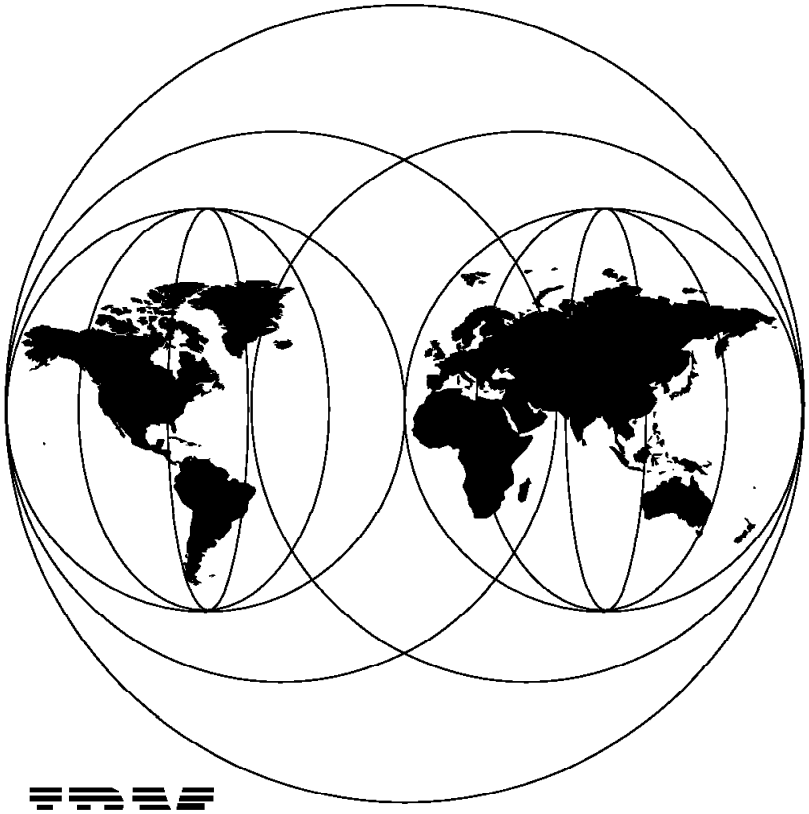
SG24-2216-00

Creating Java Applications Using NetRexx

September 1997



IBM

International Technical Support Organization
San Jose Center

International Technical Support Organization

SG24-2216-00

Creating Java Applications Using NetRexx

September 1997

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix B, “Special Notices” on page 273.

First Edition (September 1997)

This edition applies to Version 1.0 and Version 1.1 of NetRexx with Java Development Kit 1.1.1 for use with the OS/2 Warp, Windows 95, and Windows NT operating systems.

Because NetRexx runs on any platform where Java is implemented, it applies to other platforms and operating systems as well.

SAMPLE CODE ON THE INTERNET

The sample code for this redbook is available as nrxredbk.zip on the ITSO home page on the Internet:

```
ftp://www.redbooks.ibm.com/redbooks/SG242216
```

Download the sample code and read “Installing the Sample Programs” on page 4.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Preface

NetRexx is a new human-oriented language that makes writing and using Java classes quicker and easier than writing in Java. NetRexx combines the ease of use and flexibility of Rexx with the robust structure and portability of Java.

This redbook covers all aspects of NetRexx, from simple scripting programs to applications and applets using such advanced features as graphical user interfaces with animation, access to relational databases, communication over TCP/IP sockets, client/server programming using remote method invocation (RMI), Common Gateway Interface (CGI) programming, and JavaBeans. The sample programs are freely available on the Internet.

This redbook applies to NetRexx Version 1.0 and 1.1, and the Java Development Kit (JDK) Version 1.1.1. The sample programs were tested on Windows 95, Windows NT, and OS/2 Warp; they should also run on other platforms that support JDK 1.1.

How This Document is Organized

Chapter 1, "Introduction" explains the purpose of NetRexx, the installation of the product and the sample programs, and the NetRexx documentation.

Chapter 2, "Starting with NetRexx" describes a simple NetRexx program and shows how to compile and run it. It also covers the file types used by NetRexx.

Chapter 3, "The NetRexx Compiler" explains in detail how NetRexx programs are translated into Java programs. It covers all compile options and explains how to invoke the compiler from a NetRexx or Java program.

Chapter 4, "The NetRexx Language" contains a comprehensive introduction to the NetRexx language.

Chapter 5, "Using NetRexx As a Scripting Language" explains how NetRexx is used to write simple, straightforward programs without explicitly coding Java classes. Included are subroutines and functions, handling of parameters, and invocation of non-Java programs.

Chapter 6, "Creating and Using NetRexx Classes" introduces object-oriented programming with NetRexx. Classes, methods, inheritance, interfaces, class libraries, and packages are explained in detail.

Chapter 7, "Creating Graphical User Interfaces" shows how NetRexx creates applications and applets with elaborate GUIs. Many sample programs are provided to illustrate certain aspects of GUI programming, such as layout managers, menus, dialogs, event handling, images, and keyboard input. A package of useful classes is developed to simplify GUI programming and solve common GUI problems.

Chapter 8, "Threads" introduces threads for parallel processing, including starting, stopping, and synchronization.

Chapter 9, "Handling Files" explains the different ways NetRexx can handle flat files, including line-mode, byte-oriented, and data-oriented input/output. It also covers serialization, that is, the storage and retrieval of objects in files.

Chapter 10, "Database Connectivity with JDBC" shows how NetRexx programs access relational databases by using the Java Database Connectivity API. Sample programs are developed to access the DB2 sample database.

Chapter 11, "Network Programming" shows how to write network applications with sockets, universal resource locators (URLs), and RMI. It concludes with a client/server implementation of the JDBC application.

Chapter 12, "Using NetRexx for CGI Programs" discusses how NetRexx can be used to write CGI programs for a Web server. The sample programs access DB2 to create HTML pages.

Chapter 13, "Creating JavaBeans With NetRexx" introduces JavaBeans and shows how to create simple beans with NetRexx for use with VisualAge for Java.

Chapter 14, "Why NetRexx?" summarizes the advantages of using NetRexx over Java.

Appendix A, "Redbook Package Reference" describes the classes of the redbook package. The redbook package simplifies GUI programming by providing solutions for many common problems.

The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Peter Heuchert is a software developer in Germany. He has 10 years of experience in software development. His areas of expertise include software design, Smalltalk, C, and Rexx.

