric format.

```
+----------------------------------------------------------------------+
| -- method......: hostname                                            |
| -- purpose.....: get the hostname                                    |
| --
|
|   method hostname() public static                                    |
|     do
```

```
      host = InetAddress.getlocalHost()
    catch err = UnknownHostException
      say err
    end
    parse host name'/'
    return name

+------------------------------------------------------------------+
                                          xsock.nrx(Method:hostname)
```

Resources... [Download the complete source for the xsock.nrx library]

Client/Server applications

a small client-server application

A **client** is a process (or a program) that sends a message to a server process (or program); it requests the server to perform a task (also called service).

Client programs usually manage the user-interface portion of the application, validate data entered by the user, dispatch requests to server program, and sometimes execute some logic. The client-based process is the front-end of the application that the user sees and interacts with. The client process contains solution-specific logic and provides the interface between the user and the rest of the application system.

A **server** process executes the client request performing the task the client requested. Server programs generally: receive requests from client

- receive requests from client programs,
- execute database retrieval and updates,
- manage data integrity,
- dispatch responses to client requests

Sometimes server programs execute common or complex business logic. The server-based process "may" run on another machine on the network. This server could be the host operating system or network file server; the server is then provided both file system services and application services. Resuming, the server process acts as a software engine that manages shared resources such as databases, printers, communication links, or high powered-processors. The server process performs the back-end tasks that are common to similar applications.

In this section we examine a very small client-server application.

Our goal is to explain the basics of the client-server model, with the instructions that allows us to connect the client and the server. For this reason all the details about catching errors are ignored.

Our **server** is a socket application waiting on a port (we randomly choose the number **6001**). The server receives a line of information, constituted by 2 numbers (**n1** and **n2**). The server computes the sum (**n3**) and returns it to the client.

```
client                         server
side                           side
---------                      ----------

CLIENT    ---- n1 n2 -------> SERVER
                              (...)
                              n3 = n1+n2
                              (...)
CLIENT    <----- n3 --------- SERVER
```

```
+--------------------------------------------------------------------------+
 -- sserv
 -- a VERY primitive socket server
 --
 port   = int 6001
 listen = ServerSocket null
 client = Socket null

 do
   say 'Listening on port "'port'".'
   listen = ServerSocket(port)

   -- wait for a client
   -- get the numbers, add them, return to him
   --
   loop forever
     say 'Waiting connection'
     client = listen.accept();

     -- we got something
     --
     say 'Got request from' client.getInetAddress().getHostName() -
         ||':'client.getPort();
     in = DataInputStream(client.getInputStream());
     out = PrintStream(client.getOutputStream());
     line = in.readLine();
     if line = 'exit' then leave
     parse line n1 n2
     say 'Got "'line'".'
     sum = n1+n2
     out.println(sum);
   end
 catch e=IOException
   say 'Error:' e'.'
 end
 exit 0
+--------------------------------------------------------------------------+
```

sserv.nrx

Resources... [Download the source for the sserv.nrx example](http://www.netrexx.org/Tutorial/nr_15.html)

```
-- the SIMPLE client
--
parse arg line
if line = '' then exit 1
port = int 6001
host = 'pcl307'      -- should be your host name
s = Socket null;
do
   s = Socket(host,port);        -- hard wire it
   sin = DataInputStream(s.getInputStream());
   sout = PrintStream(s.getOutputStream());

   sout.println(line)                -- send command
   line = sin.readLine();            -- get answer
   say line
   s.close()
catch e=IOException
   say 'Error:'e'.'
end
exit 0
```

sclie.nrx

Resources... [Download the source for the sclie.nrx example]

A revised finger program

The following code is an implementation of the "classical" finger program as you find on UNIX boxes or on WIN/95 WIN/NT.

```
-- finger
--

import java.net
import java.io

VERSION          = 'v1r001'
AUTHOR           = '(c) P.A.Marchesini, ETHZ'
DEFAULT_PORT     = int 79;
CRLF = '\x0D\x0A'

parse arg uargs
if uargs = '-h' | uargs = '--help' | uargs = '' then
```

```
  do
    parse source . . myname'.'
    say myname 'version' VERSION AUTHOR
    say 'Purpose : sample implementation of a finger client.'
    say
    say 'java finger user@system'
    say
    exit 1;
  end

user = ''
if uargs.pos('@') <> 0
  then parse uargs user'@'node
  else node = uargs

-- issue the client socket command
--
s = Socket null;
do
  s = Socket(node, DEFAULT_PORT);
  sin   = BufferedReader(InputStreamReader(s.getInputStream()))
  sout  = PrintWriter(s.getOutputStream(),1)
  line  = String
  line  = user||crlf
  sout.println(line)
  loop forever
    line = sin.readLine();
    if (line = null) then do
      leave
    end
      say line
    end
catch e1 = IOException
  say '# Error from Socket function.'
  say '# Message is "'e1'".'
  say '# Abending.'
finally
  do
    if (s \= null) then s.close()
  catch e1 = IOException
    say '# Error from close.'
    say '# Message is "'e1'".'
    say '# Abending.'
  end
end
exit
```

finger1.nrx

Resources... [Download the source for the finger1.nrx example](#)

In the following session we'll develop an even shorter version of finger, using the "xsock" libraries.

The "xsock" library.

As I did in the previous (and following) chapters, instead of presenting "dumb" examples, I'll build a small library of socket methods. This library is called **xsock.nrx** and is available for download on the "usual" WEB directory for libraries.

It should give you enough programming examples to build (eventually) your own socket application. You can of course immediately use it, as shown in the **Using the xsock library** section.

The "open" method

```
+----------------------------------------------------------------------+
  -- method......: open
  -- purpose.....: open a socket
  --
    method open(host=Rexx,prot=Rexx) public

      -- check if the user entered a protocol or a simple
      -- port number
      --
      rc = 0
      if prot.datatype('D') = 0 then
        do
          -- he just entered a port with a name,
          -- try to find the port, unless abort
          dport = getservbyname(prot)
          if dport = -1 then
            do
              say 'Invalid protocol "'prot'".'
              exit 990
            end
          port = dport
          setprotdef(prot)
        end
      else
        do
          -- he just entered a numeric port
          -- we need to do nothing
          port = prot
```

```
      end

    -- do the REAL job
    --
    do
      s    = Socket(host, port);
      sin  = BufferedReader(InputStreamReader(s.getInputStream()));
      sout = PrintWriter(s.getOutputStream(),1);
    catch err = IOException
      say err
    end

+-----------------------------------------------------------------------+
                                              xsock.nrx(Method:open)
```

Resources... [Download the complete source for the xsock.nrx library]

The "getservbyname" method

```
+-----------------------------------------------------------------------+
|  -- method......: getservbyname
|  -- purpose.....:
|  --
|   method getservbyname(serv=Rexx) public static
|     table = 'DAYTIME 13 FTP   21   TELNET 23'  -
|             'FINGER  79 NNTP 119   IMAP   143' -
|             'HTTP    80'
|     serv = serv.upper()
|     res = -1
|     loop while table <> ''
|       parse table sn sp table
|       if sn = serv then return sp
|     end
|     return res
|
+-----------------------------------------------------------------------+
                                      xsock.nrx(Method:getservbyname)
```

Resources... [Download the complete source for the xsock.nrx library]

Using the xsock library

Finding info about a protocol

One of the best places to start is:

http://www.freesoft.org/Connected/RFC/index.html

Writing a daytime client.

```
+----------------------------------------------------------------------+
| /* simple daytime */
|
| parse arg node .
| if node = '' then node = 'shift1.cern.ch'
|
| so = xsock(node,'DAYTIME')
| so.receive()
| so.close()
|
| exit
+----------------------------------------------------------------------+
                                                           daytime.nrx
```

Resources... Download the source for the daytime.nrx example

Writing a finger client.

```
+----------------------------------------------------------------------+
| /* simple finger client */
|
| parse arg what .
| parse what user'@'node
| if node = '' then
|    do
|       say 'finger user@node'
|       exit 1
|    end
|
| so = xsock(node,'FINGER')
| so.send(user)
| so.receive()
| so.close()
|
| exit
+----------------------------------------------------------------------+
                                                            finger.nrx
```

Resources... Download the source for the finger.nrx example

Writing an FTP client using "sun.net.ftp".

The FTP support is contained in the package **sun.net.ftp**. The package allows easily to implement an FTP client (to GET and PUT files).

The API documentation can be found at:

http://www.java.no/javaBIN/docs/api/sun.net.ftp.FtpClient.html

The actual implementation of the FTP client wants to mimic the "standard" UNIX ftp command (which you can find also on Windows/NT). We will call our class **xftp** and it will be an extention of FtpClient (or sun.net.ftp.FtpClient if you prefer)

To get the functions in the package **sun.net.ftp**, we need to type:

```
import sun.net.ftp.FtpClient
import sun.net.ftp.FtpInputStream
import sun.net.TelnetInputStream
```

The basic functions are:

```
+-----------------------------------------------------------------------+
| -- method......: xget
| -- purpose.....: fetch the remote file
| --
|
|   method xget(fids=Rexx) public
|     rcclear()
|
|     parse fids fidr fidl
|
|     if fidl = '' then fidl = fidr
|
|
|
|     -- small check: if the local file is there, prompt the user
|     --
|
|     if xfile.fexist(fidl) & replace = 'NO' then
|       do
|
|         say 'Local file "'fidl'" already exists. OK to overwrite? (Y|
|         if ask.upper <> 'Y' then
|
|           do
|
|             say 'ABORTED by user.'
|             return
|
|           end
|
|       end
|
|
|     say 'Remote file........:' fidr'.'
|     say 'Local file.........:' fidl'.'
|     say 'Transfer type is...:' modeab'.'
|
```

```
    buff = byte[16000]
    t = timer()
    totsize = 0
    do
      os = FileOutputStream(fidl)
      tis = host.get(fidr)
      str = '(READING) Tranferred:' totsize 'bytes.'
      loop forever
        System.out.print(str'\x0D')
        n = tis.read(buff)
        if n = -1 then leave     -- there are no more bytes in tis
        totsize = totsize + n
        str = '(WRITING) Tranferred:' totsize 'bytes.'
        System.out.print(str'\x0D')
        os.write(buff,0,n)
        str = '(READING) Tranferred:' totsize 'bytes.'
      end
      System.out.print('                                        \x0D')
      say
      os.close()
      sec = t.elapsed()
      say 'Transferred "'totsize'" bytes in' sec 'seconds.'
    catch err = exception
      say 'ERROR:' err
      rcset(12)
    end

+-----------------------------------------------------------------------+
                                                 xftp.nrx(Method:xget)
```

Resources... [Download the complete source for the xftp.nrx library]

```
+-----------------------------------------------------------------------+
| -- method......: xput
| -- purpose.....: put the remote file
  --
   method xput(fids=Rexx) public
    rcclear()
    parse fids fidl fidr
    if fidr = '' then fidr = fidl

    -- small check: if the local file is not there
    --
    if xfile.fexist(fidl) = 0 then
      do
        say 'Local file "'fidl'" does not exist.'
        return
      end
```

```
    say 'Local file.........:' fidl'.'
    say 'Remote file........:' fidr'.'
    say 'Transfer type is...:' modeab'.'

    buff = byte[16000]
    t = timer()
    totsize = 0
    do
      is = FileInputStream(fidl)
      tos = host.put(fidr)
      str = '(READING) Tranferred:' totsize 'bytes.'
      loop forever
        System.out.print(str'\x0D')
        n = is.read(buff)
        if n = -1 then leave     -- there are no more bytes in is
        totsize = totsize + n
        str = '(WRITING) Tranferred:' totsize 'bytes.'
        System.out.print(str'\x0D')
        tos.write(buff,0,n)
        str = '(READING) Tranferred:' totsize 'bytes.'
      end
      System.out.print('                                     \x0D')
      say
      tos.close()
      is.close()
      sec = t.elapsed()
      say 'Transferred "'totsize'" bytes in' sec 'seconds.'
    catch err = exception
      say 'ERROR:' err
      rcset(13)
    end

+-----------------------------------------------------------------------+
                                                  xftp.nrx(Method:xput)
```

Resources... [Download the complete source for the xftp.nrx library]

```
+-----------------------------------------------------------------------+
  -- method......: xls
  -- purpose.....: list the remote directory (on screen)
  --
  method xls(t=rexx) public
    t = t
    rcclear()
    do
      tis = host.list()
      line = ''
      loop forever
        n = rexx tis.read
```

```
|        if n = -1 then leave     -- there are no more bytes in tis
|        if n = 10 then
|          do
|            say line
|            line = ''
|            iterate
|          end
|        line = line||n.d2c()
|      end
|      tis.close()
|    catch err = exception
|      say 'ERROR:' err
|      rcset(3)
|    end
|
+----------------------------------------------------------------------+
                                                         xftp.nrx(Method:xls)
```

Resources... [Download the complete source for the xftp.nrx library]

Another function (which is NOT in the standard FTP clients) is the **xmore**

```
+----------------------------------------------------------------------+
| -- method......: xmore
| -- purpose.....: type the file on terminal
| --
|   method xmore(fid=Rexx) public
|     rcclear()
|     nlin = 1
|     do
|       tis = host.get(fid)
|       line = ''
|       loop forever
|         n = rexx tis.read
|         if n = -1 then leave     -- there are no more bytes in tis
|         if n = 10 then
|           do
|             say line
|             line = ''
|             nlin = nlin+1
|             if nlin > pagesize then
|               do
|                 nlin = 1
```

```
                  say '+++ (ENTER to continue; Q to quit) \-'
                  if ask = 'Q' then
                    do
                      leave
                    end
                end
              iterate
            end
          line = line||n.d2c()
        end
      catch err = exception
        say 'ERROR:' err
        rcset(5)
      end
```

xftp.nrx(Method:xmore)

Resources... [Download the complete source for the xftp.nrx library]

A small program using the xftp class

As an example of usage of the **xftp** class, look at the following program:

```
 -- xftp1.nrx
 --    this program just lists the files from a anonymous server
 --    and fetches a big one.
 --
 h = xftp('asisftp.cern.ch')
 h.exec('user anonymous toto@test.cern.ch')
 h.exec('ls')
 h.exec('replace Y')
 h.exec('get README.cernlib')
 h.exec('get toto')
 say h.rc
 say h.globrc
 exit
```

xftp1.nrx

Resources... [Download the source for the xftp1.nrx example]

Writing a trivial NNTP client.

The **NNTP** protocol is described by **RFC 977** The NNTP specifies a protocol for the distribution, inquiry, retrieval, and posting of news articles using a reliable stream-based transmission of news among the ARPA-Internet community. NNTP is designed so that news articles are stored in a central database allowing a subscriber to select only those items he wishes to read. Indexing, cross-referencing, and expiration of aged messages are also provided.

We will implement a TRIVIAL NNTP client, using the **xsock.nrx** library. Our program **nnt** does allow the reading of a news article and the list of the available ones.

```
+------------------------------------------------------------------------+
 -- trivial NNTP client
 --
 parse arg group article .

 -- trivial checks
 --
 if group = '' then
   do
     say 'Please enter a group. (like "comp.lang.rexx").'
     exit 1
   end

 -- connect and get the greating message
 --
 node = 'news.cern.ch'     -- change this with your local news server
 so = xsock(node,'NNTP')
 so.readline()

 -- select the right group
 -- and check it's existence
 --
 so.send('group' group)
 nn = so.readline()
 parse nn rc . first last .
 if rc <> 211 then
   do
     say 'Sorry but group "'group'" is not active.'
     exit 3
   end

 -- OK, now we can
 -- - get all the headers
 -- - get the article body
 if article = ''
```

```
   then cmd = 'xhdr subject' first'-'last
   else cmd = 'article' article
 so.send(cmd)
 nn = so.readline()
 parse nn rc .
 if rc > 240 then
   do
     say 'Sorry, but article "'group':'article'" is not available.'
     exit 4
   end

 so.receive(",")

 -- that's all
 --
 so.close()

 exit 0
+-------------------------------------------------------------------------+
                                                                   nnt.nrx
```

Resources... [Download the source for the nnt.nrx example]

```
..................................................................
rsl3pm1 (68) java nnt comp.lang.rexx
19083 rexx under DOS?
19084 Re: Program Priority in REXX or C - How Set?
19085 Suggestions on how to keep a "table" OUTSIDE of Rexx?
19086 Re: Suggestions on how to keep a "table" OUTSIDE of Rexx?
(...)

rsl3pm1 (69) java nnt comp.lang.rexx 20132
From: Dave
Newsgroups: comp.os.os2.setup.misc,comp.lang.rexx
Subject: Lost rxFTP
(...)
I re-installed OS/2 this weekend and now rxFTP doesn't work.When I try
(...)

..................................................................
```

Executing NNTP commands interactively

Some small modifications to the above program will allow you to execute commands in an interactive way, in a line mode like shell.

Once you started the command with **java nntp1**, just type **help** and the server will answer with the available commands.

```
+-------------------------------------------------------------------------+
```

```
-- simple INTERACTIVE
-- news client
--
node = 'news.cern.ch'  -- change it to your local news server

-- connect to the NEWS server
--
so = xsock(node,'NNTP')
parse so.readline() . welcome 'ready'
say welcome

-- wait for commands
--
onelineansw = 'next group'
i = 1
loop forever
  say 'NNTP@'node' ['i'] > \-'
  i = i+1
  cmd = ask
  if cmd = 'quit' | cmd = 'exit' then leave
  so.send(cmd)
  line = so.readline()
  say line
  parse cmd cmd rest
  if onelineansw.wordpos(cmd) <> 0 then iterate
  parse line cc rest
  if cc > 300 then
    do
      iterate
    end
  so.receive(",")
end

-- we're done
--
so.close()
say 'Bye.'
exit
+-------------------------------------------------------------------------+
                                                                   nnt1.nrx
```

Resources... [Download the source for the nnt1.nrx example]

- **Writing a trivial IMAP client.**

RFC 1064 describes the IMAP protocol. IMAP stands for **Interactive Mail Access Protocol**. The idea is that your mail messages are stored into a server. Your client connects to the server, so you can read your mail using a PC, a UNIX workstation, a MAC or whatever without storing the messages locally.

The protocol is a bit more complicate than the above ones: all messages must be prefixed by a TAG that identify the command. The TAG is in the format "ANNN".

```
client                          server
------                          ------

A001 command1 -------------->
                                Answer
                                (...)
              <--------------   A001 status1

A002 command2 -------------->
                                Answer
                                (...)
              <--------------   A002 status2
```

The small program that follows implements (again) a trivial IMAP client. You need to change the **mail.cern.ch** address with the address of the IMAP server of your Organization.

```
+----------------------------------------------------------------------------+
 -- simple INTERACTIVE
 -- news client
 --
 node = 'mail.cern.ch'  -- change it to your local news server

 -- connect to the NEWS server
 --
 so = xsock(node,'IMAP')
 say so.readline()

 -- wait for commands
 --
 i = 1
 loop forever
   say 'IMAP@'node' ['i'] > \-'
   i = i+1
   cmd = ask
   if cmd = 'help' then
     do
       say 'LOGIN userid passwd'
       say 'SELECT mailbox        (ex. SELECT INBOX)'
       say 'LOGOUT'
       say 'FETCH sequence data   (ex. FETCH 1 RFC822)'
       say 'see RFC1064'
       iterate
```

```
      end
    tag = 'A'||i.right(3,'0')
    so.send(tag cmd)
    loop forever
      line = so.readline()
      say line
      parse line atag .
      if tag = atag then leave
    end
    if cmd = 'logout' then leave
  end

  -- we're done
  --
  so.close()
  say 'Bye.'
  exit
+----------------------------------------------------------------------+
                                                              imapt.nrx
```

Resources... [Download the source for the imapt.nrx example](#)

URLs and WEB pages

The basic concepts

The URL

The **URL** identifies uniquely a document on the Network.