Frederik Haesbrouck is a systems engineer in Belgium. He has one year of experience in the services field. He holds a degree in computer science from the University of Ghent. His areas of expertise include C++, C, Smalltalk, object-oriented analysis and design, the VisualAge product family, and Rexx. He is IBM Certified in OS/2.

Norio Furukawa is an IT specialist in Japan. He has 15 years of experience in software development, education, and technical support. His areas of expertise include VM/CMS, OS/2, and the Rexx product family.

Ueli Wahli is a Consultant Application Development Specialist at the International Technical Support Organization, San Jose Center. He writes extensively and teaches IBM classes worldwide on all areas of application development and object-oriented technology. Before joining the ITSO 13 years ago, Ueli worked in technical support in Switzerland as a Systems Engineer. He holds a degree in Mathematics from the Swiss Federal Institute of Technology. His areas of expertise include many programming languages, visual development environments, as well as data dictionaries, repositories, and library management. He has written many redbooks on these topics.

Thanks to the following people for their invaluable contributions to this project:

Christian Michel, German Software Development Lab, Boeblingen, for his thorough review of the book, additional ideas, and the porting of all examples to OS/2.

Mike Cowlishaw, IBM Fellow, UK Laboratories Hursley, for inventing NetRexx and supporting our effort all the way.

Comments Welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in “ITSO Redbook Evaluation” on page 283 to the fax number shown on the form.
- Use the electronic evaluation form found on the Redbooks Web sites:

  For Internet users `http://www.redbooks.ibm.com`
  For IBM Intranet users `http://w3.itso.ibm.com`

- Send us a note at the following address:

  `redbook@vnet.ibm.com`

Chapter 1. Introduction

In this chapter we introduce the NetRexx product and provide brief instructions for installing NetRexx in a Java environment.

What Is NetRexx?

NetRexx is a new human-oriented programming language designed as an alternative to the Java language. NetRexx compiles to the Java Virtual Machine and enables programmers to create programs and applets easily and conveniently.

NetRexx combines the strengths of two very different languages: Rexx and Java. The result is a language that is tuned for both scripting and application development. For example, NetRexx has its own String class, like the classic Rexx strings, and uses classes and exceptions like Java.

Because the NetRexx compiler is written in NetRexx, it runs on every platform that supports Java.

Design Objectives

NetRexx is designed to make life easy for users, not for compiler developers.

NetRexx automates many tasks, such as variable declaration and selection, without the risk of unstable programs, enabling programmers to concentrate on writing the applications.

NetRexx takes the object model from Java and the safe syntax from classic Rexx, making it an object-oriented language with a readable, easy-to-understand syntax.

Why NetRexx?

We could tell you that right now, but we want you to read the book first!

If you cannot wait, see Chapter 14, “Why NetRexx?” on page 259.

Installation

The NetRexx package is shipped in two formats:

NETREXX.ZIP used for OS/2, Windows 95, and Windows NT

netrexx.tar.Z commonly used on AIX and other UNIX systems

To install NetRexx, download the zip file to a drive (or directory) of your choice and unpack it (see Figure 1).

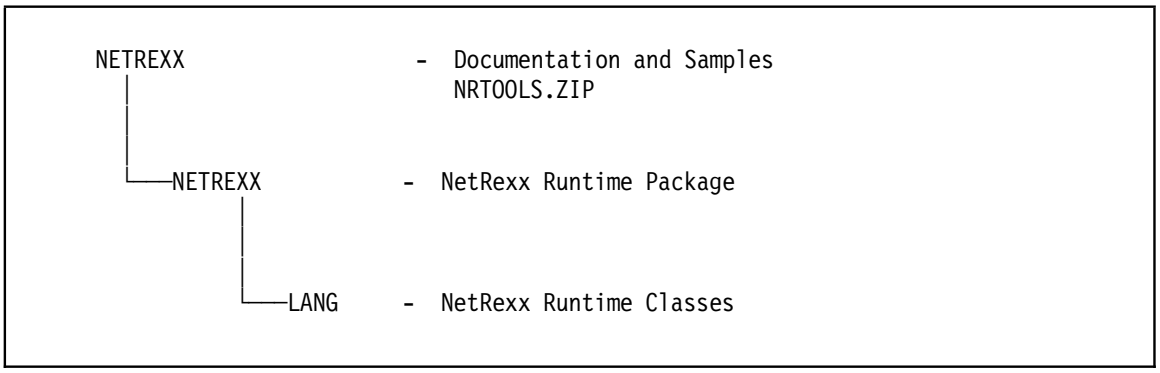


Figure 1. Directories Added by Unpacking NETREXX.ZIP

Important notes for unpacking NetRexx:

- Ensure that you are unzipping to a disk that supports long file names (for example, a high-performance file system (HPFS) disk on OS/2).
- Ensure that your UnZip program creates the subdirectories stored in the zip file.
- If you are using Info-ZIP, use Version 5.12 (August 1994) or later.

Copy the NRTOOLS.ZIP file from the NETREXX directory to your Java home directory, for example, JAVAOS2 or JAVA11 for OS/2, and JDK1.1.1, for Windows.

With the Java home directory as your current directory, unpack NRTOOLS.ZIP to add the zip files for the NetRexx runtime and compiler classes and the compiler error message file to the LIB directory, and some command files and a test program to the BIN directory (see Table 1).

Table 1. Files Added by Unpacking NRTOOLS.ZIP

Path	File	Description
BIN	hello.nrx	Sample “Hello World” program for installation verification
BIN	NetRexxC.cmd	NetRexx compiler command in Rexx (OS/2)
BIN	NetRexxC.bat	NetRexx compiler command for Windows
BIN	nrc.cmd	Abbreviated command file, invokes NetRexxC.cmd
BIN	nrc.bat	Abbreviated .bat file, invokes NetRexxC.bat
LIB	NetRexxC.zip	NetRexx Compiler and Runtime Classes
LIB	NetRexxR.zip	NetRexx Runtime Classes
LIB	NetRexx.properties	Java properties file used by NetRexx compiler for error messages (NetRexx 1.0 only)

Check the names of all files in the LIB directory. Pay special attention to the case of the files. If the name of the NetRexxC.properties file is truncated, the package has not been unpacked correctly.

For Java to find the NetRexx classes, you must update the CLASSPATH environment variable by adding the NetRexxC.zip file to the CLASSPATH setting, after the standard Java classes.zip file. Add the full path (disk, directory, and file specification) of your NetRexxC.zip file:

```
SET CLASSPATH=........;d:\...javahome...\LIB\NetRexxC.zip
                       d:\JAVA11\LIB\NetRexxC.zip          <=== OS/2
                       d:\JDK1.1.1\LIB\NetRexxC.zip        <=== Windows
```

On a machine where you intend to install the runtime facility only, add NetRexxR.zip to the CLASSPATH instead.