URL is an acronym. It stands for **Uniform Resource Locator**; it is the address (or if you prefer, the reference) of an Internet resource, usually an HTML document.

You probably saw thousands of URLs when "surfing" the Network, in the form of:

http://java.sun.com/javastation/jstn.html

In this URL, like in all other URLs, we can identify 4 basic components, which are:

```
http://java.sun.com:80/javastation/jstn.html
----   ------------ -- --------------------
 |           |       |          |
 |           |       |          +------------> filename
 |           |       |                         (pathname of the file
 |           |       |                          on the server machine)
```

```
     |            |       |
     |            |       +---------------> port number
     |            |                         (default is 80, so you
     |            |                          can happily ommit :80
     |            |                          here shown for educational
     |            |                          purposes)
     |            |
     |            +-----------------> host name
     |                                (the name of the machine
     |                                 where the document is located)
     |
     +-----------------------> protocol identifier
                               (can be HTTP,
                                       FTP
                                       News
                                       Gopher
                                       File  - for local files)
```

The HTTP daemon

The most common type of WEB documents are handled by **HTTP daemons**. Those daemons are waiting (usually) on port 80, and accept an handfull of commands.

The most common command is **GET** followed by a path name. The daemon will answer sending back to the client the selected document.

```
 +----------+
 |          |
 | HTTP     |
 | daemon   |---------+
 |          | PORT 80 |    <--------   GET Welcome.html
 +----------+---------+

     (sends                --------->
      Welcome.html)
```

Retriving WEB documents, the basic code.

As we saw, the HTTP protocol used by the HTTP daemons is something very similar to the protocols we already examined for the various socket daemons we encountered so far.

We can write a small program to retrieve a WEB page, using pure sockets:

```
+----------------------------------------------------------------------+
| -- REALLY primitive HTTP client
| -- use basic sockets
| --
| class w3dmp public
|   properties constant
|     DEFAULT_PORT    = int 80;
|
|   method usage public static
|     say "Usage:   java w3dmp URL"
|     say "Example: java w3dmp http://wwwcn.cern.ch/Welcome.html"     |
```

```
  exit 1


method main(args=String[]) public static
  -- Get the arguments
  argsl = Rexx(args)
  if argsl = '' then
    do
      usage()
    end

  -- get the URL components
  -- very easy with PARSE
  parse argsl protocol'://'node'/'document
  parse node node':'port

  -- basic checks
  if protocol <> 'http' then
    do
      say 'Only HTTP protocol, please.'
      exit 1
    end
  if node = '' then
    do
      say 'Missing server name.'
      exit 2
    end
  if port = '' then port = DEFAULT_PORT

  -- do the real job
  s = Socket null;
  do
    s    = Socket(node,port);
    sin  = DataInputStream(s.getInputStream());
    sout = PrintStream(s.getOutputStream());

    cmd  = 'GET' '/'||document
    sout.println(cmd)
    line = String
    loop forever
      -- Read a line from the server.
      line = sin.readLine();
      -- Check if connection is closed (i.e. for EOF)
      if (line = null) then leave
      -- And write the line to the console.
      Say line
```

```
      end
    catch e1=IOException
      System.err.println(e1)
    finally
      do
        if (s \= null) then s.close()
      catch e2=IOException
       e2=e2
      end
    end

 exit 0
+--------------------------------------------------------------------+
                                                           w3dmp.nrx
```

Resources... [Download the source for the w3dmp.nrx example](#)

The parsing of the URL components is done (of course) with two **parse** instructions, in order to correctly extract the (optional) port number, in case it is different from 80.

The code can be made even shorter, using the already discussed **xsock** library functions.

```
+--------------------------------------------------------------------+
 -- REALLY primitive HTTP client
 -- use basic sockets (and xsock library)

 -- Get the arguments
 if arg = '' then
   do
     say "Usage:   java w3dmp URL"
     say "Example: java w3dmp http://wwwcn.cern.ch/Welcome.html"
     exit 1
   end

 -- get the URL components
 -- very easy with PARSE
 parse arg protocol'://'node'/'document
 parse node node':'port

 -- basic checks
 if protocol <> 'http' then
   do
     say 'Only HTTP protocol, please.'
     exit 2
   end
```

```
if node = '' then
  do
    say 'Missing server name.'
    exit 3
  end
if port = '' then port = 'HTTP'

-- do the real job
so = xsock(node,'HTTP')
so.send('GET /'document)
so.receive()
so.close()

exit 0
```

w3dmp1.nrx

Resources... [Download the source for the w3dmp1.nrx example]

***** This section is:**

***** and will be available in next releases**

Summary

Let's resume what we saw in this chapter.

File: nr_15.html.

Last update was done on 18 May 1998 21:47:51(GMT +2).

The NetRexx Tutorial
- Interface with the system

Interface with the system

Introduction.

One of the most important points for coding effective NetRexx programs, is the ability to dialogue with the operating system. Thus we want to be capable of executing OS commands, getting the output in a variable or in a array, starting other processes, and so on.

Calling System Commands.

Sooner or later you will find yourself in the need to call a System Command from your NetRexx code, and have the output (if any) stored somewhere.

You should also note that you have ALWAYS an output from a System Command or Program. This is the Return Code **rc** from the Command itself.

Pictorially:

```
+---------+           +--------------------+
| O.S.    |<----------| NetRexx            |
| (OS/2   |           | Program            |
|  UNIX   |---------->|....................|---------> display
|  W95    |  result   | .                  |            immediately
|  WNT)   |           | ....> save in a    |
|         |           | .     VARIABLE     |
|         |           | ....> save in an   |
|         |           |       ARRAY        |
+---------+           +--------------------+
```

As we have just stated, we will distinguish three cases:

- Call a command.
- Call a command and get the result in a variable.
- Call a command and get the result into an array.

We want also to make some decisions depending on the result of the command we just executed. If the command fails, i.e. exits with a $RETURN, not zero we want to be able to choose to continue, inform the user, or abort.

Related JAVA classes

```
java.lang.Process
java.lang.Runtime
```

Calling a command immediately

This is probably the easiest instance: you want to execute an OS command (or a program). This means you will write:

```
(...)
cmd = 'zip files.zip file1 file2'
r  = Runtime.GetRuntime()
p  = r.exec(cmd)
(...)
```

It is ALWAYS a good practice to check the return code **rc**: a command or a program can fail for many reasons, and your program must be prepared for such eventualities. Note that if you do not check the **rc**, the program will happily continue with the following instructions. So we modify the above code as:

```
(...)
cmd = 'zip files.zip file1 file2'
r  = Runtime.GetRuntime()
p  = r.exec(cmd)
rc = p.exitValue()
if rc <> 0 then
  do
    say 'Command "'cmd'" failed with rc:' rc'.'
    exit rc
  end
(...)
```

This will allow us to check if the **zip** command in the above example didn't crash for a disk full problem, or for a missing input file.

Note that in the 2 above examples the output of the command is NOT displayed

A final WARNING

WARNING: I feel necessary to warn you about a potential problems if you abuse of calls to System Commands.

You should NEVER use a call to System Commands if your call can be implemented in Java itself. So you should not (if you're a UNIX user) do:

```
--
-- NEVER DO THIS !!!
--
ls = xexec('ls -l toto','VAR','ABORT')
parse ls.out . . . . size .
```

This code is, infact, no portable (DOS and Windows) do not know about "ls".

NOTE: if you want to implement "ls" you do something like:

```
l = String[]
f = File(".")
l = f.list()
loop i = 0 to l.length-1
  say l[i]
end
```

lls.nrx

Resources... [Download the source for the lls.nrx example]

Simple examples

Execute a System command

```
-- syex1.nrx
-- SYstem EXec
--
class syex1 public

  method main(args=String[]) public static

    arg = Rexx(args)
    parse arg cmd

    -- do the REAL job
    --
    do
      rtim = Runtime.GetRuntime()
      proc = rtim.exec(cmd)
      dis  = DataInputStream(proc.getInputStream())

      loop forever
        line = dis.readline()
        if line = NULL then leave
        say line
      end
      rc   = proc.waitFor()
```

```
      say 'Return code:' rc'.'
    catch err = IOException
      say 'ERROR:' err
    end
    exit 0
```

syex1.nrx

Resources... [Download the source for the syex1.nrx example]

Execute an "interactive" System command

Some programs, like the following one, might require some "interactive" input.

```
n = 0
loop forever
  n = n+1
  say 'Please enter something (quit to QUIT)'
  parse ask line
  if line = 'quit' then leave
  say n '>>>' line
end
```

interact.nrx

Resources... [Download the source for the interact.nrx example]

It would be nice if it was possible to make (when needed) the input "automatic". This small example shows how.

```
-- syex2.nrx
-- SYstem EXec
--
class syex2 public

  method main(args=String[]) public static

    -- this is the interactive command
    cmd = 'java interact'

    -- do the REAL job
    --
    do
      rtim = Runtime.GetRuntime()
```

```
|       proc = rtim.exec(cmd)
|       dos  = PrintStream(proc.getOutputStream())
|       dis  = DataInputStream(proc.getInputStream())
|       dos.println('help')
|       dos.println('quit')
|       dos.close()
|
|       loop forever
|         line = dis.readline()
|         if line = NULL then leave
|         say line
|       end
|       rc   = proc.waitFor()
|       say 'Return code:' rc'.'
|     catch err = IOEXception
|       say 'ERROR:' err
|     end
|     exit 0
+--------------------------------------------------------------------+
                                                              syex2.nrx
```

Resources... [Download the source for the syex2.nrx example]

The "key" instruction is:

```
dos   = PrintStream(proc.getOutputStream())
```

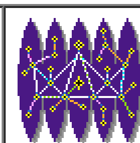
where we get an OUTPUT stream to the process **proc**. We now can simulate the keyboard input, which we do via:

```
dos.println('help')
dos.println('quit')
```

so all is like if you were typing **help** and **quit** from your keyboard.

The xexec method

```
+--------------------------------------------------------------------+
| -- method......: xexec
| -- purpose.....: constructor
| --
|   method xexec(cmd=String,dest=Rexx,oner=Rexx) public
|     dest = dest.upper()                   -- uppercase params
|     oner = oner.upper()
|     valid_dest = 'ARRAY SCREEN VAR NULL'
|     valid_oner = 'WARNING ABORT IGNORE'
```



```
    -- setting the defaults
    --
    if dest = '' then dest = default_dest
    if oner = '' then oner = default_oner

    -- check if the parms are OK
    --
    if valid_dest.wordpos(dest) = 0 then
      do
        say 'Error: "'dest'" is not a valid destination.'
        exit 1
      end
    if valid_oner.wordpos(oner) = 0 then
      do
        say 'Error: "'oner'" is not a valid ONERROR action.'
        exit 1
      end

    -- do the real job
    --
    do
      r  = Runtime.GetRuntime()
      p  = r.exec(cmd)
      cr = DataInputStream(BufferedInputStream(p.getInputStream()))

      -- Output handling
      --
      lines   = 0
      out = ''
      j = 0
      loop forever
        s = cr.Readline()
        if s = NULL then leave
        if dest.wordpos('SCREEN')
          then say s
        if dest.wordpos('VAR')
          then out = out s
        if dest.wordpos('ARRAY')
```

```
        then
          do
            j = j+1
            line[j] = s
          end
      end
    lines = j
    line[0] = lines

    -- Return code handling
    --
    rc = p.exitValue()
    if rc <> 0 then
      do
        select
          when oner = 'WARNING' then
            do
              say 'WARNING: rc=' rc 'from "'cmd'".'
            end
          when oner = 'ABORT' then
            do
              say 'WARNING: rc=' rc 'from "'cmd'".'
              say 'ABORTING.'
              exit 5
            end
           otherwise NOP
        end
      end
  catch error = IOException
    say error
end

method xexec(cmd=Rexx,dest=Rexx) public
  this(cmd,dest,default_oner)

method xexec(cmd=Rexx) public
  this(cmd,default_dest,default_oner)

```

```
+------------------------------------------------------------------------+
                                                   xsys.nrx(Method:xexec)
```

Resources... [Download the complete source for the xsys.nrx library]

Some application: a simple "shell"

With the knowledge we developped in this chapter, we can now imagine to write a simple shell

```
+------------------------------------------------------------------------+
 --       package:  xshell
 --       version:  1.000 beta
 --          date:  23 FEB 1997
 --        author:  P.A.Marchesini
 --     copyright:  (c) P.A.MArchesini, 1997
 --  latest vers.:  http://wwwcn.cern.ch/news/netrexx
 --
 --  This program is free software; you can redistribute it and/or mod
 --  it under the terms of the GNU General Public License as published
 --  the Free  Software  Foundation;  either version 2 of the License,
 --  (at your option) any later version.
 --
 --  This program is distributed in the hope that it will be useful,
 --  but WITHOUT ANY WARRANTY; without even the  implied warranty of
 --  MERCHANTABILITY or FITNESS FOR A  PARTICULAR  PURPOSE.  See the
 --  GNU General Public License for more details.
 --
 --  You should have received a copy of the GNU General Public License
 --  along with  this  program;  if not,  write  to  the Free Software
 --  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 --


 -- class xshell
 --   This class implements a "shell" environment, something like
 --   'zsh' or 'bash' (with very less functions!)
 --
 class xshell


   properties public static
   properties private static
     version   = 'v0r000 beta'
     copyright = '(c) 1997 Pierantonio Marchesini, ETH Zurich'
     contact   = 'Pierantonio.Marchesini@cern.ch'

 -- method......: shell
 -- purpose.....: constructor
 --
   method xshell() public
     version = version          -- make NetRexx happy
     copyright = copyright      -- ditto
     contact = contact          -- ditto

 -- method......: main
 -- purpose.....: just run typing "java shell"
```

```
 --
   method main(args=String[]) public static
     args = args

     -- Initialization
     --
     cmdno = 1
     rc = 0
     validlcmds = 'history'
     validecmds = 'ls pwd java' -
                  'ftp cp help dir'
     host = xsock.hostname()    -- get my host,pls
     extracmd = ''
     his = history(100)

     loop forever
       say host '['his.counter()':'rc'] 'extracmd'\-'
       todo = ask
       if extracmd <> ''
         then todo = extracmd||todo

       --  check special cases
       --
       if todo = '' then iterate
       if todo = 'exit' | todo = 'quit' then leave
       if todo.left(1) = '!' then
         do
           parse todo '!'rest
           select
             when rest = '!' then ptr=cmdno-1
             otherwise ptr = rest
           end
           if ptr < 1 then ptr = 1
           extracmd = his.retrieve(ptr)
           iterate
         end

       extracmd = ''
       cmdno = cmdno+1
       his.save(todo)
       parse todo cmd arg
       arg = arg

       -- process local commands
       --
```

```
      if validlcmds.wordpos(cmd) <> 0 then
        do
          select
            when cmd = 'history' then his.dump(10)
            otherwise say 'Sorry. "'cmd'" is not yet implemented.'
          end
          iterate
        end

      -- check for .class
      --
      if xfile.fexist(cmd'.class') then
        do
          todo = 'java' todo
          cmd = 'java'
        end


      -- process external commands
      --
      if validecmds.wordpos(cmd) = 0 then
        do
          say 'Invalid command "'cmd'".'
          iterate
        end
      c = xexec(todo,'SCREEN','IGNORE')
      rc = c.rc
    end
    exit 0
+---------------------------------------------------------------------+
                                                              xshell.nrx
```

Resources... [Download the source for the xshell.nrx example]

***** This section is:**

***** and will be available in next releases**

File: nr_16.html.

Last update was done on 18 May 1998 21:47:53(GMT +2).

The NetRexx Tutorial
- Process Control and Exceptions

Process Control and Exceptions

Introduction

In this chapter we will analyse how to better control the program flow of a NetRexx application.

Basic Concepts

Exception

The **exception** is a mechanism that allows you to (eventually) change the flow of control whenever some important or unexpected event (usually an error) occurs in your program. You then can try to cope with the problem (usually alerting the user that the problem has occurred), and avoid major disasters (usually exiting the program).

Exception Handling

Although NetRexx allows you to ignore (even explicitly) an exception, it is always a good idea to handle it, especially in the debugging phase of a program.

Exceptions in real life.

One way to happily generate exceptions, is to avoid any checking of input data. Not performing any validation on input data is REALLY a bad programming. In this case we'll avoid the checking on purpose, just to see what can happen.

Look at the following code:

```
+----------------------------------------------------------------------+
| -- expp1.nrx
| -- WARNING: this is bad programming: no checks on input
| --          data are performed
| --
| parse arg n
```

```
inv = 1/n
say 'Inverse is:' inv
exit 0
+-------------------------------------------------------------------+
                                                           expp1.nrx
```

Resources... [Download the source for the expp1.nrx example]

This is definitely a bad code since:

- we do not check for an empty input
- we do not check for non-numeric input
- we do not check for zero input

So let's the fun begin and try to run some examples:

```
.......................................................
-- this is OK
sp069.marchesi ~/src/java/Java/bin [0:18] java expp1 1
Inverse is: 1

-- this is divide by 0
sp069.marchesi ~/src/java/Java/bin [0:19] java expp1 0
netrexx.lang.DivideException: Divide by 0
        at netrexx.lang.Rexx.dodivide(Rexx.nrx:1648)
        at netrexx.lang.Rexx.OpDiv(Rexx.nrx:1557)
        at expp1.main(expp1.nrx:6)

-- non numeric input
sp069.marchesi ~/src/java/Java/bin [0:20] java expp1 popo
java.lang.NumberFormatException: popo
        at netrexx.lang.Rexx.dodivide(Rexx.nrx:1647)
        at netrexx.lang.Rexx.OpDiv(Rexx.nrx:1557)
        at expp1.main(expp1.nrx:6)

-- no input at all
sp069.marchesi ~/src/java/Java/bin [0:21] java expp1
java.lang.NumberFormatException:
        at netrexx.lang.Rexx.dodivide(Rexx.nrx:1647)
        at netrexx.lang.Rexx.OpDiv(Rexx.nrx:1557)
        at expp1.main(expp1.nrx:6)
.......................................................
```

Those messages are really scaring, aren't they?

Handling exceptions: catch

Suppose that we have a block of code that, like in the example above, might generate an exception.