For OS/2 the CLASSPATH variable is set in the CONFIG.SYS file, for Windows 95 it is set in AUTOEXEC.BAT.

Reboot the machine.

Alternative Installation

You can unpack NRTOOLS.ZIP in the NETREXX directory itself to create a BIN and a LIB subdirectory. Add the NETREXX\BIN subdirectory to the system PATH so that the NetRexx compiler can be found. Set the CLASSPATH environment variable to point to the NetRexxC.zip file:

```
SET PATH=........;d:\NetRexx\bin
SET CLASSPATH=........;d:\NetRexx\lib\NetRexxC.zip
SET NETREXX_HOME=d:\NetRexx
```

The NETREXX_HOME environment variable points to the lib\NetRexxC.properties file for error messages during compilation; however, it only works on OS/2. For Windows systems you must copy the NetRexxC.properties file to the LIB subdirectory of the Java home directory to get proper error messages during compilation. (Note: This file does not exist in NetRexx 1.1.)

Installation Verification

To verify the installation, change the directory to the Java BIN (or NetRexx BIN) directory where the hello.nrx program resides and enter the *nrc -run hello* command. This command runs the compiler and starts the program after successful compilation (see Figure 2).

```
[C:\javaos2\bin]nrc -run hello
NetRexx portable processor, version 1.00
Copyright (c) IBM Corporation, 1997.  All rights reserved.
Compilation of 'hello.nrx' successful
Running hello ...
Hello World!

[C:\javaos2\bin]
```

Figure 2. Compilation and Run of hello.nrx

Software Prerequisites

The NetRexx compiler (NetRexxC) is written in NetRexx and should run on any Java platform that supports the JDK.

The examples in this book are based on and tested with NetRexx Version 1.0 and 1.1, using JDK 1.1.1; most of them will not run on JDK 1.0.2 or earlier releases.

Installing the Sample Programs

All of the sample NetRexx programs discussed in this redbook are available on the Internet (see Figure 3).

```
ftp://www.redbooks.ibm.com/redbooks/SG242216

or:   ftp ftp.almaden.ibm.com
      cd redbooks\SG242216
      binary
      get nrxredbk.zip
```

Figure 3. NetRexx Sample Programs on the Internet

Download the NRXREDBK.ZIP file from the Internet. Create a directory, for example, NRXREDBK. Move the zip file into the new directory and unzip the code to create a directory structure like that shown in Figure 4.

```
Directory                    Sample Programs for Chapter
-------------------------    -----------------------------------/o
NRXREDBK
   FIRST                     Starting with NetRexx
   COMPILER                  The NetRexx Compiler
   LANGUAGE                  The NetRexx Language
   EXCEPTIONS                 - simple exception programs
   SCRIPT                    Use NetRexx as Scripting Language
   GUI                       Graphical User Interfaces
      many subdirectories     - individual GUI functions
   THREAD                    Threads
      CONSUMER                - consumer example
      SYNCH                   - synchronization example
      PHILFORK                - philosophers' forks example
   FILE                      File Handling
   JDBC                      Java Database Connectivity (JDBC)
   NETWORK                   Network Programming
      NET                     - sockets
      URL                     - URLs
      RMI                     - remote method invocation (RMI)
      RMICHAT                 - RMI chat sample
      RMIJDBC                 - RMI sample with JDBC
   CGI                       Using NetRexx for CGI Programs
   NRXBEANS                  NetRexx Beans
      LAB                     - Counter, Light, Timer, LED
      SAMPLE                  - Sample bean, applets using beans
   REDBOOK                   Redbook Package Classes
      GUI                     - 15 gui classes (source)
      EXCEPTION               - RedBookException classes (source)
      UTILITY                 - RedbookUtility class (source)
```

Figure 4. Sample Programs

Installing the Packages of this Redbook

The main directory, NRXREDBK, must be added to the CLASSPATH to make the packages available when running some of the sample programs:

```
SET CLASSPATH=.....;d:\NRXREDBK
```

The source files for the REDBOOK package are contained in the REDBOOK subdirectories. Other packages are stored in the NRXBEANS subdirectory and in the NETWORK\RMI* subdirectories.

We provide a BUILD.BAT (Windows) or BUILD.CMD (OS/2) file in each directory for compiling the source into the package subdirectory.

Note: You can also create an uncompressed ZIP file containing all the classes of the REDBOOK, NRXBEANS, and NETWORK\RMI* subdirectories and point to it from the CLASSPATH:

```
SET CLASSPATH=.....;d:\NRXREDBK\nrxclass.zip
```

NetRexx Documentation

NetRexx documentation comes in three flavors:

- The language specification, *The NetRexx Language*, by M. F. Cowlishaw, Prentice Hall, 1997, ISBN 0-13-806332-X, IBM number SR23-7771
- *Online documentation* in HTML format, in the NetRexx directory:
  - *NetRexx 1.xx*, nrdocs.htm, the master HTML file containing pointers to the other documents
  - *NetRexx User's Guide*, doc-nrinst.htm
  - *NetRexx Language Quick Start*, doc-nrover.htm
  - *NetRexx Samples and Examples*, nrsample.htm
  - *NetRexx 1.xx Links*, nrlinks.htm, containing links to other sources on the Web
  - *JavaBeans Support in NetRexx - Draft*, nrbean.htm, or *NetRexx Language Supplement*, nrlsupp.htm
- *NetRexxD.zip*, which contains the license agreement and the language specification as a postscript file (nrldef.ps)

NetRexx Home Page on the Internet

The NetRexx product and additional information are available on the NetRexx home page:

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

Extract of the information on the home page:

- Latest product code
- News
- Reviews
- Tutorial
- Samples and freely available code (FAC)
- Frequently asked questions (FAQs)
- Packages (RxFile, RxDBase, MaxBase)

Java Toolkit Documentation

The documentation for the JDK is provided as HTML files with the code. The master file is called *index.html*. The Java platform API with documentation for all the classes is called *api**packages.html*.

Chapter 2. Starting with NetRexx

In this chapter we write a small NetRexx program and list the different file types that NetRexx uses.

Our First NetRexx Program

We are sorry, but we do not present the 4232th copy of a “Hello World” program in this book.