So:

```
(...)
-- this code might generate an exception
--
...
BLOCK_OF_CODE
...
(...)
```

In NetRexx, if you want to handle exceptions, you'll write the above code as:

```
(...)
do
    -- this code might generate an exception
    --
    ...
    BLOCK_OF_CODE
    ...
  catch variable_name = EXCEPTION_NAME
    CODE_TO_RUN_IN_CASE_OF_EXCEPTION
end
(...)
```

In a nutshell, you put your code into a **do ... end** clause, and add a **catch** instruction. Program flow will be passed to CODE_TO_RUN_IN_CASE_OF_EXCEPTION in case of any EXCEPTION_NAME encountered

The special instruction is **catch**. Catch is (usually) followed by a statement of the format:

```
catch error = EXCEPTION_NAME
  say 'EXCEPTION_NAME: got error:' error'.'
```

Always run a piece of code: finally.

Sometimes it is important to catch the exception, but also to be guaranteed that some "critical" code is run, whatever happens to the program, i.e. if the exception is cached or not. Think about a file lock, for example, that you MUST clean, in case of a program crash.

You use the **finally** statement, which you are guaranteed is ALWAYS run.

```
(...)
do
    -- this code might generate an exception
    --
    ...
    BLOCK_OF_CODE
    ...

  catch variable_name = EXCEPTION_NAME
    CODE_TO_RUN_IN_CASE_OF_EXCEPTION

  finally
    CODE_TO_RUN_ALWAYS_AND_ANYWAY

end
(...)
```

Resume

To resume what we saw so far:

```
...
do
  ...                       -- This code MIGHT
  BLOCK_OF_CODE             -- generate an exception
  ...                       --
  catch [ err = ] EXCEPTION1
```

```
      ...                        --
      CODE FOR EXCEPTION1        --
      ...                        --
    catch [ err = ] EXCEPTION2
      ...                        -- You can catch as many
      CODE FOR EXCEPTION2        -- exceptions you want
      ...                        --
    finally
      ...                        -- code ALWAYS run
      CODE FOR EXCEPTION1        --
      ...                        --
  end
```

A revisited 'bad-programmer' inverse computation program

Let's apply what we saw so far to the example we initially made:

```
+------------------------------------------------------------------------+
| -- expp2.nrx
| -- WARNING: this is bad programming: no checks on input
| --          data are performed
| --
| parse arg n
| ok = 0
| do
|   inv = 1/n
|   say 'Inverse is:' inv
|   ok = 1
| catch DivideException
|   say 'Division exception'
| catch ex=NumberFormatException
|   say 'Number "'n'" bad for division.'
|   say 'message is "'ex'".'
| end
| if ok
|   then say 'Division is OK.'
|   else say 'Problems found.'
| exit 0
+------------------------------------------------------------------------+
                                                                  expp2.nrx
```

Resources... [Download the source for the expp2.nrx example]

```
.........................................................
sp069.marchesi ~/src/java/Java/bin [0:29] java expp2 1
Inverse is: 1
Division is OK.
sp069.marchesi ~/src/java/Java/bin [0:29] java expp2 0
Division exception
Problems found.
sp069.marchesi ~/src/java/Java/bin [1:30] java expp2 toto
Number "toto" bad for division.
message is "java.lang.NumberFormatException: toto".
Problems found.
```

```
sp069.marchesi ~/src/java/Java/bin [1:31]
.........................................................
```

Output the stack trace information

The **stack trace** contains the information about your program at the time the exception occurred. In particular, it shows you the line number where the problem did occur. This might help you to solve a LOT of problems.

If you **catch** the exception, and you want to see the stack trace, you just add the following line:

```
do
  (...)
  catch er = EXCEPTION
    say 'ERROR: EXCEPTION'
    er = printStackTrace()
end
```

NOTE: printStackTrace() outputs to System.err, If you want the output to System.out, just type:

```
er = printStackTrace(System.out)
```

Changing the format of the Stack Trace

Maybe you do not like the output format of the stack trace. This function will show you how to change it:

```
+------------------------------------------------------------------------+
| -- method......: dump
| -- purpose.....:
| --
|
|   method dump(e=Exception) public static
|     -- trace buffer
|
|     trace = Rexx('')
|
|
|     -- get the error message
|
|     --
|
|     err = e.tostring()
|
|
|     -- printStackTrace outputs to a PrintStream
|     -- we connect a PipedInput to grab the output
|     --
|
|     pout = PipedOutputStream()
|     pin  = PipedInputStream()
|     pin.connect(pout)
|     out  = PrintStream(pout)
|     in   = DataInputStream(pin)
|
|
|     -- get the stack
|
|     --
|
```

```
    e.printStackTrace(out)

    j = 0

    loop while in.available() <> 0
      str = in.readLine()
      parse str 'at' rest

      if rest = '' then iterate

      j = j+1

      trace[j] = rest

    end

    trace[0] = j

    parse trace[j] ':'line')'
    say '(dump) Error found line..:' line'.'
    say '(dump) Message is........:' err'.'
    say '(dump) Full dump follows.:'
    say

    loop i = trace[0] to 1 by -1

      parse trace[i] p1'('prog':'line')'
      if line = '' then iterate

      p1 = '('p1.space()')'
      say '(dump)' prog.left(12) p1.left(30) 'line:' line.right(5)
    end

    say


+-----------------------------------------------------------------------+
                                                  xsystem.nrx(Method:dump)
```

Resources... [Download the complete source for the xsystem.nrx library]

If we now modify our simple buggy program, like this:

```
+-----------------------------------------------------------------------+
 -- expp2.nrx
 -- WARNING: this is bad programming: no checks on input
 --          data are performed
 --
 parse arg n
 ok = 0
 do
   inv = 1/n
   say 'Inverse is:' inv
   ok = 1
 catch er1 = DivideException
   xsystem.dump(er1)
 catch er2 = NumberFormatException
   xsystem.dump(er2)
 end
```

```
  if ok
    then say 'Division is OK.'
    else say 'Problems found.'
  exit 0
+-----------------------------------------------------------------------+
                                                                expp3.nrx
```

Resources... [Download the source for the expp3.nrx example]

we get the following result:

```
.....................................................................
sp069.marchesi ~/src/java/Java/bin [0:69] java expp3 0
(dump) Error found line..: 8.
(dump) Message is........: netrexx.lang.DivideException: Divide by 0.
(dump) Full dump follows.:

(dump) expp3.nrx    (expp3.main)                  line:     8
(dump) Rexx.nrx     (netrexx.lang.Rexx.OpDiv)     line:  1557
(dump) Rexx.nrx     (netrexx.lang.Rexx.dodivide)  line:  1648

Problems found.
.....................................................................
```

Summary.

***** This section is:**

***** and will be available in next releases**

File: nr_17.html.

Last update was done on 18 May 1998 21:47:54(GMT +2).

The NetRexx Tutorial
- Database Operations

Database Operations

Introduction

An interface to some primitive database functions is available as a NetRexx extension.

***** This section is:**

***** and will be available in next releases**

Use NetRexx with JDBC

The following code atom shows how to use NetRexx with JDBC

```
+----------------------------------------------------------------------+
| -- original sample from Gerhard Hofstaetter (hofg@edvg.co.at)        |
| -- and posted on ibm-netrexx                                          |
| -- use NetRexx with JDBC
| --
| import java.net.URL                                                   |
| import java.sql.                                                      |
| import ibm.sql.                                                       |
|
| class jdbct1
|
|
|    method jdbct1                                                      |
|       class.forName('ibm.sql.DB2Driver')                             |
|
|    method main( args = string[]) static                               |
|       jdbct1()
|
|
|       -- set database as URL
|
|       url = 'jdbc:db2:edvr0s3'                                        |
|
```

```
      -- connect to database
      connect = DriverManager.getConnection(url)


      -- retrieve data from the database
      say 'Retrieve some data from the database...'
      sqlstmt = connect.createStatement()
      resultset = -
         sqlstmt.executeQuery('select tabschema, tabname' -
                              'from syscat.tables')

      -- display the result set
      -- resultset.next() returns false when there are no more rows
      say 'Received results:'
      loop while resultset.next()
         owner = resultset.getString(1)
         table = resultset.getString(2)
         say 'Owner =' owner 'Table =' table
      end


      resultset.close()
      sqlstmt.close()
      connect.close()
+------------------------------------------------------------------+
                                                        jdbct1.nrx
```

Resources... [Download the source for the jdbct1.nrx example]

***** This section is:**

***** and will be available in next releases**

File: nr_18.html.

Last update was done on 18 May 1998 21:47:56(GMT +2).

The NetRexx Tutorial - Applets

Applets

```
*** This section is:
```

Under Construction

```
*** and will be available in next releases
```

Creating and running your first Applet.

I want to show you how to create and run a very simple Applet. As in the "Hello World!" example, the issue is not really the code (that giving the colours I use I think you'll just run only one time), but the whole procedure.

The steps can be resumed:

- step 1: Create a **class** that extends the Java **Applet**. You'll need to define **at least** two methods: an **init** method and a **paint** method. This class will be the usual **.nrx** file that you know how to compile.
- step 2: Create an **html** file with the right applet definitions.
- step 3: run **appletviewer** over the above HTML file.

The Applet.

```
+----------------------------------------------------------------------+
| -- Your very first applet
| --
| class aphello extends Applet
|   properties private
|     fo = Font
|     XMAX = 500
|     YMAX = 500
|
|   method init
|     resize(XMAX,YMAX)
|     fo = Font("Helvetica",fo.BOLD,36)
|
```

```
|
|   method paint(g=graphics)
|     g.setFont(fo)                    -- set font
|     g.setColor(Color.Pink)           -- all pink, pls
|     g.fillrect(0,0,XMAX,YMAX)        --
|     g.setColor(Color.Yellow)         -- write yellow
|     g.drawString('Hello there!',10,200) -- message
+-----------------------------------------------------------------------+
                                                           aphello.nrx
```

Resources... [Download the source for the aphello.nrx example]

The HTML.

```
+-----------------------------------------------------------------------+
| <html>
| test
| <applet code="aphello.class" height=100 width=100 length=100>
| </applet>
| </html>
+-----------------------------------------------------------------------+
                                                          aphello.html
```

The full procedure as typed in.

```
.....................................................................
             --
             -- build the Applet
             --
rsl3pm1 (68) edit aphello.nrx

             --
             -- compile it
             -
rsl3pm1 (69) java COM.ibm.netrexx.process.NetRexxC aphello

             --
             -- edit the HTML
             --
rsl3pm1 (70) edit aphello.html

             --
             -- try it out
             --
rsl3pm1 (71) appletviewer aphello.html
.....................................................................
```

File: nr_20.html.

Last update was done on 18 May 1998 21:47:57(GMT +2).

The NetRexx Tutorial
- Graphical Interfaces

Graphical Interfaces

```
*** This section is:
```


```
*** and will be available in next releases
```

File: nr_21.html.

Last update was done on 18 May 1998 21:47:57(GMT +2).

The NetRexx Tutorial
- Advanced Graphics

Advanced Graphics

```
*** This section is:
```


```
*** and will be available in next releases
```

File: nr_22.html.

Last update was done on 18 May 1998 21:47:58(GMT +2).

The NetRexx Tutorial
- Advanced Networking

Advanced Networking

In this chapter we will analyse some of the most recent goodies available in JDK 1.1, and consequently in NetRexx.

In this chapter we will analyse:

- *The Remote Method Invocation (RMI)*
- *The Java/NetRexx Servlets*

Basic Concepts

Remote Method Invocation

The RMI (Remote Method Invocation) is a technique by which an object on SYSTEM A can call a method in an object on SYSTEM B, located somewhere else in the network.

All the sending of parameters, and retrieving of the result will happen in a transparent way, so that the user (and, before him, the application developper) has the feeling that the method was called locally (like any other method we saw so far).

So far we saw how the methods are pieces of code run locally by an object:

```
MACHINE A
---------

object OBJ

  method METHOD
    (...)

    code for METHOD  <--- runs
                          locally

    (...)
```

Using RMI we move **code for METHOD** to be remote.

```
MACHINE A                 MACHINE B
---------                 ---------

object OBJ

  method METHOD             method METHOD
```

```
      (...)
               ========>
                              code for METHOD   <--- runs
                                                     remote
               <=======
      (...)
```

This "extention" of the method across the Network is done using sockets; but all the programming details are hidden to the programmer, who just have to realize that, being the call remote, the chances that "something-goes-wrong" are bigger, so he MUST be more carefull for error handling.

The Client/Server Model

The following picture might help understanding the Client/Server in the RMI implementation.

```
+------------------------+             +-------------------------+
| CLIENT    rmicl.nrx    |             | SERVER   rmise.nrx      |
|                        |             |                         |
| a = obj.method()  +--------+    +-----+                        |
|                   |rem obj|    |     |                         |
|                   | method|===>|     |     +---------------+   |
|                   |       | S  |     |     | actual OBJ    |   |
|                   |       | O  |     |===> |   method      |   |
|                  S|       | C  |     |<=== |     return    |   |
|                  T|       | K  |     |     +---------------+   |
|     return       U| return|<===|     |S                        |
|                  B|       | E  |     |T                        |
|                   +-------+ T  +-----+U                        |
|                        |    S        |  B                       |
+------------------------+             +-------------------------+
```

As you see, the REAL object exists on the SERVER; from the SERVER's point of view, the object IS the SERVER.

First example: a time RMI.

This is probably the simplest code you can try, in order to implement an application using the Remote Method Invocation.

We'll write a program to grab the time information from another machine (even if, for practical purposes, the example will run Client and Server on the same machine).

Define the remote interface

```
+----------------------------------------------------------------------------+
| class Time public implements java.rmi.Remote interface                     |
|
|
|   method sayTime() returns String signals java.rmi.RemoteException         |
+----------------------------------------------------------------------------+
                                                                      Time.nrx
```

Resources... [Download the source for the Time.nrx example](http://www.netrexx.org/Tutorial/nr_23.html)

Write the Implementation Class

```
import java.rmi.
import java.rmi.server.UnicastRemoteObject

class TimeImpl public extends UnicastRemoteObject implements Time

  properties private
    myname

  method TimeImpl(s=String) signals RemoteException
    super();
    myname = s;

  method sayTime() returns String
    return 'Hello from' myname 'at' xsys.time('N')

  method main(a=String[]) public static

    -- Create and install a security manager
    System.setSecurityManager(RMISecurityManager());

    do
      obj = TimeImpl("TimeServer");
      Naming.rebind("//pcl307/TimeServer", obj);
      say "TimeServer bound in registry";
    catch e=Exception
      say "TimeImpl err: " + e.getMessage();
    end
```

TimeImpl.nrx

Resources... [Download the source for the TimeImpl.nrx example]

Write an application that uses the Remote Service

```
import java.rmi.         -- MUST be here!

class TimeCl public

  method main(arg=String[]) public static
    arg = arg      -- keep NR silent

    do
```

```
|        obj = Time Naming.lookup("//pcl307/TimeServer")
|        message = obj.sayTime();
|      catch e=Exception
|         say "TimeCl exception:" e.getMessage()
|      end
|
|      say  message
|
|      exit
|
+-----------------------------------------------------------------------+
                                                                  TimeCl.nrx
```

Resources... [Download the source for the TimeCl.nrx example]

Putting all those pieces together

Provided you have the three above .nrx files stored in the same directory, in order to run the example, you have to issue the following commands, in your shell

```
...............................................................
  -- 1. edit the sources and change "pcl307" to your
  --    node name

> edit TimeCl.nrx
> edit TimeImpl.nrx

  -- 2. Compile the 3 programs
  --

> java COM.ibm.netrexx.process.NetRexxC Time.nrx
> java COM.ibm.netrexx.process.NetRexxC TimeCl.nrx
> java COM.ibm.netrexx.process.NetRexxC TimeImpl.nrx

  -- 3. Generate the stubs
  --

> rmic TimeImpl

  -- 4. Start the registry
  --          (WNT;W95)  start rmiregistry
  --          (ditto)    javaw rmiregistry
  --          (UNIX)     rmiregistry &

> start rmiregistry

  -- 5. Start the Server part
  --

> java TimeImpl

  -- 6. On another window, you can run the
  --    client

> java TimeCl
...............................................................
```

First real example: a remote controlled VOLTAGE controller

What we saw so far might appear a little "too much" for such a simple application. In fact, it is.

In the following example we use what we have learnt to build an application where objects last LONGER than the lifetime of the client application.

The code for Interface, Server and Client

```
class volt public implements java.rmi.Remote interface

  method get(ch=int) returns int signals java.rmi.RemoteException
  method set(ch=int,value=int) signals java.rmi.RemoteException
```

volt.nrx

Resources... [Download the source for the volt.nrx example]

```
-- voltimpl.nrx
-- voltage controller implementation
--

import java.rmi.
import java.rmi.server.UnicastRemoteObject

class voltimpl public extends UnicastRemoteObject implements volt

  properties private
    myname

    channel = int[100]

  method voltimpl(s=String) signals RemoteException
    super();

    myname = s;


  -- set a channel

  method set(ch=int,value=int)
    say myname 'channel:' ch 'set to:' value
    channel[ch] = value

  -- fetch a value

  method get(ch=int) returns int
    return channel[ch]

  -- main method

  method main(a=String[]) public static

    -- Create and install a security manager
    System.setSecurityManager(RMISecurityManager());

    do

      obj = voltimpl("voltageserver");
      Naming.rebind("//pcl307/voltageserver", obj);
      say "voltageserver bound in registry";
```

```
    catch e=Exception
      say "voltimpl err: " + e.getMessage();
    end

+------------------------------------------------------------------------+
                                                              voltimpl.nrx
```

Resources... [Download the source for the voltimpl.nrx example]

```
+------------------------------------------------------------------------+
 -- voltcl.nrx
 -- client example
 --
 import java.rmi.          -- MUST be here!

 class voltcl public

   method main(args=String[]) public static
     arg = rexx(args)
     parse arg act ch val   -- get args
     act = act.upper()      -- upperacase the action

     do
       -- get the remote object
       obj = volt Naming.lookup("//pcl307/voltageserver")

       -- do the job
       if act = 'SET' then  -- set a channel
         do
           obj.set(ch,val)
           n = obj.get(ch)
         end
       if act = 'GET' then  -- get a channel
         do
           n = obj.get(ch)
         end
     catch e=Exception
        say "voltcl exception:" e.getMessage()
     end
     say 'Channel' ch 'value:' n'.'
     exit 0
+------------------------------------------------------------------------+
                                                                voltcl.nrx
```

Resources... [Download the source for the voltcl.nrx example]

Build it

We saw already how to build an RMI application, so I just show again the commands.

```
.............................................................
> edit voltcl.nrx
> edit voltimpl.nrx
> java COM.ibm.netrexx.process.NetRexxC volt.nrx
> java COM.ibm.netrexx.process.NetRexxC voltCl.nrx
> java COM.ibm.netrexx.process.NetRexxC voltimpl.nrx
> rmic voltimpl
> start rmiregistry
> java voltimpl
> java voltcl set 2 33
> java voltcl get 2
  33          <===  This is MAGIC!
.............................................................
```

Remote File Access

Let's now analyse a real case study. We want to implement some (tough primitive) file access method. Our client application will then be capable to access a Server's file just like if the file was local.