Our first NetRexx program is a mathematical program. It calculates the factorial of a given number. Of course this is not a big deal, but try it with languages other than Rexx, and you will see. Figure 5 shows the program, which consists of a few statements to print a prompt for a number, get your input, calculate the factorial, display the result, and do basic error handling.

```
/* first\Factor.nrx

   This is our first NetRexx program.   The program asks the user for a
   number and calculates the factorial of the given number.
   You should try big numbers. */

numeric digits 64 -- switch to exponential format when numbers become
                  -- larger as 64 digits
say 'Factorial program'
say '-----------------------------------------------------------'
say 'Input a number: \-'
do
   n = int ask       -- Gets the number, must be an integer
   if n < 0 then signal RuntimeException
   fact = 1          -- Initial value
   loop i=1 to n
      fact = fact * i
   end
   say n'! =' fact
 catch RuntimeException
   say 'Sorry, but this was not a positive integer'
end
```

Figure 5. Our First NetRexx Program: Factor.nrx

Compile the program:

```
d:\NrxRedBk\first>nrc Factor
```

Now run it:

```
d:\NrxRedBk\first>java Factor
```

Here is some sample program output:

```
Factorial program
-------------------------------------------------------------
Input a number: 49
49! = 608281864034267560872252163321295376887552831379210240000000000

Factorial program
-------------------------------------------------------------
Input a number: 5000
5000! = 4.228577926605543522201064200233584405390786674626646748849782400E+16325

Factorial program
-------------------------------------------------------------
Input a number: Peter
Sorry, but this was not a positive integer
```

Try the program on your own. Try to fool it.

Note for Java programmers

You can use NetRexx as a scripting language. There is no need to define a class and a main method to run a few statements. For more information see Chapter 5, “Using NetRexx As a Scripting Language” on page 49.

Now let’s describe the first program in detail:

- The first statements in the program are a comment and the basic print statements:

```
/* first\Factor.nrx

   This is our first NetRexx program.  The program asks the user for a
   number and calculates the factorial of the given number.
   You should try big numbers. */

numeric digits 64 -- switch to exponential format when numbers become
                  -- larger as 64 digits
say 'Factorial program'
say '-------------------------------------------------------------'
say 'Input a number: \-'
```

  The numeric digits instruction tells NetRexx to use an exponential form when the number becomes larger than 64 digits.

  The \- in the last say statement suppresses the line termination so that the user input is shown on the same line as the text of the say instruction.

- The do statement defines a code block and is used to catch errors:

```
do
   ...
 catch RuntimeException
   say 'Sorry, but this was not a positive integer'
end
```

- A line from the keyboard is read and converted to an integer number:

```
n = int ask
```

- If the number is not positive, an error is signaled:

```
if n < 0 then signal RuntimeException
```

- The factorial is calculated and printed:

```
loop i=1 to n
   fact = fact * i
end
say n'! =' fact
```

If you are interested in the exact number of the factorial of 1000, change the numeric digits statement to 10000 and try the program again.

File Types Used by NetRexx

The NetRexx compiler translates the NetRexx programs to Java and then compiles the generated Java programs. The NetRexx compiler uses or generates these file types:

***.nrx**	NetRexx program files
***.class**	Compiled NetRexx or Java program files
***.crossref**	Cross reference file—lists the variables, their types, and where they are used
***.java.keep**	NetRexx program translated to Java (see option *-keep* in "Compile Options" on page 14)
***.java**	Temporary generated Java program—renamed to *.java.keep or erased after compilation (see option *-nocompile* in "Compile Options" on page 14)

Chapter 3. The NetRexx Compiler

In Chapter 2, “Starting with NetRexx” on page 7 we show how to compile and run a NetRexx program with this simple syntax:

```
nrc Factor
java Factor
```

or, even simpler:

```
nrc -run Factor
```

In this chapter we explain in detail how the NetRexx compiler works.

Command Files

The NetRexx package provides some convenient command files to compile and run your NetRexx programs. These command files are available on only some platforms, so check the documentation of the package or the NetRexx Internet home page for the availability of versions of the command files for the platform you use.

NetRexxC.cmd OS/2 REXX script file that takes the file names and the options as arguments. It also supports the -run option that runs the programs after they have been compiled.

nrc.cmd OS/2 REXX script file, abbreviation of NetRexxC.cmd

NetRexxC.bat DOS batch file for the Windows platforms, a (simple) clone of NetRexxC.cmd. Be careful with the -run option; it has to be the first argument, and it is case sensitive.

nrc.bat DOS batch file, abbreviation of NetRexxC.bat

For more detailed information, look at the content of the command files.

The command files are provided to make your life (and that of your keyboard) easier. The native method of compiling and running NetRexx programs looks like this:

```
java COM.ibm.netrexx.process.NetRexxC Factor.nrx
java Factor
```

which is the equivalent of:

```
nrc -run Factor
```

What happens here?

The NetRexx compiler (*NetRexxC*) translates NetRexx code into Java byte code.

NetRexxC itself is a Java class from the *COM.ibm.netrexx.process* package. Because it has a main method, the NetRexxC class is a real Java application (or Java program), and so it can be run with the Java interpreter:

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

The file to be compiled is supplied as an argument to this Java program, NetRexxC.

The generated Java byte code is stored in a file with the *.class* extension. In our case the compiler creates the file:

```
Factor.class
```

This generated file also represents a real Java application that can be executed by entering:

```
java Factor
```

The NetRexxC (or nrc) command file automatically executes *java factor* when the -run option is used. When you supply more than one file name, the sources are compiled first and then run—in the same order.

Note, however, that the -run option is provided by the command files and not by the compiler.

Arguments and Return Codes

The compiler can take two kinds of arguments, file names and options. The file names represent NetRexx source files to be compiled together. For example:

```
java COM.ibm.netrexx.process.NetRexxC Pinger qtime
```

As you can see, there is no need to enter the file names with their default extension of *.nrx*.

You can also supply options as arguments. The options always begin with a hyphen (-), for example:

```
java COM.ibm.netrexx.process.NetRexxC Pinger -keep
```

This particular option tells the compiler to save the intermediate Java code that NetRexxC generates and passes to the Java compiler. The intermediate Java code is stored in a file with the extension of *.java.keep*. We discuss these options in more detail in “Compile Options” on page 14.

The compiler always returns a return code, which can be 0, 1, or 2. A return code of 0 indicates that the compiler found no errors and no warnings. A return code of 1 indicates that there were warnings, and a return code of 2 indicates that the compile was not successful. In the latter case you will get some error messages, and the generated Java file (with extension *.java*) is kept instead of the anticipated Java class.

How Does the Compiler Work?

In this section we discuss the implementation of the NetRexxC class in more detail to explain what happens in case of an error.

The NetRexxC class runs in two phases:

1. Translate NetRexx code into Java source code
2. Let the Java compiler generate Java byte code from the Java source

The first phase transforms the NetRexx source into Java source. Errors encountered in this phase are reported with the NetRexx error codes. The explanation that comes with the error codes should help you on the way to correct the errors.

If a more detailed message is not available, as in this case:

```
NetRexx portable processor, version 1.00
Copyright (c) IBM Corporation, 1997.  All rights reserved.
  9 +++ elect
    +++ ^^^^^
    +++ Error: keyword.expected (sorry, full message unavailable)
```

the system gives the (cryptic) error and appends the message “Sorry, full message unavailable.”

Similar to the Java compiler, NetRexxC uses a *.properties* file to store its warning and error messages. Therefore, if you encounter a situation in which the compiler tells you: “Sorry,...,” check that the *NetRexxC.properties* file is present in the LIB subdirectory of the NetRexx home directory.

The NetRexx home directory (netrexx.home) is usually the same as the Java home directory. To change the home directory in the OS/2 environment, use the environment variable called *NETREXX_HOME*, or use the -D option of the Java interpreter when invoking the compiler:

```
java -Dnetrexx.home=d:\NetRexx COM.ibm.netrexx.process.NetRexxC Factor
```

which sets the *d:**NetRexx* directory path as the NetRexx home directory. Under Windows 95 and Windows NT you must copy the NetRexxC.properties file to the LIB subdirectory of the Java home directory.

In the second phase of the execution, NetRexxC compiles Java code into byte code, using the default Java compiler, *javac*. Determining what went wrong when you get errors in this phase is not intuitive.

In the occasional case when javac fails to compile the Java source, there will be a file with the extension of *.java* (the input file for javac) in the current directory. The errors that the Java compiler reports are mostly due to incorrect conversions from NetRexx constructs to the corresponding Java constructs. NetRexx reports an error if the .java file already exists. Delete the file before you restart the NetRexx compiler, or use the REPLACE compiler option.

Invoking the Compiler from NetRexx or Java

You can use NetRexxC as a normal Java class. For this purpose the compiler provides the *main* method. This is its signature (see “Signature of Methods” on page 60):

```
method main(arg=Rexx) constant returns int
```

To use NetRexxC from within NetRexx (or from plain Java), call the main method with the file names and the options provided in a REXX string. Figure 6 shows an example of the use of the NetRexxC class from within a NetRexx program.

```
/* compiler\NrcAsClass.nrx

   This NetRexx program shows the use of the NetRexxC compiler as a normal class */

argument = 'Factor -nocrossref'
rc = COM.ibm.netrexx.process.NetRexxC.main(argument)
select
   when rc=0 then say 'Compilation was OK.'
   when rc=1 then say 'Check the warnings!'
   when rc=2 then say 'Some errors occurred!'
   otherwise say 'NetRexxC returned an unexpected returncode:' rc
end
```

Figure 6. Invoking the NetRexx Compiler from a NetRexx Program

Compile Options

You can supply two kinds of options to the compiler: compiler-only options (see Table 2) and the options you can write in the source program, using the OPTIONS keyword (see Table 3 on page 15).

Compiler-Only Options

Table 2 lists the compiler-only options.

Table 2. Table of Compiler-Only Options for NetRexxC

	Description
-keep	When this option is specified, NetRexxC saves the intermediate Java source file in a file with the *.java.keep* extension.
-nocompile	This option instructs NetRexxC to stop after the first phase; the generated Java source code is kept in a file with the (normal) *.java* extension, so that you can easily compile it further using another Java compiler.
-time	This option displays processing times for all compiled files: • Translation time • Compile time • Total time

Options Keyword

Table 3 lists all options that you can use on the command line or in the NetRexx source code, using the OPTIONS keyword. For a more detailed explanation of the options marked with an asterisk in the *More* column, see "More Details on Options" on page 16.

The options are identified by their name. To set an option to *off*, you add the *no* prefix to the option name; for example, to suppress the creation of the cross reference file you code:

```
nrc -run -nocrossref Factor
```

Note that the options are case insensitive.

Table 3 (Page 1 of 2). Compiler Options of the Options Keyword

	Default Value	Description	More
BINARY	nobinary	This option lets the programmer specify that all classes will be treated as binary classes.	*
CROSSREF	crossref	When this option is specified, NetRexxC generates a file containing a cross-reference listing for the variables, organized by class. This file name has the extension of *.crossref*.	
DIAG	nodiag	This option displays diagnostic information. (It acts like the DEBUG option in traditional programs, and it can also have side-effects!)	
FORMAT	noformat	This option adds spaces and new line characters to the generated Java source to make it more readable. (Be careful, this option does not preserve the line numbers of your original code, and run-time errors show incorrect line numbers.)	

LOGO	logo	This option controls the printing of the compiler logo, for example:
		`NetRexx portable processor, version 1.00`
		`Copyright (c) IBM Corporation, 1997.`
		`All rights reserved.`	

REPLACE	noreplace	If there is an existing result file with the extension of .java, this option enables NetRexxC to overwrite it.	*
STRICTARGS	nostrictargs	This option enforces the style rule that you always have to use parentheses for method invocations. The option is highly recommended for readability.	
STRICTASSIGN	nostrictassign	This options checks that the type of assignments (using the = operator) and the arguments passed in method invocations match exactly. This checking is stronger than the Java requirements.	*
STRICTCASE	nostrictcase	When this option is specified, the case of all names used in the NetRexx code and in references to Java classes have to match. STRICTCASE enables the enforcement of Java-like rules.	
STRICTSIGNAL	nostrictsignal	This options lets the NetRexx compiler complain if exceptions are omitted from the signal list. STRICTSIGNAL is also recommended in pursuing a good programming style.	*

Table 3 (Page 2 of 2). Compiler Options of the Options Keyword			
	Default Value	**Description**	**More**
TRACE	trace	This option lets you disable all trace instructions in a NetRexx program (by specifying NOTRACE).	
UTF8	noutf8	If this option is used, the source code is processed as being UTF-8 encoded. UTF-8 is an encoding of unicode characters; consult the specialized literature for more information.	
VERBOSE[n]	verbose3	This options specifies the number of messages you see when NetRexxC is executing. The range for *n* is from 0 to 5. VERBOSE equals VERBOSE[3] and NOVERBOSE equals VERBOSE[0]. In the latter case, you still receive all error and warning messages, but the logo is not shown.	