The files

For this project we again need 4 files, which are:

```
rfile.nrx         - the Interface
rfileimpl.nrx     - the Implementation
rfileserv.nrx     - the Server's part
rfileclie.nrx     - the Client's part
```

Interface

```
+--------------------------------------------------------------------------+
| -- rfile.nrx
| --    Remote File Access
| --    Interface part
| --
|
| class rfile public implements java.rmi.Remote interface
|   method setfilename(s=String) signals java.rmi.RemoteException
|   method exists() returns int signals java.rmi.RemoteException
|   method list() returns String[] signals java.rmi.RemoteException
|   method cat() returns String[] signals java.rmi.RemoteException
+--------------------------------------------------------------------------+
                                                                  rfile.nrx
```

Resources... [Download the source for the rfile.nrx example]

Implementation

```
-- rfileimpl.nrx
--   Remote File Access
--   Implementation part
--

import java.rmi.
import java.rmi.server.UnicastRemoteObject

class rfileimpl public extends UnicastRemoteObject implements rfile

  --
  properties private
    myname
    fid = File
    fname

  -- constructor
  method rfileimpl(s=String) signals RemoteException
    super();
    myname = s;

  -- set the filename
  method setfilename(fn=String)
    say myname 'selects' fn
    fname = fn
    fid = File(fn)

  -- check if file exists
  method exists() returns int
    return fid.exists()

  -- list a directory
  method list() returns String[]
    return fid.list()

  -- cat a file
  method cat() returns String[]
    d = xfile(fname)        -- use xfile
    rc = d.read()           --
    say '(cat) File "'fname'" read rc:' rc'.'

    -- I need this till I cannot return REXX
    nl = d.lines
    s = String[nl]
    loop i = 1 to d.line[0]
      s[i-1] = d.line[i]
    end
    return s
```

```
rfileimpl.nrx
```

Resources... [Download the source for the rfileimpl.nrx example](#)

Server

```
-- rfileserv.nrx
--   Remote File Access
--   Server code
--

import java.rmi.
import java.rmi.server.UnicastRemoteObject

class rfileserv public

  -- main method
  method main(a=String[]) public static

    myname = "remfileaccess"
    mynode = "pcl307"

    -- Create and install a security manager
    System.setSecurityManager(RMISecurityManager());

    do
      obj = rfileimpl(myname);
      Naming.rebind('//'mynode'/'myname, obj);
      say 'Bind of' myname 'OK.'
      say 'Node is' mynode '.'
      say 'SERVER now ready for connections.'
      say 'HIT CNTRL-C to ABORT'
    catch e=Exception
      say 'rfileserv error:' + e.getMessage();
    end
```

rfileserv.nrx

Resources... [Download the source for the rfileserv.nrx example](#)

Client

```
-- rfileclie.nrx
--   Remote file Access
```

```
--   Client part
--
import java.rmi.          -- MUST be here!

class rfileclie public

  properties public static
    fn

  method help() public static
    say 'implemented commands are:'
    say 'java rfileclie ls <FILE>'
    say '               state <FILE>'
    say '               cat <FILE>'
    exit 6

  method ls(fid=rfile) public static
    if fid.exists() = 0 then
      do
        say 'Sorry: remote file "'fn'" does not exist.'
        exit 1
      end
    dd = String[]
    dd = fid.list()
    loop i = 0 to dd.length - 1
      say dd[i]
    end

  method cat(fid=rfile) public static
    if fid.exists() = 0 then
      do
        say 'Sorry: remote file "'fn'" does not exist.'
        exit 1
      end
    dd = String[]
    dd = fid.cat()
    loop i = 0 to dd.length - 1
      say dd[i]
    end


  method main(args=String[]) public static
    arg = rexx(args)
    parse arg cmd fn
```

```
    if cmd = 'help' then
      do
        help()
      end
    do
      -- get the remote object
      fid = rfile Naming.lookup("//pcl307/remfileaccess")

      -- do the job
      if fn = '' then fn = '.'
      fid.setfilename(fn)
      select
        when cmd = 'ls' then ls(fid)
        when cmd = 'cat' then cat(fid)
        otherwise say 'Unimplemented command.'
      end
    catch e=Exception
       say "rfileclie exception:" e.getMessage()
    end
    exit 0
+-------------------------------------------------------------------+
                                                        rfileclie.nrx
```

Resources... [Download the source for the rfileclie.nrx example](#)

Additional sources of documentation.

RMI is a rather new topic (at least it is in June 1997). You might find some additional information at:

http://chatsubo.javasoft.com/current/doc/tutorial/getstart.doc.html
http://www.widget.com/ggainey/java/rmi_talk/rmi_talk.html

Problems and limitations

Stubs not updated.

If you forget to update the stubs, since you forgot to run "rmic IMPLEMENTATION_FILE", you get a message like:

```
java.lang.IllegalAccessError: unimplemented interface method
  at ...
  (... follows tracedump ...)
  ...
```

You should then run rmic IMPLEMENTATION_FILE to have the correct interface.

rmiregistry problem

You might get an error like:

```
java.lang.NumberFormatException: SERVER error
  at (TRACE)
```

You usually clear it stopping and restarting the rmiregistry program.

Method returning REXX variable

There are currently problems if the method returns a REXX type. The message you get is something like:

```
client exception:
  Error unmarshaling return
  nested exception is:
  java.io.NotSerializableException: netrexx.lang.Rexx
```

***** This section is:**

***** and will be available in next releases**

File: nr_23.html.

Last update was done on 18 May 1998 21:47:59(GMT +2).

The NetRexx Tutorial
- Full OOP projects

Full OOP projects

Introduction

In this chapter I'd like to show some "real" projects developped using OOP techniques and then implemented using NetRexx.

Those projects are far to be completed; this explains the quotes I used in the previous sentence using the word "real". But they are definitely larger than the examples showed so far.

Where possible, I'll give some comparison code to show the implementation using other OO languages, notably C++.

The projects developped are:

- *A Finite Element Method Analysis Program*
- *A Mail Client Application*

A Finite Element Method Analysis program

A Mailer Application

Mail Headers

You find all the information you need about the MAIL headers in the RFC 822 (STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES), available at:

```
ftp://ds.internic.net/rfc/rfc822.txt
```

`*** This section is:`

`*** and will be available in next releases`

`File: nr_24.html.`

Last update was done on 18 May 1998 21:48:00(GMT +2).

The NetRexx Tutorial
- Additional Instructions

Additional Instructions

Introduction.

We collect here all those instructions that have we have not so far had the pleasure to comment on or show, because they did not fall into any of the categories we looked at. This does not imply that they are any less important.

Arrays

The xarray function package

```
+--------------------------------------------------------------------+
| -- method......: dump
| -- purpose.....: dump array's contents
| --
|
   method dump(a=rexx[],name) public static
     len = a.length
|
     fil = name'(dim='len')'
     fil = fil.left(10)
     oval  = 'DUMMY'
|
     skip  = 0
|
     dosay = 0
|
     loop i = 0 to len-1
|
       if a[i] = NULL
|
         then val = 'NULL'
|
         else val = a[i]
|
       dosay = 0
|
       if val = oval
|
         then skip = skip+1
|
         else dosay = 1
|
       if i = len-1 then dosay = 1
|
       if dosay then
|
         do
|
```

```
          if skip > 0 then
            do
              if (i = len - 1) then skip = skip - 1
              if skip > 0 then say fil '(...' skip 'lines not display
              if i <> len-1 then say fil '['i-1']' oval
              skip = 0
            end
          say fil '['i']' val
        end
      oval = val
      fil = ' '.copies(10)
    end
    say

+--------------------------------------------------------------------------+
                                                 xarray.nrx(Method:dump)
```

Resources... [Download the complete source for the xarray.nrx library]

```
+--------------------------------------------------------------------------+
| -- method......: copy
| -- purpose.....: copy array's contents
| --
|   method copy(a=rexx[],b=rexx[]) public static
|     System.arraycopy(a,0,b,0,a.length)
|
+--------------------------------------------------------------------------+
                                                 xarray.nrx(Method:copy)
```

Resources... [Download the complete source for the xarray.nrx library]

Code example no.1

Let's use the routines we've built in the **xarray** library.

```
+--------------------------------------------------------------------------+
| -- arrex1.nrx
| -- Simple example of array handling
| --
|
| a = rexx[10]            -- define array dimentions
| b = rexx[12]            --
|
| xarray.init(a,'')      -- initialize array a
| a[0] = 'line1'          -- with some values
| a[1] = 'line2'
```

```
 xarray.dump(a,'a')       -- look at a and b
 xarray.dump(b,'b')

 xarray.copy(a,b)         -- copy a to b

 b[0] = 'XXXXXXXXXXXX'
 xarray.dump(a,'a')       -- look at a and b
 xarray.dump(b,'b')

 exit 0
+----------------------------------------------------------------------+
                                                            arrex1.nrx
```

Resources... [Download the source for the arrex1.nrx example]

Non NetRexx Arrays

In this small example we consider how to deal with non NetRexx (Rexx) arrays.

```
+----------------------------------------------------------------------+
 -- tstring.nrx
 -- small example of String[] handling
 --

 class tstring1 public

   method t1() returns String[] public static
     s = String[2]

     s[0] = 'Francesca'
     s[1] = 'Elisabetta'
     say s.length

     return s


   method main(args=String[]) public static
     arg = rexx(args)

     parse arg .


     in = String[100]
     in = t1()

     loop i = 0 to in.length - 1

       say in[i]

     end


     line = rexx(in)

     say line
```

```
    exit 0
+----------------------------------------------------------------------+
                                                           tstring1.nrx
```

Resources... [Download the source for the tstring1.nrx example]

Byte Arrays conversion methods

Byte array handling is a bit tedious. This is the motivation of the methods described in **xarray**.

In a byte array, infact, the quantities are, from the NetRexx point of view, stored as signed integer, so it will be:

```
a[0] = '01'      1
a[1] = '81'   -127
a[2] = 'FE'     -2
a[3] = '41'     65
```

In order to convert it to HEX, for example, you'll need to follow the procedure:

```
ch = rexx a[2]   -> -2
ch = ch.d2x(2)   -> FE
```

The methods we've developed are:

```
xarray.ba2x(array,start,length)
xarray.ba2c(array,start,length)
xarray.ba2d(array,start,length)
xarray.bagrepx(array,HEX,start)
```

Using the **a[]** array, we can look at some simple examples, like:

```
                               will give:
                               ----------
xarray.ba2x(a,1,2)          -> 81FE

xarray.ba2c(a,3,1)          -> A

xarray.bagrepx(a,'81FE',0) -> 2
```

REMARK: those methods are SLOW! I should probably find a faster way to implement them. Suggestions are welcome!

Some sample routines

```
+----------------------------------------------------------------------+
| -- method......: ba2x                                                |
| -- purpose.....: ByteArray to HEX                                    |
| --
|   method ba2x(a=byte[],start=rexx,length=rexx) public static        |
```

```
    ostr = ''
    loop i = start to start + length - 1
      ch = rexx a[i]
      xch = ch.d2x(2)
      ostr = ostr||xch
    end
    return ostr

+--------------------------------------------------------------------------+
                                                    xarray.nrx(Method:ba2x)
```

Resources... [Download the complete source for the xarray.nrx library]

The following method will search an ARRAY for an HEX quantity, which you write in the form (for example):

```
'A0FF'
```

the methods returns the value of the FIRST occurrence (from the start) of the HEX string.

```
+--------------------------------------------------------------------------+
| -- method......: bagrepx
| -- purpose.....: grep an HEX qty in a ByteArray
| --
|
|  method bagrepx(a=byte[],search=rexx,start=rexx) public static
|    l = search.length()
|    b = byte[l/2]
|
|
|    -- convert the HEX string
|    -- to decimal
|
|    --
|    list = search
|    i = 0
|    loop while list <> ''
|      parse list nb +2 list
|      b[i] = nb.x2d(2)
|      i = i+1
|    end
|
|    lend = a.length - 1
|    match = 0
|    loop i = start to lend
|      if a[i] == b[0] then
|        do
```

```
        match = 1
        loop j = 1 to b.length - 1
          if b[j] <> a[i+j] then
            do
              match = 0
              leave
            end
        end
        if match then leave
      end
  end
  if match
    then return i
    else return -1

+-----------------------------------------------------------------------+
                                               xarray.nrx(Method:bagrepx)
```

Resources... [Download the complete source for the xarray.nrx library]

Example: a JPEG info grabber

To apply the methods described above, let's write a small program that finds the size, in pixels, of a JPEG picture file.

Without going into details, we say that a JPEG (Joint Photographic Experts Group) file is a binary file. The header looks like:

```
Marker: FF D8
      : FF E0 00 10
    ID: 4A 46 49 46 (== JFIF)
```

JFIF stands for JPEG File Interchange Format. The marker we look at is 'FFC0' that contains the image size.

```
+-----------------------------------------------------------------------+
 -- grab info on JPEG file
 --
 parse arg fn .
 if fn = '' then
   do
     say 'usage: java jpginfo FILEID'
     exit 1
```

```
   end

 -- read input file;
 -- if ERROR, abort
 --
 fid = xfile(fn)
 rc = fid.readbuf()
 if rc <> 0 then
   do
     say 'Error reading file "'fn'".'
     exit 2
   end
 buf = fid.buffer

 -- check for signature
 --
 si = xarray.ba2c(buf,6,4)
 if si <> 'JFIF' then
   do
     say 'Unable to find signature.'
     exit 3
   end

 -- find the marker
 --
 p = xarray.bagrepx(buf,'FFC0',0)
 if p = -1 then
   do
     say 'Could not locate "FFC0" mark.'
     exit 4
   end

 -- all OK,
 -- get the info
 --
 w  = xarray.ba2d(buf,p+7,2)
 h  = xarray.ba2d(buf,p+9,2)
 say h'*'w

 exit 0
+-----------------------------------------------------------------------+
                                                            jpginfo.nrx
```

Resources...

Download the source for the jpginfo.nrx example

Additional Readings.

For the graphics formats, look at:

http://wsspinfo.cern.ch/faq/graphics/fileformats-faq/part3

The Independent JPEG Group archive on ftp.uu.net contains an on-line copy of the JFIF specification and additional JPEG information. Look at:

```
ftp://ftp.uu.net/graphics/jpeg/jfif.ps.gz
ftp://ftp.uu.net/graphics/jpeg/jpeg.documents.gz
```

The xsys.time() function.

We use the **xsys.time()** function to get the local time in the format "hh:mm:ss" (hours, minutes, seconds). The **xsys.time()** function can be called with arguments that change the output format a little. The complete list of arguments is:

```
N  -   hh:mm:ss          - Normal (the default);
C  -   hh:mmxx           - Civil
L  -   hh:mm:ss.uuuuu    - Long

H  -   hh                - Hours
M  -   mmmm              - Minutes
                           (minutes since midnight)
S  -   ssss              - Seconds
                           (seconds since midnight)
```

The best way to see all those options is to write a small program that shows all of them. The small **timeexa1** program does it.

```
-- simple test of the xsys.time()
-- function

list = 'N C H M S Z'
loop while list <> ''
  parse list kind list
  say xsys.time(kind)
end
exit 0
```

timeexa1.nrx

Resources... Download the source for the timeexa1.nrx example

Here is what you get if you run it. The output will of course depends on the time at which you run it.

```
.....................................................................
rsl3pm1 (68)  eti1
Option "N" returns:  17:46:30
Option "H" returns:  17
Option "M" returns:  1066
Option "S" returns:  63990
Option "L" returns:  17:46:30.121
Option "C" returns:  5:46pm
Option "Z" returns:  GMT
rsl3pm1 (69)
.....................................................................
```

Time your programs with a timer class.

The problem

You usually need to measure time intervals in your programs. In this way you can measure how long an operation takes to perform.

You can use the Java **System** class **System.currenttimemillis()**time method, and measure the time differences yourself.

```
  now = System.currenttimemillis
```

This method returns the current time in milliseconds GMT since the EPOCH (00:00:00 UTC, January 1, 1970).

The numbers returned are BIG

The idea.

We define then a **timer** class. The two basic instructions are:

```
(...)
-- define a timer
timer1 = timer()
(...)

(...)
-- get the elapsed time
elapsed = timer1.elapsed()
   (to get the elapsed time since the LAST reset)
(...)

(...)
-- reset the timer
zero = timer1.reset()
   (to reset the timer)
(...)
```

The timer class implementation

```
+-------------------------------------------------------------------+
| -- method......: elapsed
| -- purpose.....: returns the elapsed time in SSSS.MMM
| --
|   method elapsed() public returns Rexx
|     current = System.currenttimemillis
|     numeric digits 16
|     delta = current - start
|     delta = delta/1000
|     numeric digits 9
|     delta = delta.format(NULL,3)
|     return delta
|
+-------------------------------------------------------------------+
                                              xsys.nrx(Method:elapsed)
```

Resources... [Download the complete source for the xsys.nrx library]

```
+-------------------------------------------------------------------+
| -- method......: reset
| -- purpose.....: reset the timer; returns '0.000' seconds
| --
|   method reset() public returns Rexx
|     start = System.currenttimemillis
|     return '0.000'
|
+-------------------------------------------------------------------+
                                                xsys.nrx(Method:reset)
```

Resources... [Download the complete source for the xsys.nrx library]

The date() function.

```
*
* WARNING:
*   REXX's date function
*   will be implemented in xsys v2.000.
*
```

Use the **date()** instruction to get the current local date in the format 'dd Mmm yyyy'. As we saw for **time()** also **date()** has many options. These are:

```
 N  -  dd Mmm yyyy      - Normal;

 E  -  dd/mm/yy         - European;
 U  -  mm/dd/yy         - USA;
 O  -  yy/mm/dd         - Ordered;

 C  -  ddddd            - days (so far)
                          in this century;
 D  -  ddd              - say (so far)
                          in this year;

 S  -  yyyymmdd         - Standard;
```

As for **time()**, we do the same exercise also for **date()**. I simply write the results, since the program is easily modified from **eti1**.

```
.....................................................................
rsl3pm1 (75) eda1
Option "N" returns:  () 5 Feb 1995

Option "E" returns:  () 05/02/95
Option "U" returns:  () 02/05/95
Option "O" returns:  () 95/02/05

Option "S" returns:  () 19950205
Option "C" returns:  () 34734
Option "D" returns:  () 36

Option "M" returns:  () February
Option "W" returns:  () Sunday

rsl3pm1 (76)
.....................................................................
                                                               eda1.out
```

The xdate() function

The NetRexx **xsys** function **xdate** (for eXtended DATE) is **the** function for performing all imaginable operations related to date. The original code was developed for VM/CMS by Bernard Antoine of CERN/CN in IBM/370 assembler code. The version I describe here is a porting of that code done by its original author in pure NetRexx.