Here are some general remarks about the OPTIONS keyword and the way in which you specify options as arguments to the compiler:

- If you specify (accidentally or on purpose) some options twice, the last option is used. For example, if you disable an option on the command line when calling the compiler (-nologo), the option in your source code (OPTIONS LOGO) is used, because it is encountered after the command line arguments (and the logo will be printed).
- Watch out for "typos." A mistyped option in your NetRexx code is (silently) ignored because it might be of use in subsequent releases of the compiler. However, a mistyped option in the command line produces this error:

```
nrc factor -nolgo
 +++ Error: Unknown command option '-nolgo'
```

More Details on Options

Some options require further clarification.

BINARY: The *BINARY* option changes all classes in the specified programs to binary classes. This change has two consequences:

- All literals are of either a *primitive* type (*Boolean, char, byte, short, int, long, float, double*) or the Java String type.
- Where appropriate, operations are implemented as binary, instead of their default REXX variants.

Be aware of the possible computational errors you get when using these binary classes. Overflows, underflows, truncation, and other traditional digital calculation "goodies" from which the NetRexx user is usually relieved are the price to pay for the performance boost. This is a trade-off you have to make.

REPLACE: The *REPLACE* option defaults to NOREPLACE to prevent you from accidentally overwriting an existing (valuable) Java file with the same name. Therefore, use this option with care!

STRICTASSIGN: The *STRICTASSIGN* option enables a very strict checking of your code for all sorts of *hidden* conversions. For example, the source code in Figure 7 compiles fine without the STRICTASSIGN option.

```
/* compiler\StrictassignTst.nrx

   This NetRexx program demonstrates the STRICTASSIGN option */

OPTIONS strictassign

testB = B()
StrictassignTst.print(testB)
exit 0

method print(anA=A) static
      say anA.getSomeProperty()

class A
   someProperty='This a dummy-property'

   method getSomeProperty()
     return someProperty

class B extends A
```

Figure 7. STRICTASSIGN Test Program: StrictassignTst.nrx

With the STRICTASSIGN option, the compiler produces two errors:

```
NetRexx portable processor, version 1.00
Copyright (c) IBM Corporation, 1997.  All rights reserved.
 10 +++ StrictassignTst.print(testB)
    +++                 ^^^^^
    +++ Error: Cannot find method 'strictassigntst.print(B)'
 11 +++ exit 0
    +++      ^
    +++ Error: EXIT needs an integer result (expression result type is 'byte')
    function print(A)
```

As you can see, the compiler is very strict. The idea of STRICTASSIGN is to guarantee that no *under the cover* conversion costs are incurred. This might be of help when tuning some time-critical code.

STRICTSIGNAL: The *STRICTSIGNAL* option endorses good programming style by forcing you to have a *signal list* (a list of exception types that can be thrown) for every method. (This is the default behavior for Java programs.) If this option is not used and you do not code every possibly thrown exception in the signal list of all your methods, the NetRexx compiler automatically adjust the lists, to be compliant with the Java language. It is, however, good programming practice to let the users of your NetRexx classes know which method can throw which exception!

Chapter 4. The NetRexx Language

In this chapter we present a comprehensive introduction to the NetRexx language. We cover the most interesting aspects of the language. Please refer to the NetRexx documentation for further details.

Case Sensitivity

NetRexx is a case insensitive language. There is no difference between a variable named *fred* or *Fred*.

Even if NetRexx is case insensitive, it is case preserving. Thus any class, variable, or method is used in the same way in which it was defined the first time. This is important when using NetRexx classes from Java, because Java is very case sensitive.

When using Java classes and methods in NetRexx programs, you do not have to enter the names in the exact Java spelling; NetRexx will attempt to find the class or method anyway. However, we recommend using the exact Java spelling so that users can easily understand your NetRexx programs.

Comments

NetRexx supports two types of comments:

- A standard Java or Rexx style comment that begins with /* and continues until its matching */. Comments can be nested with matching pairs of /* and */.
- A full-line or partial-line comment begins with a double hyphen (--) and continues until the end of the line. The double hyphen can appear anywhere within a NetRexx statement.

Here are two examples of NetRexx comments:

```
/* This is NetRexx comment 1
   /* This is a nested NetRexx comment */
   end of comment 1*/

-- A single line comment
say 'Hi redbook reader'    -- a partial line comment
```

Continuation Character

Each NetRexx statement ends with the end of the line or at a semicolon (;).

If a statement is too long for a line, use a hyphen (-) at the end of the line for continuation:

```
say 'This is a long text to be' -
    'displayed to the user'
```

Input and Output

Programs without a graphical user interface can use the NetRexx instructions for data input and output. The say instruction (Figure 8) is used to write to the default character output stream.

```
say [expression]
```

Figure 8. Say Instruction

The result of the expression is expected to be a string, or it will be converted to a string. By default, the result string is treated as a line, so line termination characters are appended. If the string ends in a null character ("\-" or "\0"), line termination is skipped.

The *ask* keyword reads a line from the default input stream.

Examples of using *say* and *ask* are:

```
say 'Enter any string: \-'
a = ask
if a = 'any string' then say 'Thank you!'
                    else say 'Hey, I said "any string"'
```

Data Types

Programs written in the NetRexx language manipulate values such as character strings and numbers. All such values have an associated type.

NetRexx basically uses only one type for expressions, a NetRexx string (see "The Rexx Class for Strings" on page 30). A NetRexx string is any group of characters inside single or double quotation marks.

Examples of NetRexx strings are:

```
"This is a NetRexx string"
'This is a "NetRexx" string too'
```

All primitive Java data types are available in NetRexx (see Table 4).

Table 4. Primitive Java Data Types				
	Contains	**Size (bits)**	**Mininum Value**	**Maximum Value**
boolean	1 or 0			
char	Unicode character	16		
byte	Signed integer	8	-128	127
short	Signed integer	16	-32768	32767
int	Signed integer	32	-2147483648	2147483647
long	Signed integer	64	-9223372036854775808	9223372036854775807
float	Floating point	32	±1.40239846E-45	±3.40282347E+38
double	Floating point	64	±4.94065645841246544E-324	±1.79769313486231570E+308

NetRexx automatically converts data types when possible.

For constants NetRexx uses the smallest type possible:

```
  4  would be of type byte
257  would be of type integer
'a'  would be of type character
```

If the binary option is not specified (see “Options” on page 47), or the type of a variable is not explicitly set, NetRexx converts every primitive data type to a NetRexx string before an expression is evaluated.

If a NetRexx string is assigned to a variable of a different type, an automatic conversion occurs. Automatic conversion is more reliable than dealing directly with the primitive types, because NetRexx strings have their own numeric algorithm (see “The Rexx Class for Strings” on page 30).