This code is totally platform independent, and is available on the WWW NetRexx Tutorial page (in the xsys library). **xdate** can be used in two ways:

```
 - to display a certain date
   in a given output format
   (ex: xsys.xdate('TODAY','U') )

 - to perform a conversion of a date
   from one format to another
   (ex: xsys.xdate('E','01/12/95','J') )
```

The valid input formats are:

```
  D,ddd              - number of days since the beginning of the year
                       format;
```

```
J,[yy]yyddd       - julian format;
S,[yy]yymmdd      - sorted format;
O,[[yy]yy/]mm/dd  - ordered format;
E,dd/mm[/[yy]yy]  - European format;
U,mm/dd[/[yy]yy]  - USA format;
B,nnnn            - number of days since  the January 1st, 0001
                    format;
C,nnnn            - number of days since the beginning of the
                    century format;
K,[yy]yyww        - format according to ISO 2015 & 2711;
I,nnnn            - incremental format;
I,+nnnn
I,-nnnn
```

Output_format may be any single character accepted by the REXX DATE function:

```
O   to obtain the date in ordered form, i.e. yy/mm/dd
U   to obtain the date in USA form, i.e. mm/dd/yy
E   to obtain the date in European form, i.e. dd/mm/yy
S   to obtain the date in 'sorted' form, i.e. yyyymmdd
J   to obtain the date in julian form, i.e. yyddd
B   to obtain the number of days since  the January 1st, 0001
C   to obtain the number of days since the beginning of
    the century
D   to obtain the number of days since the beginning of the year
M   to obtain the month name
W   to obtain the weekday name
```

In addition, XDATE also accepts:

```
I   to obtain the date in increment form Ñ i.e. relative to today
K   to return the id of the current  week, in the form yyyyww
    (according to ISO 2015 & 2711)
L   a logical value to tell if the year is a leap one or not
N   to obtain the month num (instead of name as in M) in the range 1Ð12
X   to obtain the weekday num (instead of name as in W) in the range 1Ð7
```

small xdate example

Here is a small example of the **xdate** function (look at the comments to see what the program really does):

```
+----------------------------------------------------------------------+
```

```
-- xdt0
-- Exercise a bit the XDATE functions
--

-- Get today's date
--
say xmisc.xdate('TODAY')

-- Get next monday's
--
say xmisc.xdate('NEXT','MONDAY')

-- convert 31 DEC 1994 in from European to Julian Format
--
say xmisc.xdate('E','31/12/94','J')

-- find out which weekday I was born
--
say xmisc.xdate('E','28/09/67','W')

-- find out which date will be in 1000 days
--
say xmisc.xdate('I',1000,'E')

-- find out how many days I have
--
say xmisc.xdate('TODAY','C') - xmisc.xdate('E','28/09/67','C')

-- find out when I'll have 20000 days
--
nn = xmisc.xdate('TODAY','C') - xmisc.xdate('E','28/09/67','C')
nn = 20000 - nn
say xmisc.xdate('I', nn , 'S')

say 'Today is:' xmisc.date('E')
say '         ' xmisc.date('W')

exit
+--------------------------------------------------------------------+
                                                              xdt0.nrx
```

Resources... [Download the source for the xdt0.nrx example](#)

NOTEs: And here is the output:

```
.......................................................................
rsl3pm1 (39) java xdt0
5 Feb 1995
94365
Thursday
01/11/97
9992
20220701
rsl3pm1 (40)
.......................................................................
                                                              xdt0.out
```

If you are wondering about all the possible output formats, here is a program for showing them:

```
+------------------------------------------------------------------------+
| -- xdt1.nrx
| -- exercise all XDATE formats                                          |
| --
|
| kind = 'O U S J B C D M W I K L N X'
| loop while kind <> ''
|    parse kind item kind
|    date = xmisc.xdate('TODAY',item)                                    |
|    say 'Format "'item'" is: 'date'.'                                   |
| end
| exit 0
+------------------------------------------------------------------------+
                                                                  xdt1.nrx
```

Resources... [Download the source for the xdt1.nrx example]

And this is what you will get if you run the program:

```
.......................................................................
rsl3pm1 (43) java xdt1
Format "O" is: 95/02/05.
Format "U" is: 02/05/95.
Format "S" is: 19950205.
Format "J" is: 95036.
Format "B" is: 728328.
Format "C" is: 34734.
Format "D" is: 36.
Format "M" is: February.
Format "W" is: Sunday.
Format "I" is: 0.
Format "K" is: 199505.
Format "L" is: 0.
Format "N" is: 2.
Format "X" is: 7.
rsl3pm1 (44)
.......................................................................
                                                              xdt1.out
```

The xsys.sleep() function.

It is often usefull to sleep() N seconds. The easyest way is to call the Thread.sleep() function:

```
-- just pause MILLISEC
Thread.sleep(MILLISEC)
```

where MILLISEC is the time you want to sleep (expressed in milliseconds).

Complex Data Structures

As we saw in the previous chapters, there is ONLY one native data type in NetRexx, and that is the **string**. NetRexx considers even the numbers as strings. Indeed, you can build yourself data types, the most useful one being the following:

```
 list (string)       (stem)
   =
 'ITEM1' ,     --->  value[ITEM1]
 'ITEM2' ,     --->  value[ITEM2]
 'ITEM3' ,     --->  value[ITEM3]
 (...)
 'ITEMN'       --->  value[ITEMN]
```

We have a string that holds a list of items, which are in their turn pointers for an array (or for many arrays) holding the data for that particular array.

A case study: printer accounting

We want to see how this data structure works in practice. An accounting program may be the best way. Supposing we are producing some accounting records whenever an user prints something on a printer, an accounting record is generated. The format of these records would be the following:

```
  date userid nodeid printerid no_of_pages
```

where:

```
  date.......: the date in the format YYMMDDhhmmss;
  userid.....: the user identifier;
  nodeid.....: the node he used to print from;
  printerid..: the name of the printer;
  no_of_pages: how many pages he printed.
```

Here is a small (usually this kind of files is MUCH bigger) example of such a file:

```
+------------------------------------------------------------------+
| 110195110233 mount slil308.cern.ch prt21 200                     |
| 110195120000 marchesi rsl3pm1.cern.ch prt56 82                   |
| 120195120340 marchesi rsl3pm1.cern.ch prt56 20                   |
| 120195123030 mount hpl3sn1.cern.ch prt11 1                       |
| 120195123400 clare shift31.cern.ch prt11 25                      |
+------------------------------------------------------------------+
                                                       printer.CARDS
```

The structure of our accounting program will be:

```
READ the accounting file
REDUCE the data
POST processing (if any)
DISPLAY results
```

First Version

In our first version for this program, we simply want to see how many pages a user has printed. The following program (called **pracc**) will do it. In the first portion of the code, we check for the input argument and read a file. We will not go into the details: what we do is simply get the lines of the accounting cards into the array **infid.line[]**.

```
/* prologue
  */
parse arg fid .
if fid = '-h' then
  do
    say 'usage pacc <fid>'
    exit 3
  end
if fid = '' then fid = 'printer.CARDS'
if \xfile.fexist(fid) then
  do
    say 'file "'fid'" does not exist.'
    exit 4
  end
infid = xfile(fid)
rc = infid.read()
if rc <> 0 then
  do
    say 'RC:' rc 'from READ.'
    exit 3
  end
```

pracc.nrx

Resources... [Download the source for the pracc.nrx example]

We are now ready to analyse our data ,i.e. the lines contained in the stem **CARDS**. As you can see, we loop over the accounting cards Ñ from the first over to the last one. We parse the information contained in a card **line 28**. We check if the user contained in the card is known. If not, we add the user to the 'known users' list (**user_list**), and just for double security, we initialise the number of pages printed to 0 (**line 32**). We then add the pages for this accounting card to the total for the user.

```
 /* Data Collection
   */
 user_list = ''
 pages_printed_by = 0
 loop i = 1 to infid.lines
   parse infid.line[i] date user node printer pages
   if user_list.wordpos(user) = 0 then
     do
       user_list = user_list user
     end
   pages_printed_by[user] = pages_printed_by[user] + pages
 end
```

pracc.nrx

Resources... [Download the source for the pracc.nrx example]

If we take the data we showed in the example **printer.CARDS**, this is what we get at the end of the code:

```
user_list = 'mount marchesi clare'

pages_printed_by.mount = 201
pages_printed_by.marchesi = 102
pages_printed_by.clare = 25
```

Now that the raw data is reduced in this format, we can do whatever we want over it: order by name of the user the **user_list**, order by number of printed pages, etc. We can even do nothing, such as here:

```
 /* post process
   */
```

Now we can display the 'reduced' data. This is just a loop over the users, and each time we will display the user and the pages printed.

```
 /* display
   */
 list = user_list
 loop for list.words()
   parse list item list
   say item.left(12,'.')':' pages_printed_by[item].right(7)
 end

 /* end
   */
```

```
exit 0
+----------------------------------------------------------------------+
                                                              pracc.nrx
```

Resources... [Download the source for the pracc.nrx example]

That is all. Here is what you get from the program itself:

```
..................................................................
rsl3pm1 (23) java pracc2
mount.......:      201
marchesi....:      102
clare.......:       25
rsl3pm1 (24)
..................................................................
                                                          pacc.out
```

A second version

Suppose that now your manager asks you to have the report not only for users, but ALSO for printers. The modifications are quite trivial you simply need to create a new list for the printers, and clone the logic you used so far:

```
+----------------------------------------------------------------------+
 /* prologue
  */
 (LIKE ABOVE)

 /* Data Collection
  */
 user_list = ''
 printer_list = ''
 pages_printed_by = 0
 loop i = 1 to infid.lines
   parse infid.line[i] date user node printer pages
   if user_list.wordpos(user) = 0 then
     do
       user_list = user_list user
     end
   if printer_list.wordpos(printer) = 0 then
     do
       printer_list = printer_list printer
     end
   pages_printed_by[user] = pages_printed_by[user] + pages
   pages_printed_by[printer] = pages_printed_by[printer] + pages
 end

 /* display
  */
 list = user_list
```

```
 loop for list.words()
   parse list item list
   say item.left(12,'.')':' pages_printed_by[item].right(7)
 end

 list = printer_list
 loop for list.words()
   parse list item list
   say item.left(12,'.')':' pages_printed_by[item].right(7)
 end
+-------------------------------------------------------------------------+
                                                                 pacc2.nrx
```

Resources... [Download the source for the pacc2.nrx example]

And this is what you will get on your screen:

```
.......................................................................
rsl3pm1 (58) pacc2
Users:
clare.......:      25
marchesi....:     102
mount.......:     201
Printers:
prt11.......:      26
prt21.......:     200
prt56.......:     102
rsl3pm1 (59)
.......................................................................
                                                                pacc2.out
```

Linked Lists

Another kind of data structure are **linked lists**. With NetRexx you can easily simulate a linked list data structure. I remind you of what a linked list is:

```
POINTER   --->  data.1    +-->  data.2
                          |
                info.1    |     info.2
                          |
                next.1 --+      next.2 --> NULL
```

Case study: a ps tree.

A good case study for the linked lists is a program for building a **ps** command tree. The UNIX **ps** command is used to show the current status of processes running on your machine. Each process has an **id** (the **processid**) and a **parent process** (also called **ppid**). The output of the **ps** command does not immediately show how a process is "linked" in terms of **parent process** to the previous ones. Here is a typical example:

```
.......................................................................
rsl3pm1 (226) ps -f
    USER    PID  PPID    C    STIME     TTY   TIME  CMD
marchesi   8161 13399    6 20:35:42   pts/4   0:00  rexx ps1
```

```
marchesi 10723 13026   3 20:35:42  pts/4  0:00 bsh bsh bsh
marchesi 13026  8161   1 20:35:42  pts/4  0:00 rexx ps1
marchesi 13399  9555   1   Jan 25  pts/4  0:06 -usr/local/bin/tcsh
marchesi 14564 10723   8 20:35:42  pts/4  0:00 ps -f
rsl3pm1 (227)
..................................................................
                                                                ps1.out
```

For our discussion, the important columns are the second and the third: the process that started all is the PID 13399; it generated PID 8161; which generated 13026; which executed 10723; which finally executed 14564 (and fortunately for us, nothing else other than printing what you see here). This is an "easy" case: if we had done a **'ps -ef'**, you would have got even more than 100 processes in no particular order. Our **pstree** wants to make order in this 'mess', and see how each process is linked by the parental relationship. The following code does the job. We skip all the 'unrelevant' portion of the program, since it does not add anything to our discussion. The first thing we do is execute the **ps** command with the proper options, depending on whether we want to see all the processes of the system **ps -ef** or just the ones belonging to us **ps -f**.

```
+-------------------------------------------------------------------+
| if all                                                            |
|   then rc = xexec('ps -ef' , 'ARRAY' , 'ABORT')                   |
|   else rc = xexec('ps -f'  , 'ARRAY' , 'ABORT')                   |
+-------------------------------------------------------------------+
```

We reorder copy the array **out[]** into the array **ps[]**. We skip the very first line of the **ps** command output.

```
+-------------------------------------------------------------------+
| j = 0                                                             |
| loop i = 2 to out[0]                                              |
|   j = j+1                                                         |
|   ps[j] = out[i]                                                  |
| end                                                               |
| ps[0] = out[0] -1                                                 |
+-------------------------------------------------------------------+
```

We now create two lists: the **pidl** is a string containing all the process_ids, while the **ppidl** is a string containing all the processes that are parents. The full information about the process is stored in the array **info[PID]** and the parent for each process is in **ppid[PID]**

```
+-------------------------------------------------------------------+
| pidl  = ''                                                        |
| ppidl = ''                                                        |
| do i = 1 to ps[0]                                                 |
|   parse ps[i] . pid ppid .                                        |
|   pidl = pidl pid                                                 |
|   ppidl = ppidl ppid                                              |
|   info[pid] = ps[i]   -- full process info                        |
|   ppid[pid] = ppid    -- parent                                   |
| end                                                               |
+-------------------------------------------------------------------+
```

We loop over the process list. We look for the processes that **are not parents of other processes**. Those processes are saved in **lastl**: they are the last in a chain of processes.

```
list  = pidl
lastl = ''
loop list.words()
  parse list item list
  if ppidl.wordpos(item) = 0 then
    do
      lastl = lastl item
    end
end
```

Now the most tricky part. We start from all the processes in **lastl** and go backwards. This is where we use the pseudo linked list. For each process in **lastl** we build the chain with the processes in order of generation.

```
list = lastl
loop list.words()
  parse list item list
  titem = ppid[item]
  chain[item] = titem item
  loop forever
    titem = ppid[titem]
    if pidl.wordpos(titem) = 0 then leave
    chain[item] = titem chain[item]
  end
end
```

Et voila': we have now only to print this chain.

```
list = lastl
loop list.words()
  parse list item list
  llist = chain[item]
  say ' '
  loop llist.words()
    parse llist item2 llist
    parse info[item2] owner p1 p2 . rest
    say p1.left(6) p2.left(6) '['owner']'left(,10) rest.left(50)
  end
end
```

A short output example:

```
.....................................................................
rsl3pm1 (231) pstree -a

(...)

1       0       [root]       Jan 19       -   9:28 /etc/init
2584    1       [marchesi]   Jan 19  hft/0  0:01 -tcsh
5668    2584    [marchesi]   Jan 19  hft/0  0:00 xinit
6698    5668    [marchesi]   Jan 19  hft/0  0:19 mwm
7220    6698    [marchesi]   Jan 19       -   0:16 aixterm
8765    7220    [marchesi]   Jan 19  pts/0  0:01 -tcsh
12329   8765    [marchesi]   Jan 24  pts/0  0:00 rlogin sgil301 -l f
13610   12329   [marchesi]   Jan 24  pts/0  0:00 rlogin sgil301 -l f

1       0       [root]       Jan 19       -   9:28 /etc/init
9555    1       [marchesi]   Jan 25  hft/0  1:16 aixterm
13399   9555    [marchesi]   Jan 25  pts/4  0:06 -usr/local/bin/tcsh
8174    13399   [marchesi] 20:59:12  pts/4  0:00 rexx pstree -a
```

```
13039   8174    [marchesi] 20:59:13  pts/4   0:00 rexx pstree -a
10736   13039   [marchesi] 20:59:13  pts/4   0:00 bsh bsh bsh
14577   10736   [marchesi] 20:59:13  pts/4   0:00 ps -ef

1       0       [root]      Jan 19       -   9:28 /etc/init
7746    1       [marchesi]  Jan 19  hft/0   6:15 aixterm
9030    7746    [marchesi]  Jan 19  pts/2   0:03 -usr/local/bin/tcsh
15292   9030    [marchesi] 20:14:31  pts/2   0:48 x rxuser.texinfo
rsl3pm1 (231)
.....................................................................
                                                            pstree.out
```

Additional information on Data Structures

You can find additional information about data structures in Java at those URLs:

http://www.geocities.com/SiliconValley/Way/7650/javadata.html

http://www.objectspace.com/jgl/

***** This section is:**

***** and will be available in next releases**

Summary

A resume' of the main concepts encountered in this chapter.

***** This section is:**

***** and will be available in next releases**

File: nr_26.html.

Last update was done on 18 May 1998 21:48:01(GMT +2).

The NetRexx Tutorial
- Advanced Algorithms

Advanced Algorithms

Introduction

Recursive Algorithms

A question that usually crops up in discussion groups about languages (notably **comp.lang.rexx**) is : 'Can I implement a recursive algorithm using REXX?'. The answer is: 'Yes'. You can easily make your NetRexx (or REXX) code re-entrant, and in this way implement any recursive algorithm. You perform this with a **method** clause.

The towers of Hanoi.

Text books usually provide as an example of recursive algorithm, the computation of a factorial (n!). This is probably not a good choice, as one can easily avoid recursion for this algorithm. I prefer to give the example of the 'Towers of Hanoi' [KRUSE, 1984, 273]. The game is well known: one must move disks from one 'tower' (1) to a third (3), without placing a larger disk on top of a smaller.

```
      (1)                   (2)                   (3)
       |                     |                     |
       #                     |                     |
      ###                    |                     |
     #####                   |                     |
    #######                  |                     |
   #########                 |                     |
  ###########                |                     |
 ---------------------------------------------------------

 Towers of Hanoi
```

Using recursion, the solution is extremely simple. Taking the algorithm from the cited source, we can write this small REXX program.

```
+-----------------------------------------------------------------+
| class hanoi
|   method move(n=rexx,a=rexx,b=rexx,c=rexx) public static        |
|     if n>0 then
|       do
|         move(n-1,a,c,b)                                          |
```

```
        say 'Move disk from' a 'to' b '.'
        move(n-1,c,b,a)
      end

   method main(args=String[]) public static
     n = args[0]
     move(n,1,2,3)
     exit 0

+----------------------------------------------------------------------+
                                                                hanoi.nrx
```

Resources... [Download the source for the hanoi.nrx example]

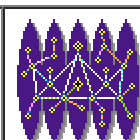
Believe it or not, this is the solution you get from the program. Note that it is also the best possible solution.

```
.....................................................................
rsl3pm1 (122) java hanoi1 4

  Move a disk from 1 to 2 .
  Move a disk from 1 to 3 .
  Move a disk from 2 to 3 .
  Move a disk from 1 to 2 .
  Move a disk from 3 to 1 .
  Move a disk from 3 to 2 .
  Move a disk from 1 to 2 .
  Move a disk from 1 to 3 .
  Move a disk from 2 to 3 .
  Move a disk from 2 to 1 .
  Move a disk from 3 to 1 .
  Move a disk from 2 to 3 .
  Move a disk from 1 to 2 .
  Move a disk from 1 to 3 .
  Move a disk from 2 to 3 .

rsl3pm1 (122)
.....................................................................
                                        result of the hanoi1 program
```

In the section about the **curses()** interface we will see how to get a better output for the solution of the game.