The following example shows how NetRexx behaves when an overflow occurs:

```
/* Overflow */
numeric digits 20    -- adjust the precision for rounding
x = int 2147483647   -- maximum integer
y = int

say "x  =" x         -- show x
say "x+1=" x+1       -- show x+1 (calculated using Rexx class)
y=x+1                -- too big for an integer
say "y  =" y         -- not possible ==> overflow
```

When you run this code, the program stops and reports an error:

```
x  = 2147483647
x+1= 2147483648
java.lang.NumberFormatException: Conversion overflow
       at netrexx.lang.Rexx.toint(Compiled Code)
       at overflow.main(Compiled Code)
```

If you run a similar program as a native Java program, the program shows incorrect results, and no error comes up:

```
x  = 2147483647
x+1= -2147483648
y  = -2147483648
```

Operators and Expressions

NetRexx handles four different expressions: string, arithmetic, comparative, and logical.

Expressions are evaluated from the left to the right, modified by parentheses and operator precedence.

String Expressions

The concatenation operators combine two strings, by appending the first string to the right side of the second string. The concatenation can occur with or without an interleaving blank:

(blank) Concatenate with an interleaving blank:

```
"book" "store"     -> "book store"
```

|| Concatenate without an interleaving blank:

```
"book"||"store"    -> "bookstore"
"book" || "store"  -> "bookstore"
```

(abuttal) Concatenate a variable with a literal string without an interleaving blank:

```
abc = "book"
abc"store"         -> "bookstore"
```

Arithmetic Expressions

Character strings that are numbers and Java primitive type variables can be combined with these arithmetic operators:

\+ Add

\− Subtract

***** Multiply

/ Divide. If the remainder of the division is not 0, the result is a floating point number.

% Divide and return the integer part of the result (integer division). The arguments do not have to be integers.

// Divide and return the remainder of the division. This is not the same as modulo, because the result can be negative. If one operand is a floating point number, the remainder can be a floating point number too.

This operation is equivalent to: a − b * (a%b)

****** Power. Raise a number to a whole number power.

Prefix − Is equivalent to: 0 − number

Prefix + Is equivalent to: 0 + number

Comparative Expressions

Comparative operators compare two terms and return 0 (false) or 1 (true). Two sets of comparative operators are available.

Normal Comparative Operators

The rules for a normal comparison are:

- The comparison is not case sensitive.
- Leading and trailing blanks are removed before comparison.
- If one string is shorter than the other, it is padded with blanks on the right.

The operators for a normal comparison are:

=	Equal ('FRED' = ' fred ')
\= or <> or ><	Not equal
>	Greater than
<	Less than
>= or \<	Greater or equal
<= or \>	Less or equal

Strict Comparative Operators

The rules for a strict comparison are:

- The comparison is case sensitive.
- If two strings are equal, except that one string is shorter, the shorter string is less than the longer string.

The operators for a strict comparison are:

= =	Equal
\ = =	Not equal
> >	Greater than
< <	Less than ('fred' << 'fred ')
>>= or \<<	Greater or equal
<<= or \>>	Less or equal

Logical Expressions

Logical operators can be applied to character strings with value 0 (false) or 1 (true) or to boolean variables:

&	And, returns 1 if both terms are true
|	Or, returns 1 if one or both terms are true
&&	Exclusive or, returns true if only one term is true
**Prefix **	Logical not

Variables

A variable is a named object whose value may change during the execution of a NetRexx program.

A variable is defined by an assignment:

```
fred = 'Fred Firestone'   -- assigns the value 'Fred Firestone' to
                          -- the variable fred
```

A variable name is case insensitive, cannot begin with a digit, and does not contain a period.

Each variable has an associated type, which cannot change during execution. The type of a variable is determined by the type of the result value of the expression that is first assigned to it:

```
fred  = 'Fred Firestone' -- fred  has type Rexx
count = 5                -- count has type Rexx
max   = 3.56             -- max   has type Rexx
obj   = Cache()          -- obj   has type Cache
```

The last example invokes the constructor of the Java cache class to construct a new object (see Chapter 6, “Creating and Using NetRexx Classes” on page 57).

A variable can be declared by simply assigning a type to it:

```
fred   = Rexx        -- fred is a Rexx string
count  = int         -- count is an int (32 bit integer)
window = Frame       -- window is a Frame
```

If a variable is declared with a type only, it will be initialized to a default value depending on the type of the variable:

Type	Default Value
boolean	0 (false)
char	character with the decimal expression of 0
byte	0
short	0
int	0
long	0
float	0
double	0
String	null
Rexx	null

If the type of the variable is a Java class, the default value is null. The variable is a reference to an object of the class. The null value is a special value which indicates that there is no reference yet.

Class Definition

A class definition (Figure 9) consists of a class instruction and optional property and method definitions (see also Chapter 6, “Creating and Using NetRexx Classes” on page 57).

```
class instruction
  property definition
    .
    .

  method definition
    .
    .
```

Figure 9. Class Definition

A class definition does not need to have property or method definitions.

Class Instruction

The class instruction (Figure 10) is used to define a class as described in Chapter 6, “Creating and Using NetRexx Classes” on page 57.

```
class name [public | private]
       [abstract | final | interface]
       [binary]
       [extends classname]
       [uses classname [,classname]...]
       [implements interfacename [,interfacename]...]
```

Figure 10. Class Instruction

The options can appear in any order.

If more than one class is defined in a file, only the first class in the file can be public.

The public class must have the same name as the file (without the file extension).

If none of the abstract, final, or interface keywords is used, objects can be created from the class, and the class can be subclassed.

Options

public The class can be used by all other classes (default for the first class in a file).

private The class can be used by the classes of the same file or by the classes of the same package.

abstract The class is not completely implemented. No objects can be created from this class (see “Abstract Classes” on page 66).

final The developer considered that the class is complete, and it cannot be subclassed. This option may allow the compiler to improve the performance of classes that refer to a final class. This option reduces the reusability of a class and should be avoided.

interface The class is an interface class and cannot be used to construct an object. See “Interfaces” on page 68 for a detailed description.

binary In a binary class, strings and numeric symbols are native Java strings or primitive types and are not converted automatically to Rexx objects before an expression is evaluated. The binary option can increase the speed of the program; see “Options” on page 47.

extends The class inherits the properties and methods from the class specified. If there is no extends clause, the class inherits from the Object class by default (see “Inheritance” on page 63).

implements The class implements the interfaces defined by the listed interface classes. All methods of the specified interfaces have to be implemented.

uses The *uses* keyword introduces a list of classes that will be used as a source of constant or static properties and methods. Every class method, class property, or constant of the classes specified can be used by their name without the need to specify the class name:

```
class ConstClass
   properties constant
      PI = Rexx 3.14159265358979323846

class Example uses ConstClass
   method Example()
     say 'Pi =' PI     -- equivalent to: ConstClass.PI
```

The *uses* keyword affects only the syntax of the current class. It is not inherited by subclasses of the current class.