Recursive sort algorithms

```
+----------------------------------------------------------------------+
| -- method......: partition
| -- purpose.....:
| --
|  method partition(l=rexx[],low=rexx,high=rexx) public static returns
|    swap(l,low,(low+high)%2)           -- swap pivot in 1st location
|    pivot = l[low]
|    lastsmall = low
|    loop i = low+1 to high
|      if l[i] < pivot then
```



```
        do
          lastsmall = lastsmall + 1
          swap(l,lastsmall,i)         -- move large to right, small to
        end
    end
    swap(l,low,lastsmall)             -- put pivot into its proper pos
    pivotlocation = lastsmall
    return pivotlocation

+-----------------------------------------------------------------------+
                                                qsn.nrx(Method:partition)
```

Resources... [Download the complete source for the qsn.nrx library]

Removing recursion

```
+-----------------------------------------------------------------------+
 -- method......: sort_qsnr
 -- purpose.....: sort the list using QuickSort Nonrecursive
 --
   method sort_qsnr(l=rexx[]) public static

    maxstack = 20                        -- up to 1,000,000 items
    lowstack = rexx[maxstack]            -- arrays used for the st
    highstack = rexx[maxstack]

    low = 0                              -- list bounds
    high = l.length - 1

    nstack = 0

    loop until nstack = 0
      if nstack > 0 then
        do
          low = lowstack[nstack]         -- pop the stack
          high = highstack[nstack]
          nstack = nstack - 1
        end

      loop while low < high
        pivotloc = partition(l,low,high)

        -- push larger list into stack, and do the smaller
        --
        if (pivotloc - low) < (high - pivotloc) then
          do
            -- stack right sublist and do left
            --
```

```
            if nstack > maxstack then overflow()
            nstack = nstack + 1
            lowstack[nstack] = pivotloc + 1
            highstack[nstack] = high
            high = pivotloc - 1
          end
        else
          do
            -- stack left sublist and do right
            --
            if nstack > maxstack then overflow()
            nstack = nstack + 1
            lowstack[nstack] = low
            highstack[nstack] = pivotloc - 1
            low = pivotloc + 1
          end
      end
  end
```

qsn.nrx(Method:sort_qsnr)

Resources... [Download the complete source for the qsn.nrx library]

```
 -- method......: main
 -- purpose.....: just test the main functions simply running
 --               "java qsn"
 --
  method main(args=String[]) public static
    args = args

    l = rexx[100]

    build_list(l)
    display_list(l)
    sort_qsnr(l)
    display_list(l)

    exit 0
```

qsn.nrx(Method:main)

Resources... [Download the complete source for the qsn.nrx library]

***** This section is:**

***** and will be available in next releases**

Summary

Here is the usual resume' of some of the concepts we have encountered in this chapter:

***** This section is:**

***** and will be available in next releases**

File: nr_27.html.

Last update was done on 18 May 1998 21:48:02(GMT +2).

The NetRexx Tutorial
- NetRexx for REXXers

NetRexx for REXXers

Introduction

In this chapter we analyse the main differences between the Classical REXX and the NetRexx languages.

NetRexx is NOT REXX, and this you will see from all the following sections.

***** This section is:**

***** and will be available in next releases**

NetRexx is compiled, and not interpreted.

One of the biggest differences that REXX (or ooREXX) users will find in NetRexx is the fact that now you need to compile your program.

The usual approach:

```
LOOP till it works
  edit program
  run program
END
```

has now an extra step:

```
LOOP till it works
  edit program
  compile program
  run program
END
```

Not only, but since the object is a Java class, you also must call the program using java.

Differences.

This sections covers all the instructions that are changed, between REXX and NetRexx.

Continuation Character.

The continuation character is different in NetRexx. The reason is that the "old" REXX one (the ",") could be difficult to read if (as usually happens) you where calling a function or a procedure.

```
REXX      result = myfunction( arg1 , ,
                               arg2 )

NetRexx   result = myfunction( arg1 , -
                               arg2 )
```

Entering Arguments.

In REXX we use the instruction **parse pull**, or the simple **pull** to get arguments from the keyboard.

```
REXX      say 'Enter Name'
           parse pull upper name .

NetRexx   say 'Enter Name'
           parse ask.upper() name .
```

STEMs and ARRAYS.

The STEMs are present in NetRexx, but they're called with a different name. They're are called ARRAYs and the compound variable separator is not the "." but the "[ ]" characters. Like STEMS, ARRAY should be initialised to a value.

```
REXX      list. = "
           list.0 = 2
           list.1 = 'Test'
           list.2 = 'Toast'

NetRexx   list   = "
           list[0] = 2
           list[1] = 'Test'
           list[2] = 'Toast'
```

Dealing with multidimensional arrays use the "," character to separate the dimensions; in REXX you still were using the "." .

```
REXX      list.1.2 = 4
           list.i.j = 6

NetRexx   list[1,2] = 4
           list[i,j] = 2
```

function calls

Any internal NetRexx function is called in an Object Oriented fashion.

```
REXX      n = abs(n)
```

```
NetRexx   n = n.abs()
REXX      sn = right(s,2,'0')
NetRexx   sn = s.right(2,'0')
```

ALL the functions are effected. **NOTE:** This is clearly a major change. I had a bit of hard time to get used to it, but after an initial rejection, I find it more "natural".

Look at this example:

```
REXX      bin = x2b(c2x(s))
NetRexx   bin = s.c2x.x2b()
```

From the second writing it comes very much more evident that what I'm trying to do is a:

```
    c2x.x2b
     =====
      c2b
```

conversion.

xrange()

There is NO **xrange** instruction in NetRexx.

```
REXX      str = xrange('00'X,'1F'X)
NetRexx   str = '\x00'.sequence('\x1F')
```

xrange() is implemented in **xstring**.

HEX characters.

You use a different method to enter HEX quantities in NetRexx.

```
REXX      crlf = '0D0A'X
NetRexx   crlf = '\x0D\x0A'
```

Missing instructions.

find() and index()

The **find()** and **index()** functions have always been available in the VM/CMS implementation of REXX. Indeed, they've never been in the "official" REXX.

```
REXX:     find(list,item)
```

```
                |
                V
NetRexx:   list.wordpos(item)


REXX:      index(string,item)
                |
                V
NetRexx:   list.pos(item)
```

Of course you can write your own **find()** and **index()** that just do **pos()** and **wordpos()**.

Additions

upper() and lower()

The **upper()** and **lower()** functions are native in NetRexx. They were not available in native REXX.

```
/* */                     REXX

str = str.lower()        NetRexx
str = str.upper()
```

Associative Arrays

Indexed Strings are used to set up "Associative Arrays" in which the sunscript is not necessarily numeric.

In "classic" REXX you would code:

```
+--------------------------------------------------------------------------+
| authorizelist = 'BOB JENNY PENNY'
| authorize.jenny = 'list cat'
| authorize.bob   = 'list cat write'
| authorize.penny = 'list'
| list = authorizelist
| do while list <> ''
|   parse var list id list
|   say id 'can do "'authorize.id'".'
| end
| exit
+--------------------------------------------------------------------------+
                                                                    asar.rex
```

```
+--------------------------------------------------------------------------+
| authorize = ''
```

```
 authorize['jenny'] = 'list cat'
 authorize['bob'] = 'list cat write'
 authorize['penny'] = 'list'
 loop id over authorize
   say id 'can do "'authorize[id]'".'
 end
 exit
```

asar.nrx

Resources... [Download the source for the asar.nrx example](#)

Program structure

This is probably the biggest difference between REXX and NetRexx. Subroutines and procedures like you knew them in REXX disappear, and the concept of method replaces them.

The following are some small examples.

Argument passing

```
 /* compute the mean value of two numbers
   */
 parse arg n1 n2 .
 say 'The mean value of' n1 'and' n2 'is:' mean(n1,n2)'.'
 exit

 mean: procedure;
   parse arg i1 , i2
   m = (i1+i2)/2
   return m
```

tnr1.rex

```
 -- tnr1.nrx
 --    Show the usage of a function
 class tnr1

```

```
   method mean(i1=Rexx,i2=Rexx) public static
     out = (i1+i2)/2
     return out

   method main(args=String[]) public static
     arg = Rexx(args)
     parse arg n1 n2 .
     say 'mean of' n1 'and' n2 'is:' mean(n1,n2)'.'
     exit 0
+----------------------------------------------------------------------+
                                                               tnr1.nrx
```

Resources... [Download the source for the tnr1.nrx example]

Exposing variables

```
+----------------------------------------------------------------------+
 /* tnr2.rex
  */
 avar1 = 'MAIN'
 avar2 = 'MAIN'
 call sub1
 say avar1
 say avar2
 exit

 sub1: procedure expose avar1
   avar1 = 'SUB1'
   avar2 = 'SUB1'
   say avar1
   say avar2
   return
+----------------------------------------------------------------------+
                                                               tnr2.rex
```

```
+----------------------------------------------------------------------+
 class tnr2
   properties public static
     avar1

   method sub1() public static
```

```
|    avar1 = 'SUB1'     -- will be changed                              |
|    avar2 = 'SUB1'     -- will NOT be changed                          |
|    say avar1                                                          |
|    say avar2                                                          |
|                                                                       |
|  method main(args=String[]) public static                             |
|    args = args                                                        |
|    avar1 = 'MAIN'                                                     |
|    avar2 = 'MAIN'                                                     |
|    sub1()                                                             |
|    say avar1                                                          |
|    say avar2                                                          |
|    exit 0                                                             |
+-----------------------------------------------------------------------+
                                                                 tnr2.nrx
```

Resources... [Download the source for the tnr2.nrx example]

***** This section is:**

***** and will be available in next releases**

This really got me!

In this section I collect all "nasty" problems that I found in NetRexx, and which probably were due to my REXX background. I hope that this collection will avoid you loosing the time I did lost to find out why a particular algorithm was not working.

Variable and array/stem with the same name.

In **REXX** you can have variables that share the same name of a STEM. You can happily write:

```
line = line.i
```

and line (a variable), will get the value of the stem variable line.i.

```
+-----------------------------------------------------------------------+
| line.1 = 'Test line'                                                  |
| line.2 = 'another one'                                                |
| line.3 = 'last one'                                                   |
| do i = 1 to 3                                                         |
```

```
|    line = line.i
|    say line
|  end
|  exit
+--------------------------------------------------------------------+
                                                              tgm1.rex
```

In NetRexx such approach will not work. In the following program, infact, the statement:

```
line = line[i]
```

will just initialise the whole array line[] to line[1]. SO ALL THE ARRAY INFORMATION WILL BE OVERWRITTEN.

```
+--------------------------------------------------------------------+
|  line = Rexx('')
|  line[1] = 'test line'
|  line[2] = 'another one'
|  line[3] = 'last one'
|  loop i = 1 to 3
|    line = line[i]
|    say line
|  end
+--------------------------------------------------------------------+
                                                              tgm1.nrx
```

Resources... [Download the source for the tgm1.nrx example]

In REXX, you would have achieved the same result writing:

```
line. = line.1
```

Chapter FAQ.

Would it be possible to make a REXX to NetRexx translator?

Yes, as you could see a lot of the differences in the syntax could be made in an automatic way. It is simple to translate an instruction like:

```
s1 = left(s,3)
```

to:

```
s1 = s.left(3)
```

I plan to write some code that will do a 'first step' translation. So far I know nobody who did it.

Summary

```
*** This section is:
```


```
*** and will be available in next releases
```

File: nr_28.html.

Last update was done on 18 May 1998 21:48:03(GMT +2).

The NetRexx Tutorial - Tools

Tools

Introduction

General Tools

Get your environment

Whenever you have a problem, or you suspect a bug in NetRexx, you should always report it to the NetRexx mailing list.

To give people a better idea of your environment, you might also provide the information that this small program provides, so to help the readers to guess where the problem is.

The real important instruction is:

```
p2 = Rexx System.getProperty(item)
```

So, for example:

```
myos = Rexx System.getProperty('os.name')
       -- will display your OS

myid = Rexx System.getProperty('user.name')
       -- will display your USERID
```

```
/* Program   :  nrenv
 * Subsystem :  nrtools
 * Author    :  Pierantonio Marchesini.
 * Created   :  7 Feb 1997.
 * Info      :  Get the NetRexx environment
 * Copyright :  (c) P.A.Marchesini / ETHZ 1997.
 *
 * Id     Info
 * ------ --------------------------------------------------------
 * v1r000 First release.
```

```
  * v1r000 Latest release
  *
  */
 pro_ver = 'v1r000';

 parse source env mc myname'.' .
 say 'Welcome to "'myname'". Version ' pro_ver'.'
 say
 say 'NetRexx........:' version
 say 'Environment....:' env

 --
 -- set the properties
 --

 prop = 'java.version java.vendor'                          -
        'java.vendor.url java.class.version'                -
        'java.class.path os.name os.version file.separator' -
        'path.separator user.name user.home user.dir'       -
        'awt.toolkit'

 -- find out which string is longer, in order
 -- to have a cleaner output
 --
 list = prop
 max_len = 0
 loop while list <> ''
   parse list item list
   if item.length() > max_len
     then max_len = item.length()
 end

 -- loop over properties.
 -- display the property and the value
 --
 say
 loop while prop<>"
   parse prop item prop
   p1 = '<'item'>'
   p1 = p1.right(max_len+2)
   p2 = Rexx System.getProperty(item)

   if item.pos('separator') <> 0    -- if it's a separator,
     then                            -- we print also the HEX value
       do
         p2 = "'"p2.c2x()"'X :" p2'.'
       end

   if item = 'java.class.path' then  -- if it's a path, then split
     do                              -- the different directories
```

```
      pathl = p2
      loop while pathl <> ''
        parse pathl path';'pathl
        say p1 '=' path
        p1 = ''.right(20)
      end
      iterate
    end

  say p1 '=' p2
end
say
exit 0
+--------------------------------------------------------------------------+
                                                                  nrenv.nrx
```

Resources... [Download the source for the nrenv.nrx example]

Depending on your Operating system, you can redirect the output of the program to a file, like:

java nrenv > nrenv.out

This is what I get if I run the command on my system.

```
..........................................................
Welcome to "nrenv". Version  v1r000.

NetRexx........: NetRexx 1.00 24 May 1997
Environment....: Java

      <java.version> = 1.1.1
       <java.vendor> = Sun Microsystems Inc.
   <java.vendor.url> = http://www.sun.com/
<java.class.version> = 45.3
   <java.class.path> = .
                     = C:\java\lib\NetRexxC.zip
                     = C:\java\NetRexx\examples
                     = C:\java\lib
                     = c:\java\bin\..\classes
                     = c:\java\bin\..\lib\classes.zip
           <os.name> = Windows NT
        <os.version> = 4.0
    <file.separator> = '5C'X : \.
    <path.separator> = '3B'X : ;.
         <user.name> = Administrator
         <user.home> = C:\
          <user.dir> = c:\Java\NetRexx\examples
       <awt.toolkit> = sun.awt.windows.WToolkit
....................................................
```

Building the Tutorial's libraries

In order to get the libraries provided with the tutorial correctly installed, you have to follow the procedure described in this section.

Getting the code.

The code is freely available at:

http://wwwinfo.cern.ch/news/netrexx/library/alllib.tar.gz

or, at the URL:

http://wwwinfo.cern.ch/news/netrexx/library/

as individial files. Download all the files inside a single directory, using your preferred

Installing the libraries.

You have to compile "by hand" two programs: **xsys.nrx** and **xbuild.nrx**, in EXACTLY this order. Then you just use the newly created **xbuild.class** to build all the other libraries.

So you'll type:

```
>java COM.ibm.netrexx.process.NetRexxC xsys.nrx
>java COM.ibm.netrexx.process.NetRexxC xbuild.nrx
>java xbuild
```

If you do not get any nasty error messages, you're done, and you can use the libraries.

Some notes on xbuild

The most important part of the **xbuild.nrx** program is the following:

```
+------------------------------------------------------------------------+
| -- method......: main                                                  |
| -- purpose.....: just run typing "java xbuild"                         |
  --
|
|   method main(args=String[]) public static                             |
|     arg = Rexx(args)
|
|
|     -- Need help?
|
|     --
|
|     if arg = '-h' | arg = '--help' then
|
|       do
|
|         help()
|
|         exit 1
|
|       end
|
|
```

```
    version()
    -- OK, let's do it
    --
    todo = 'xmath.nrx xstring.nrx xsys.nrx xsock.nrx' -
           'xshell.nrx xurl.nrx'

    say 'Checking libraries.'
    list = todo
    loop while list <> ''
      parse list item list
      if state(item) = 0 then
        do
          say 'File "'item'" does not exist. Aborting.'
          exit 2
        end
      say 'Library "'item'" present.'
    end
    say

    say 'Building now the libraries.'
    list = todo
    loop while list <> ''
      parse list item list
      say 'Building now "'item'".'
      cmd = 'java COM.ibm.netrexx.process.NetRexxC' item
      c = xexec(cmd,'SCREEN','IGNORE')
      rc = c.rc
      if rc = 0
        then say 'Compilation was OK.'
        else say 'WARNING: rc:' rc 'from "'cmd'".'
    end
    exit 0
+--------------------------------------------------------------------+
                                                xbuild.nrx(Method:main)
```

Resources... [Download the complete source for the xbuild.nrx library]

***** This section is:**

***** and will be available in next releases**

File: nr_29.html.

Last update was done on 18 May 1998 21:48:04(GMT +2).

The NetRexx Tutorial
- The xclasses JAR library

The xclasses JAR library

Introduction

```
                   XCLASSES PACKAGE DOCUMENTATION
                   (c) P.A.Marchesini, 1998

***
***   xarray
***

   SUMMARY

     Handles array operations, and, mainly byte array
     conversions. It's a collection of static methods.
     NOTE: ARRAY needs to be defined as:
              an_array      = rexx[NNN]
              another_array = rexx[NNN]
              bytearray     = byte[MMM]

   METHODS

     xarray.init(ARRAY,VALUE)
       initializes a Rexx array ARRAY with value VALUE.
       Example
         xarray.init(an_array,'test test')

     xarray.copy(ARRAY1,ARRAY2)
       copyes a Rexx array ARRAY1 into array ARRAY2.
       Example
         xarray.copy(an_array,another_array)

     xarray.dump(ARRAY,ARRAYNAME)
       dumps the entries of ARRAY on the screen; duplicate
       lines are skipped.
       Example
         xarray.dump(an_array,'an_array')

     xarray.ba2x(BYTEARRAY,START,LENGTH)
       convert byte array BYTEARRAY from byte to HEX string
       starting at byte START for LENGTH bytes.

     xarray.ba2c(BYTEARRAY,START,LENGTH)
       as above, but converting to CHAR.

     xarray.ba2d(BYTEARRAY,START,LENGTH)
       as above, but converting to DECIMAL.

     loc = xarray.bagrepx(BYTEARRAY,SEARCH,START)
       will search in byte array BYTEARRAY the HEX string
       SEARCH, starting from START.
       Example:
         ptr = xarray.bagrepx(buf,'0D0F',0)

     xarray.bahexdump(BYTEARRAY,START,END)
       will dump HEX the contents of bytearray BYTEARRAY
       Example:
         fid = xfile('xarray.class')
         rc = fid.readbuf()
         xarray.bahexdump(fid.buffer,0,100)
```

```
 ***
 ***  xcmdline
 ***

     SUMMARY

      use this class to parse the command line options (which, in the
      UNIX convention, are entered with a '-' sign).

     METHODS

        cl = xcmdline(LINE,CONTROL)
             where LINE    : line entered by the user
                   CONTROL : defines the control sequence to parse the options
                             the format is
                               FLAG/[FLA|VAR]/VARIABLE_NAME/DEFAULT_VALUE

      EXAMPLE

         cl = xcmdline(rexx(args),'t/FLA/TRACE/0'           -
                                  'r/FLA/REPLACE/0'         -
                                  'o/VAR/OUTFID/test.out')

         If the user types:
           mytest test -ro pippo.txt
             ->  say cl.arguments()       = test
                 say cl.option('TRACE')   = 1
                 say cl.option('REPLACE') = 0
                 say cl.option('OUTFID')  = pippo.txt

      NOTES

        - next release will have a syntax like PERL getopt() available too

 ***
 ***  xdir
 ***

     SUMMARY
       Handles all operations on a directory, listing, comparing
       etc.

     METHODS
       xdir(DIRECTORY)
       xdir()
         constructors. Default directory is the
         current directory (".")

       str_ls(DIRECTORY) -
         issue a "ls" command. Output returned in a REXX
         string.

     PROPERTIES
       rc        - return code of last valid operation
       options

     EXAMPLES

        say xdir.str_ls("/java")

     NOTES

 ***
 ***  xexec
 ***

     SUMMARY

       Use this class to run a system command.

     METHODS

       cmd = xexec(COMMAND,OUTPUT,ONERROR)

       where:
          COMMAND  : is a valid command on the system you are
                     running on (e.g. "ls","cp","copy", etc.)
          OUTPUT   : can be any combination of:
                        SCREEN : the output will go on STDOUT
                        VAR    : the output will go on a variable
```

```
                     ARRAY  : the output will go on an array
                  or
                     NULL   : forget about output
         ONERROR   : is one of:
                     IGNORE    : a return code <> 0 is ignored
                     WARNING   : print a warning message if rc <> 0
                     ABORT     : abandon ship if rc <> 0

     PROPERTIES

         lines     : lines of output
         line      : array of output lines; line[0]=no.of out lines
         out       : string of output (when VAR is active)
         rc        : return code

     EXAMPLES

         test = xexec('cp test toast','NULL','ABORT')

         test = xexec('pwd','VAR','ABORT')
         say 'The path is "'test.out'".'

         test = xexec('ls -l','ARRAY','WARNING')
         loop i = 1 to test.line[0]
           say '>>>' test.line[i]
         end

      NOTES

        -  The examples are valid on a UNIX platform
        -  The examples are provided just as EXAMPLES
           there are infact better ways to do 'ls','pwd'
           in NetRexx itself

 ***
 ***  xfile
 ***

     SUMMARY

     METHODS

     PROPERTIES

     EXAMPLES

     NOTES

 ***
 ***  xftp
 ***

     SUMMARY

     METHODS

     PROPERTIES

     EXAMPLES

     NOTES

 ***
 ***  xmath
 ***

    SUMMARY

      Mainly provide conversion tools

    METHODS

      str = xmath.n2cu(NNN)
        converts numeric quantity NNN to computer units
        Example:
          say xmath.n2cu(2048)    -> 2K

      str = xmath.s2h(SEC)
         converts SEC to HH:MM:SS
         Example:
           say xmath.s2h(3661)    -> 1:01:01
```

```
      str = xmath.dotify(NNN)
         puts the "," in a big number, for easy reading
         Example:
           say xmath.dotify(100203) -> 100,203

      str = xmath.hexop(HEXOP)
         will execute a simple hex operation
         Example:
            say xmath.hexop('1A + 10')  -> 2A

      str = xmath.binop(HEXOP)
         executes a simple bin operation.
         Example:
            say xmath.binop('10 + 11')  -> 101

      n = xmath.random(MAX)
         returns an random integer with maximum value
         not greater than MAX.
         Example:
             say xmath.random(25) -> 12 (MAYBE)

      n = xmath.gcd(m,n)
         returns the Greatest Common Divisor of M and N.

      rc = xmath.gauss(N,A[,],Y[],C[])
         upon return code RC=0 it will find using the
         Gauss Method the solution C[] for the array A[,]
         and vector Y[]

 ***
 ***  xsys
 ***

    SUMMARY

      This is just a collection of methods for "system" related
      information.

    METHODS

       str = xsys.userid()
             will return your current userid.
             Example:
             say 'I am running on user "'xsys.userid()'".'

       str = xsys.nodeid()
             will return the name of the node you are running
             on.
             Example:
             say 'I am running on system "'xsys.nodeid()'".'

       str = xsys.time()
       str = xsys.time(FMT)
             will return the current time.
             FMT is the same as on Classical REXX
             Example:
             say 'Now is:' xsys.time()'.'

       str = xsys.date()
       str = xsys.date(FMT)
             will return the current date.
             FMT is the same as on Classical REXX
             Example:
             say 'Today is' xsys.date()'.'

       str = xsys.xdate(IFMT,DATE,OFMT)
             will perform date conversion.
             Example:
             say xsys.xdate('E','12/01/97'.'J')

       xsys.die(RC,MESSAGE)
             program will abort with RC return code, displaying
             MESSAGE on STDOUT;
             Example:
             xsys.die(320,'Program aborted.')

       xsys.sleep(SEC)
             program will sleep for SEC seconds

 ***
```

```
***  xtimer
***

    SUMMARY
      Use xtimer class to build timers inside your programs.

    METHODS

      xtimer()      - constructor
                      The starting time is set to 0.000 sec

      reset()       - the timer is reset to 0.000 sec

      elapsed()     - Returns the elapsed time since the
                      last reset() (or object creation)
                      Format is SSSSS.MMM
                      (seconds.milliseconds)

    PROPERTIES

    EXAMPLES

       atimer = xtimer()
       -- some job here
       --
       say 'Took.....:' atimer.elapsed'(sec).'
       atimer.reset()
       -- some other job here
       --
       say 'Took.....:' atimer.elapsed'(sec).'

    NOTES
```

File: nr_30.html.

Last update was done on 18 May 1998 21:48:05(GMT +2).

The NetRexx Tutorial
- Miscellaneous

Miscellaneous

Introduction

In this chapter I collect all the information that could not fit in the previous chapters.

You might find usefull references to additional documents as well.

Packages and JAR files

Packages

In Java terminology the word "package" means **a collection of individual .class files contained in a directory**. A package is then a directory and a library, and you use it to group more than one class together.

You then perform the grouping of the source NetRexx files in a directory. And now comes the most important point: the directory **name** MUST match the **package** name.

Real example

In this subsection I'll show how I built the first time my **xclasses.jar** file.

```
# 1.00 create a directory called "xclasses"
#      and go into it
$ mkdir xclasses
$ cd xclasses

# 2.00 edit the classes that make the package
#      ADD a "package xclasses" line at beginning
#      then compile it with nrc
$ edit *.nrx
$ nrc *.nrx

# 3.00 build the JAR file
#      FROM THE DIRECTORY ABOVE!
$ cd ..
$ jar -cvf /java/lib/xclasses.jar xclasses/*.class

# 4.00 change the CLASSPATH and add
#      C:\java\lib\xclasses.jar
$ export CLASSPATH=$CLASSPATH";C:\java\lib\xclasses.jar"

# 5.00 test it
#
$ cd /spool/test
$ cat t1.nrx
```

```
import xclasses.
rc = xexec('ls -l')
$ nrc t1
$ java t1
```

Pipes for NetRexx and Java

Ed Tomlinson has ported the VM/CMS Pipes functionality on NetRexx (and Java). You can find all the information at the URL:

http://www.cam.org/~tomlins/njpipes.html

Additional Informations available on the WEB.

Comments about NetRexx

An article about NetRexx has appeared on the Windows Magazine (Windows Magazine, July 1997, page 156). You can find a copy on:

http://www.winmag.com/library/1997/0701/winla114.htm

REXX FAQ.

For the REXX FAQ, you should consult the page:

http://www.mindspring.com/~dave_martin/RexxFAQ.html

or (in its non-frame version)

http://www.mindspring.com/~dave_martin/FAQNoFrames.html

Regular expressions.

Although I'm not a REGEX fan (since all you can do in a Regular Expression you can do with native NetRexx functions), there are a lot of colleagues who are really REGEX lovers.

So, for pattern matching issues, look at:

http://www.win.net/~stevesoft/pat
http://www.java.no/javaBIN/docs/api/sun.misc.Regexp.html
http://www.java.no/javaBIN/docs/api/sun.misc.RegexpPool.html

A good set of packages is also available at the Original Reusable Objects, ORO Site:

http://www.oroinc.com/downloads/index.html

You will find a Java regular expression package (OROMatcher), a Easy to use Perl5 regular expressions in Java package (PerlTools) and a AWK regular expressions for Java (AwkTools).

Summary

File: nr_31.html.

Last update was done on 18 May 1998 21:48:06(GMT +2).

The NetRexx Tutorial
- Appendix A: Bibliography

Appendix A: Bibliography

Non-IBM Books and Manuals on REXX

This is a list of titles you can find about classical Rexx.

1. **[OHARA GOMBERG, 1985]** Modern Programming Using REXX -- Robert P. O'Hara and David R. Gomberg In English: ISBN 0-13-597311-2 Prentice-Hall, 1985 ISBN 0-13-579329-5 (Second edition), 1988 (From REXXPress, 7 Gateview Court, SF CA 94116-1941, USA)
2. **[COWLISHAW, 1985]**The REXX Language -- M. F. Cowlishaw In English: ISBN 0-13-780735-X Prentice-Hall, 1985 ISBN 0-13-780651-5 (Second edition), 1990 In German: ISBN 3-446-15195-8 Carl Hanser Verlag, 1988 ISBN 0-13-780784-8 P-H International, 1988 In Japanese: ISBN 4-7649-0136-6 Kindai-kagaku-sha, 1988
3. **[MSG, 1985]**Personal REXX User's Guide (PC-DOS and OS/2 REXX) version 2.0 Mansfield Software Group, Inc., 1985-1990
4. **[HAWES, 1987]**ARexx User's Reference Manual (The REXX Language for the Amiga) William S. Hawes, 1987
5. **[TWG, 1990]**uniREXX Reference Manual (REXX for a variety of Unix systems) The Workstation Group, 1990
6. **[SLAC, 1990]** Proceedings of the REXX Symposium for Developers and Users SLAC Report-368, 235pp, June 11, 1990
7. **[GARGIULO, 1990]** REXX In the TSO Environment -- Gabriel F. Gargiulo ISBN 0-89435-354-3, QED Information Systems Inc., Order #CC3543; 320pp, 1990 Revised edition: ISBN 0-89435-418-3, QED Information Systems Inc., 471pp, 1993
8. **[RUDD, 1990]** Practical Usage of REXX -- Anthony S. Rudd ISBN 0-13-682790-X, Ellis Horwood (Simon & Schuster), 1990
9. **[QUERCUS, 1991]** Personal REXX User's Guide (PC-DOS and OS/2 REXX) version 3.0 Quercus Systems, 268pp, 1991
10. **[PREXX, 1991]** Portable/REXX for MS/DOS (Guide, Reference manual, Examples Reference, Reference Summary, and Learning to Program with Portable/REXX)
11. **[WATTS, 1991]** REXX/Windows (Product Guide and Reference) Keith Watts, Kilowatt Software, 1991
12. **[SLAC, 1991]** Proceedings of the REXX Symposium for Developers and Users SLAC Report-379, 244pp, May 8-9, 1991
13. **[ZAMARA, 1991]** Using ARexx on the Amiga -- Chris Zamara and Nick Sullivan ISBN 1-55755-114-6, 424pp+diskette, Abacus, 1991
14. **[GOLDBERG, 1991]** The REXX Handbook -- Edited by Gabe Goldberg and Phil Smith III ISBN 0-07-023682-8, 672pp, McGraw Hill, 1991
15. **[GIGUERE, 1991]** Amiga Programmer's Guide to ARexx -- Eric Giguere Commodore-Amiga, Inc., 1991

16. **[DANEY, 1991]** Programming in REXX -- Charles Daney ISBN 0-07-015305-1, 300pp, McGraw Hill, 1992
17. **[SLAC, 1992]** Proceedings of the REXX Symposium for Developers and Users SLAC Report-401, 401pp, May 3-5, 1992
18. **[CALLAWAY, 1992]** The ARexx Cookbook -- Merrill Callaway ISBN 0-9632773-0-8, 221pp, Whitestone, 1992 (Companion diskette: ISBN 0-9632773-1-6)
19. **[KIESEL, 1993]** REXX--Advanced Techniques for Programmers -- Peter C. Kiesel ISBN 0-07-034600-3, 239pp, McGraw Hill, 1993
20. **[BURNARD, 1993]** Denise Burnard, IBM AIX REXX/6000, Reference 1, IBM, 1993
21. **[NIRMAN, 1993]** REXX Tools and Techniques -- Barry K. Nirmal ISBN 0-89435-417-5, 264pp, QED, 1993
22. **[GORAN, 1994]** REXX Reference Summary Handbook (OS/2) -- Dick Goran ISBN 0-9639854-0-X C F S Nevada, Inc, 102pp, 1993. ISBN 0-9639854-1-8 (second edition), 148pp, 1994.
23. **[HALLETT, 1993]** OS/2 2.1 REXX Handbook: Basics, Applications, and Tips -- Hallett German ISBN 0442-01734-0, 459pp, Van Nostrand Reinhold, 1993
24. **[SLAC, 1993]** Proceedings of the REXX Symposium for Developers and Users SLAC Report-422, 247pp, May 18-20, 1993
25. **[GARGIULO, 1994]** Mastering OS/2 REXX -- Gabriel F. Gargiulo ISBN 0-471-51901-4, 417pp, Wiley-QED, 1994
26. **[RUDD, 1994]** Application Development Using OS/2 REXX -- Anthony S. Rudd ISBN 0-471-60691-X, 416pp, Wiley-QED, 1994
27. **[SCHINDLER, 1994]** Teach Yourself REXX in 21 Days -- William F. Schindler & Esther Schindler ISBN 0-672-30529-1, 527pp, SAMS, 1994
28. **[RICHARDSON, 1993]** Writing OS/2 REXX Programs -- Richardson ISBN 0-07052-372-X, McGraw-Hill, 1993
29. **[RICHARDSON, 1994]** Writing VX-Rexx for Programs (with disk) -- Richardson ISBN 0-07911-911-5, McGraw-Hill, 1994
30. **[KYNNING, 1985]** REXX Procedursprak--hur du programmerar din PC med OS/2 -- Bengt Kynning ISBN 91-44-48541-7, 300pp, Studentlitteratur (Sweden), 1994
31. **[GERMAN, 1992]** Command Language Cookbook -- Hallett German ISBN 0-442-00801-5, 352pp, Van Nostrand Reinhold, 1992
32. **[QUERCUS, 1992]** Personal REXX User's Guide, Version 3.0 -- OS/2 Supplement Quercus Systems, 94pp, 1992
33. **[HOCKWARE, 1993]** VisPro/REXX (Visual programming with REXX) Hockware Inc, 196pp, 1993
34. **[KEES, 1993]** REXX in der Praxis -- Peter Kees ISBN 3-486-22666-5, 279pp, Oldenbourg, 1993
35. **[WATCOM, 1993]** VX-Rexx for OS/2 (Programmer's Guide and Reference) 2.0 ISBN 1-55094-074-0 Watcom International Corp.,724pp, 1993

IBM Books and Manuals

These are the books that you can obtain directly from IBM. The first number is the IBM BOOK number, which you should use when ordering the book.

Cross-system books

```
ZB35-5100  The REXX Language, 2nd Ed.
           -- Cowlishaw
SC26-4358  SAA CPI: Procedures Language Reference
SC24-5549  SAA CPI: REXX Level 2 Reference
G511-1430  IBM REXX Compiler and Library/370:
           -- Introducing the Next Step in REXX
```

```
               (CMS, MVS)
   SH19-8160  REXX/370 (Compiler and Library/370):
              -- User's Guide and Reference
                 (CMS, MVS)
   SK2T1402   REXX/370 Compiler and Library V1R2.0
              -- Online Product Library
   LY19-6264  IBM REXX Compiler and Library/370:
              -- Diagnosis Guide (CMS, MVS)
   SB20-0020  The REXX Handbook
              -- Ed. Goldberg & Smith
```


System-specific books, grouped by system


```
   SC24-5708  AIX/6000:
              AIX REXX/6000 Reference

   SH24-5286  IBM REXX for Netware Reference Guide

   S01F-0271  OS/2 Version 1.3 Procedures Language
              2/REXX Reference
   S01F-0272  OS/2 Version 1.3 Procedures Language
              2/REXX User's Guide
   S10G-6268  OS/2 (Version 2.0) Procedures Language
              2/REXX Reference
   S10G-6269  OS/2 (Version 2.0) Procedures Language
              2/REXX User's Guide
   SR28-5250  OS/2 (Version 2.1) REXX Handbook
   GG24-4199  OS/2 REXX: From Bark to Byte (Redbook)

   SC24-5239  VM/SP: System Product Interpreter
              Reference
   SC24-5238  VM/SP: System Product Interpreter
              User's Guide
   SX24-5126  VM/SP: System Product Interpreter
              Reference Summary
   SB09-1326  VM/SP: System Product Interpreter Reference
              (Chinese)
   SB09-1325  VM/SP: System Product Interpreter
              User's Guide (Chinese)
   GG22-9361  The System Product Interpreter
              (REXX) Examples and Techniques -- Brodock
   SC12-1599  VM/SP: System Product Interpreter Handbuch
              (German: SC24-5239, July 1984)
   SC24-5357  VM/IS: Writing Simple Programs with REXX
   SC23-0374  VM/XA: System Product Interpreter Reference
   SC23-0375  VM/XA: System Product Interpreter User's Guide
   GH19-8118  CMS REXX Compiler General Information
   SH19-8120  CMS REXX Compiler User's Guide & Reference
   LY19-6262  CMS REXX Compiler Diagnosis Guide
   LN19-9048  CMS REXX Compiler Diagnosis Guide TNL
   SH19-8146  CMS REXX Compiler User's Guide and Reference
              -- Supplement
   GC24-5406  VM/SP: Program Update Info.
              -- REXX Language Enhancements

   LYC0-9075  VM/ESA: V1: REXX/370 LISTING
   SC24-5598  VM/ESA: R2: REXX/VM Primer
   SC24-5465  VM/ESA: R2.2: REXX/VM User's Guide
   SC24-5466  VM/ESA: R2.2: REXX/VM Reference
   ST00-8323  VM/ESA: R2.2: REXX/VM Reference Summary
   GC24-5607  VM/ESA: R2.2: REXX/EXEC Migration Tool

   SC28-1882  TSO/E V2R1.1 REXX User's Guide
   SC28-1883  TSO/E Version 2 REXX/MVS Reference

   SC23-3803  Using REXX to Access OpenEdition
              MVS Services

   SC24-5512  AS/400 Procedures Language 400/REXX
              Reference
   SC24-5513  AS/400 Procedures Language 400/REXX
              Programmer's Guide
   SC24-5552  AS/400 Procedures Language 400/REXX
              Reference, Version 2
   SC24-5553  AS/400 Procedures Language 400/REXX
```

```
              Programmer's Guide, V 2
  SBOF-6819   OS/400: REXX/400 Support

  SC33-6528   VSE/ESA: REXX/VSE User's Guide
  SC33-6529   VSE/ESA: REXX/VSE Reference
  LY33-9144   VSE/ESA: REXX/VSE Diagnosis Reference
  GC33-6533   VSE/ESA: REXX/VSE Licensed Program
                       Specifications
  SK2T-0063   VSE/ESA: REXX/VSE V1R1 Online Product
                       Library

  SH21-0482   REXX Development System for
              CICS/ESA and REXX Runtime
              Facility for CICS/ESA Guide and Reference
```


Applications and other REXX-related books


```
  GG24-1615   Using REXX in Practice: EXEC2 to
              REXX Conversion Experiences
  GG24-3401   REXX/EXEC Migration To VM/XA SP
  SC33-0478   GDDM REXX Guide
  SR21-0864   SRA VM Using the CMS System
              Product Interpreter
  SH20-7051   VM/SP System Product Interpreter:
              SQL/Data System
              Interface: Program Description/Operations
              Manual
  GG66-3144   NetView Release 3: REXX Presentation Guide
              -- Gibbons & Quigley
  GG66-3158   CMS Pipelines Tutorial
              -- Hartmann, Kraines, and Lynn
  GR28-2920   CUA 2001 VM Applications Core
              Functions Programmer's Reference Guide
              (CUA support for VM REXX applications)
  S246-0078   REXX Reference Summary Handbook (OS/2)
              -- Dick Goran
  SC23-3803   Using REXX to Access OpenEdition
              MVS Services
```

File: nr_32.html.

Last update was done on 18 May 1998 21:48:06(GMT +2).

The NetRexx Tutorial
- Appendix I: Installation

Appendix I: Installation

Installation on WIN/95 WIN/NT and SOLARIS

Download the JDK

Sun directly distributes the JDK for Windows/95, Windows/NT and Solaris (both SPARC and x86). The download can be performed from:

http://java.sun.com/products/jdk/1.1/index.html

Select the platform, read the download condition, and fetch the code using your preferred WEB browser.

NOTE: due to a problem with Netscape 3.01, I was forced to directly issue the FTP commands, in order to fetch the code.

```
ftp ftp.javasoft.com
> anonymous
> YOUR_EMAIL_ADDRESS
> bin
> cd pub/jdk1.1
> get jdk1.1.1-win32-x86.exe
> quit
```

Installing Java on AIX

Checking installation

Using your preferred editor, enter the following program, calling it **hellojava.java**.

```
+----------------------------------------------------------------------+
| class hellojava                                                      |
| {                                                                    |
|   public static void main (String args[])                            |
|    {                                                                 |
|     System.out.println("Hello World, from Java!");                   |
```

```
|     }
|  }
+----------------------------------------------------------------------+
                                                         hellojava.java
```

Then you type:

```
>javac hellojava.java         # compile the program
>java hellojava               # run it
```

If the output is the string "Hello World, from Java!" then you've almost done it!

Now you can try an applet. So edit the files **hellojavaa.java** and **hellojavaa.html**, as presented below.

```
+----------------------------------------------------------------------+
| import java.applet.Applet;
| import java.awt.Graphics;
|
| public class hellojavaa extends Applet {
|     public void paint(Graphics g) {
|       g.drawString("Hello world, from Java Applet!", 50, 25);
|      }
|  }
|
+----------------------------------------------------------------------+
                                                        hellojavaa.java
```

```
+----------------------------------------------------------------------+
| <HTML>
| <HEAD>
| <TITLE> Hello World </TITLE>
| </HEAD>
| <BODY>
| This is the applet:<P>
| <APPLET codebase="classes" code="hellojavaa.class" width=200 height=2|
| </BODY>
| </HTML>
+----------------------------------------------------------------------+
                                                        hellojavaa.html
```

```
>javac hellojavaa.java            # compile the program
>appletviewer hellojavaa.html     # run it
```

AIX known bugs

There is a bug in the AIX JIT compiler. This leads to errors like the following one, even in compiling the small **hello.nrx** program.

```
$java COM.ibm.netrexx.process.NetRexxC hello
NetRexx portable processor, version 1.120
Copyright (c) IBM Corporation, 1997.  All rights reserved.
Program hello.nrx
java.lang.ArrayIndexOutOfBoundsException: 20
        at netrexx.lang.RexxWords.space(Compiled Code)
```

```
        at netrexx.lang.Rexx.space(Compiled Code)
        at netrexx.lang.Rexx.space(Compiled Code)
        (...)
```

To turn OFF the JIT, just do:

```
SET JAVA_COMPILER=xxx
```

Download the NetRexx Distribution

The latest versions of NetRexx are available on IBM's WEB site at the following URLs:

http://www.ibm.com/Technology/NetRexx/nrdown.htm
USA Server

or at

http://www2.hursley.ibm.com/netrexx/nrdown.htm
UK Server

Installing NetRexx on UNIX

In the following example I assume that you want to install NetRexx in the directory:

```
~/src/NetRexx
```

and you've the working Java top tree in:

```
~/src/java/Java
```

This is the procedure:

1. Unpack the distribution

   ```
   > cd ~/src/NetRexx
   > uncompress NetRexx.tar
   > tar -xvf NetRexx.tar
   ```

2. Install the libraries and demo

   ```
   > cd ~/src/java/Java
   > cp ~/src/NetRexx/nrtools.tar.Z .
   > uncompress nrtools.tar
   > tar -xvf nrtools.tar
   ```

3. Set the environment variable CLASSPATH
   You need to add ~/src/java/Java/lib/NetRexxC.zip to the
   CLASSPATH environment variable

   This command will depend on your shell (csh, tcsh, ksh ...)

   ```
   > export CLASSPATH=$CLASSPATH:~/src/java/Java/lib/NetRexxC.zip
   ```

4. Test the installation

```
> cd ~/src/java/Java/bin
> java COM.ibm.netrexx.process.NetRexxC hello
> java hello
```

The following small script might save you some typing

```
+------------------------------------------------------------------------+
| echo 'java COM.ibm.netrexx.process.NetRexxC' $1                        |01
| java COM.ibm.netrexx.process.NetRexxC $1                               |02
+------------------------------------------------------------------------+
                                                                      nrc
```

Microsoft J++

The following recepy (originally provided by Bernhard Hurzeler <behurzeler@ucdavis.edu>) gives some information on how to get MS VJ++ and NetRexx working together.

```
 1. Put the files in their appropriate directories:

      NetRexxC.zip        -> c:\MSDEV\LIB
      NetRexxC.properties -> c:\MSDEV\LIB
      NetRexxR.zip        -> c:\MSDEV\LIB
      NetRexxC.bat        -> c:\MSDEV\BIN
      NetRexxC.cmd        -> c:\MSDEV\BIN
      nrc.cmd             -> c:\MSDEV\BIN
      nrc.bat             -> c:\MSDEV\BIN

  2. Set the CLASSPATH to:

      c:\Msdev\Lib\NetRexxR.zip;c:\Msdev\Lib\NetRexxC.zip;c:\Msdev\Bin

     On Windows NT 4.0, you follow the icons
       Start,
        Settings,
         Control Panels,
          System,
           Environment tab,
            System Variable

  3. Go to c:\MSDEV\BIN and type the commands:

     -- generate the java source
     > jview COM.ibm.netrexx.process.NetRexxC hello -keep nocompile
     -- compile it
     > jvc hello.java
     -- run
     > jview hello
```

File: nr_33.html.

Last update was done on 18 May 1998 21:48:07(GMT +2).

The NetRexx Tutorial

- Appendix Z: changes in this file

Appendix Z: changes in this file

This chapter will (of course) disappear in the final version.

```
*
* ver      date   pgs    action
* -------  ------ ----   ------------------------------------------
*
* v0r0035  250297  208   - HTML version + restructure of history file
*                        - put small corrections in chap 1
*
* v0r0032  200297  208   - add xsock (small) and xshell
*                          to the distribution
*
* v0r0032  180297  208   - write the RECFM F part of xfile
*                          with the I/O record access
*
* v0r0030  150297  206   - clean existing chap 9
*                          (before totally wrong)
*
* v0r0029  130297  206   - correct chap 11
*                        - add xexec example & warning
*                        - add tar.gz of examples and libraries.
*
* v0r0028  120297  204   - add other conversion examples in chap 4
*                        - build also a .zip version of the .ps
*                          Thanks to Francesc Roses for a pointer
*                          to a zip that compiles on AIX.
*
* v0r0012                - Rearrange the introduction and the Review.
*                          Restructure the Preface
*                          Rearrange the chapters in part 4
*                          Add the Tools chapter
*                          Put in Bernard's comments & fixes
*
* v0r0010                - Start writing NetRexx for REXXers
*
* v0r0001  020297        - First "public" presentation of the doc.
*                          This is what Bernard and Mike saw.
*
*
*
*
*
```

File: nr_34.html.

Last update was done on 18 May 1998 21:48:08(GMT +2).

The NetRexx Tutorial
- Index

Index

- $?
- $status

- %, %

- '00'X character
- '0D'X

- *

- +

- , as continuation character

- -
- - as continuation character

- /, /
- /* */
- //

- ;

- \x, \x
- \X

- abex1.nrx
- abex2.nrx
- abs
- abstract
- abstract class
- abstraction
- abuttal
- acos
- acosh
- Additional instructions
- AIX install
- AIX JIT bug
- aphello.html
- aphello.nrx
- API documentation
- applets
- Applets
- Applications
- arg
- args
- array_exa.nrx
- ArrayIndexOutOfBoundsException
- arrays
- arrex1.nrx
- asar.nrx
- asar.rex
- asin
- asinh
- assignments
- associative arrays
- atan
- atanh
- avoid NEWLINE char
- AwkTools

- base64

- basic file operations
- bean
- bibliography
- binary files
- BINARY numbers
- blank lines
- Blocks of READ
- Blocks of WRITE
- build libraries

- C++ function pointer
- Cafe'
- Call
- Call command
- Calling a program
- cannot find constructor
- class instances
- classes, classes
- CMSpipes
- codeex.nrx
- command line parser
- comments
- Complex Data Structures
- composers.nrx
- compound variables
- concatenation
- constructor
- cont_exa.nrx
- Continuation Character
- continuation character
- Control FAQ
- convert to CU
- cos
- cosh
- current directory

- d2c
- d2x
- daemon
- data structures
- database

- date
- date conversion tool
- daytime
- daytime.nrx
- default precision
- delim_exa.nrx
- Delimiter Character
- design patterns, design patterns
- determine Operating System
- do/end
- docs
- dumping files in HEX
- dyna2.nrx
- dyna3.nrx

- e
- elapsed time
- environment
- error compiling
- Error unmarshaling return
- eval
- eval.nrx
- exceptions, exceptions
- exec()
- exit
- exit status
- exitValue()
- expose
- expp1.nrx
- expp2.nrx
- expp3.nrx
- expression parser

- FAQ
- fexa1.nrx
- fexist
- File
- file existence check
- file operations
- file read
- file Read and Write

- file write
- file.separator
- finally
- find
- find which OS
- finger, finger
- finger.nrx
- finger1.nrx
- Finite Element Method
- fixed format
- fixed length records
- foreach
- fork()
- forkex1.rex
- format
- Frequently Asked Questions
- FTP client program
- FTP get
- FTP put
- function calls
- function pointer in C and C++
- functions

- gauss.nrx
- gcd.nrx
- GetRuntime()
- giga
- greatest common divisor
- GUI

- Hanoi
- hanoi.nrx
- hash
- hashing function
- hedit.nrx change
- hedit.nrx linedis
- hedit.nrx save
- hello.nrx
- hellojava.java
- hellojavaa.html
- hellojavaa.java

- HEX
- HEX char range
- HEX dump
- HEX edit
- HEX numbers
- HEX quantities
- hexadecimal strings
- history
- history.nrx dump
- history.nrx retrieve
- history.nrx save

- IBM redbook
- if/then/else
- IMAP client
- IMAP protocol
- imapt.nrx
- index
- indexed files
- indexed string
- infix
- initialise
- input line arguments
- installation
- Installation
- instanceof
- interact.nrx
- interpreter
- ISO 2015 & 2711
- iterate

- J++, J++
- JAR
- java
- Java Developer Kit
- Java JDK
- Java on AIX
- JAVA String[] arrays
- Java version
- Java Virtual Machine
- java.class.path

- java.lang.IllegalAccessError
- java.lang.Object File
- java.lang.Process
- java.lang.Runtime
- java.lang.Thread
- java.version
- JAVA_COMPILER env variable
- javabeans
- javascript
- JDBC
- jdbct1.nrx
- JDK
- JIT
- JPEG
- JPG
- jpginfo.nrx
- jsc.html
- julian date
- just in time compilers

- kilo

- latest NetRexx version
- leave
- length and width of a JPG
- linked lists
- list expansion
- list files in directory
- literal parsing
- literal strings
- lls.nrx
- loop
- loop over
- loop/while/until
- lower, lower
- ls

- mailing list
- main arguments
- main()

- matching pattern
- max
- measure time
- mega
- memory model
- method main()
- method overloading
- methods
- Microsoft J++, Microsoft J++
- MIME
- monthfile.nrx
- multiple
- multiple constructors

- NetRexx mailing list
- NetRexx sources
- nnt.nrx
- nnt1.nrx
- NNTP client
- NNTP protocol
- nodisp.nrx
- NOP
- NotSerializableException
- nr.HISTORY
- nrc
- nrenv
- nrenv.nrx
- Numbers
- numperf
- numperf.nrx

- object model
- objects, objects
- Operations on BINARY
- Operations on HEX
- Original Reusable Objects
- ORO
- OROMatcher
- OS version
- over

- p-code
- packages, packages
- parrot.nrx, parrot.nrx
- parrotc.nrx
- parse
- parse pull
- parsearg.nrx
- parsing
- path.separator
- pattern, pattern
- pattern design
- patterns
- PERL associative arrays
- Perl5 Regular Expressions
- PerlTools
- pex1.nrx
- pi
- pipes
- polish
- portn.nrx
- precedence
- precision
- Prerequisites
- printStackTrace()
- procedure
- process control
- program name
- Programs
- ps
- pull

- qsn.nrx main
- qsn.nrx partition
- qsn.nrx sort_qsnr
- quicksort non recursive

- random
- re-entrant
- read file
- read file line

read implementation
- readst.nrx
- RECFM F
- RECFM V
- recursion
- redbook on Netrexx
- regexp
- regular expression
- Remote Method Invocation
- resume of do instruction
- REXX FAQ
- REXX procedures
- RFC 1064
- RFC 1341
- RFC 1342
- RFC 867
- RFC 977
- rfile.nrx
- rfileclie.nrx
- rfileimpl.nrx
- rfileserv.nrx
- RMI
- rmic, rmic
- rmiregistry, rmiregistry
- roundup.nrx
- runnable.nrx
- Runtime
- rxfile

- say
- sclie.nrx
- select
- SG24-2216-0
- shell arguments
- simple1.nrx
- simple2.nrx
- simple3.nrx
- simple4.nrx
- simple5.nrx
- simple6.nrx
- simple7.nrx
- Simultaneous Linear Equations Solution
- sin

- Singleton.nrx
- sinh
- sleep
- Sockets
- Solaris SPARC
- Solaris x86
- sort
- source download
- Special Characters
- special characters
- special variables
- SQL
- sqrt
- sserv.nrx
- stack trace
- stanza
- start rmiregistry
- state
- static
- stem
- stream I/O model
- string concatenation
- string sorting
- String[]
- strings
- strings[]
- strstrict.nrx
- subroutines
- sun.net.ftp
- sun.net.TelnetInputStream
- syex1.nrx
- syex2.nrx

- tan
- tanh
- tarray.nrx
- tcl1.nrx
- tcl2.nrx
- TelnetInputStream
- tfix.nrx
- tgm1.nrx
- tgm1.rex

thread API
- thread definition
- Thread.sleep(MILLISEC)
- threads
- thrt0.nrx
- thrt1.nrx
- time
- Time.nrx
- TimeCl.nrx
- timeexa1.nrx
- TimeImpl.nrx
- timeout on a command
- timer class
- timestamp
- tnr1.nrx
- tnr1.rex
- tnr2.nrx
- tnr2.rex
- towers of Hanoi
- trace
- translate
- translate to lowercase
- translate to uppercase
- tree for ps command
- tstring1.nrx
- tvec3d.nrx
- tvec3ds.nrx
- tvecL01.nrx
- twb.nrx

- UCSD Pascal
- undefined constructor
- unimplemented interface method
- UNIX
- UNIX streams
- upper, upper
- URL
- user.dir
- userid
- userid()
- using a class

- vector
- vectorLo.nrx
- Venn Diagram
- version
- virtual
- virtual class (C++)
- volt.nrx
- voltcl.nrx
- voltimpl.nrx

- w3dmp.nrx
- w3dmp1.nrx
- watchdog
- WEB
- WEB pages
- Windows Magazine
- Windows/95
- Windows/NT
- word
- wordpos
- write file
- write implementation
- www.winmag.com

- xarray.nrx ba2x
- xarray.nrx bagrepx
- xarray.nrx copy
- xarray.nrx dump
- xbuild.nrx main
- xdate
- xdt0.nrx
- xdt1.nrx
- xexec
- xfile
- xfile.nrx read
- xfile.nrx readbuf
- xfile.nrx recio
- xfile.nrx recwrite
- xfile.nrx state
- xfile.nrx write
- xfile.nrx writebuf

- xfile.read()
- xftp
- xftp.nrx xget
- xftp.nrx xls
- xftp.nrx xmore
- xftp.nrx xput
- xftp1.nrx
- xmath.nrx binop
- xmath.nrx dotify
- xmath.nrx gauss
- xmath.nrx gcd
- xmath.nrx hexop
- xmath.nrx n2cu
- xmath.nrx random
- xmath.nrx s2h
- xrange
- xshell.nrx
- xshell1.nrx history
- xshell1.nrx historyd
- xsock.nrx getservbyname
- xsock.nrx hostname
- xsock.nrx open
- xstring.nrx a2m
- xstring.nrx a2s
- xstring.nrx censure
- xstring.nrx cmdline
- xstring.nrx display
- xstring.nrx evalrpn
- xstring.nrx hash
- xstring.nrx listexpand
- xstring.nrx m2a
- xstring.nrx option
- xstring.nrx s2a
- xstring.nrx sort
- xstring.nrx translate
- xstring.sort
- xsys.nrx elapsed
- xsys.nrx reset
- xsys.nrx xexec
- xsys.sleep(SEC)
- xsystem.nrx dump
- xvector.nrx add
- xvector.nrx mag
- xvector3d.nrx

- ||

```
File: nr_35.html.
```

Last update was done on 18 May 1998 21:48:09(GMT +2).