Properties Instruction

The properties instruction (Figure 11) is used to define the variables (properties) and their attributes of the class (see also Chapter 6, “Creating and Using NetRexx Classes” on page 57).

The following terms are used in the description of the properties instruction:

Class variable Variable (property) that belongs to the class and not to an individual object

Instance variable Variable (property) that belongs to an individual object of the class

```
properties [public | private | inheritable]
           [constant | static | volatile]
```

Figure 11. Properties Instruction

The options can appear in any order.

The properties instruction must precede the first method instruction in a class.

If no properties instruction is used, but variables are defined before the first method instruction, the *inheritable* default option applies to those variables.

A properties instruction replaces any previous properties instruction.

A properties instruction is followed by variable declarations (see “Variables” on page 24).

Options

public Public properties can be used by all other classes to which the current class is visible.

inheritable Inheritable properties can be used by classes that are in the same package or are subclasses of the current class. This is the default.

private Private properties can be used by the current class only.

constant Constant properties are constant class variables. They exist only once and cannot be changed.

static Static properties are class variables. They belong to the class and not to an individual object. Static properties can be accessed like normal variables. Static properties are initialized when the class is loaded.

volatile Volatile properties can be changed asynchronously outside the control of the interpreter.

A properties statement must be followed by a least one keyword.

Property Instruction Examples:

```
class PropertiesExample
  name
  timer    = Thread null
  elements = Vector()

  properties private
    counter = int

  properties static private
    objects = Vector()

  properties public constant
    PI = Double 3.141592653589793
```

The example defines:

name Inheritable instance variable of type Rexx

timer Inheritable instance variable of type Thread. It is initialized to null at object creation time.

elements Inheritable instance variable of type Vector. The default constructor is called at object creation time.

counter Private instance variable of type int (primitive type). The counter variable is not visible outside the class.

objects Private class variable of type Vector. The objects variable is not visible outside the class. The constructor of Vector is called when the class is loaded.

PI A public constant

Property Usage: Public and inheritable properties are used within the same class by their name; in other classes they are usually qualified with the class name:

```
area = PropertiesExample.PI * radius**2
PropertiesExample.elements.add(newobject)
```

Method Instruction

The method instruction (Figure 12) is used to define the procedures and functions of a class (see also Chapter 6, “Creating and Using NetRexx Classes” on page 57).

The following terms are used in the description of the method instruction:

Class method Method that belongs to the class and not to an individual object. It can only access constants and static variables (class variables).

Instance method
Method that belongs to an object and has access to the instance variables of the object, as well as to constants and static variables

Constructor method
Method that constructs a new object of the class. A constructor usually initializes the instance variables.

Default constructor method
Constructor method without any parameters

Superclass The class that is extended by the current class

```
method name[( [argument[,argument]...] ) ]
       [public | private | inheritable]
       [abstract | static | constant | final | native]
       [protect]
       [returns classname]
       [signals exceptionclass[,exceptionclass]...]
```

Figure 12. Method Instruction

Except for the name and the argument list, the options can appear in any order.

The method name must be a nonnumeric symbol. If the method name matches the class name, it is a constructor method.

Arguments are like variable definitions:

- If a type is not defined, the argument is of type Rexx.
- If the argument has an initializer, the argument is optional and defaults to the given value if not specified in the call of the method. Such optional arguments must be at the end of the list.

Any instruction after a method instruction is part of the method. A method ends with the next method or class instruction.

Options

public A public method can be called by all other classes to which the current class is visible. This is the default.

inheritable A inheritable method can be called by classes that are in the same package or are subclasses of the current class.

private A private method can be used by the current class only.

abstract An abstract method defines only the interface of the method. Only the name and the types of the arguments are defined. There are no instructions that implement the method. The class instruction for the class containing this method must also use the abstract keyword.

static Static methods are class methods. They belong to the class and not to an individual object.

constant A constant method is a class method that cannot be redefined in a subclass. Constant combines the static and final options.

final A final method cannot be redefined by any subclass. This option allows some performance improvements but reduces the reusability of the class. We recommend avoiding the final option.

native A native method is implemented by the environment, so instructions to implement the method are not permitted. A native method cannot be overwritten in a subclass.

protect Protected methods are automatically serialized when accessing the same object from different threads. They do not prevent the invocation of nonprotected methods of the same object in parallel. The protect option is similar to a “do protect this” as the first instruction of the method.

returns The method returns an object of the class specified. This is similar to classic Rexx functions.

Note: The returns option is not necessary if the method returns a Rexx object. However, this is not a recommended programming style, because the user of the method has to read the implementation to figure out whether a value is returned.

signals The signals option lists all exceptions that can be thrown by the method.

Method Instruction Examples:

```
class Example
  properties inheritable    -- default option, but good style
    a
    b = Vector()

  method asString()         -- method without any parameter
    say a

  method asString( prefix ) -- overwrites above method
    say prefix a            -- different signature

  method asString( prefix, postfix, size = int 40 )
                            -- method overwrites the two methods above
                            -- size is an optional parameter
    say (prefix a postfix).right(size)

  method addNext() private  -- the parenthesis are not necessary
    b.addElement(Date())
```

Special Keywords Used in Methods

In the implementation of methods the *this* and *super* keywords refer to:

this the current object (instance methods)

super the class from which the current class inherits

Exceptions

For advanced error handling NetRexx uses the concept of exceptions. When an unexpected condition occurs, such as the end of a file, an exception is signaled (thrown). A signaled exception can be caught by a catch statement (see “Do Instruction” on page 40 or “Repetitive Tasks” on page 43).

An exception breaks the flow of the program and jumps directly to a catch statement that handles the exception.

The benefits of exceptions are that the main code can be written without any error handling, and it is easy and understandable. The error handling is done by the catch statements.

If an exception is not handled in the current method, NetRexx looks for a catch statement in the calling method. NetRexx follows the call path until it finds a catch statement that handles the exception.

An exception is thrown explicitly by the signal statement (see Figure 13) or automatically by external conditions.

```
signal exceptionterm
```

Figure 13. Signal Instruction

The *exceptionterm* is an object of type Exception or any subclass of Exception, or just the name of an exception type. If an exception type is specified, NetRexx constructs an exception object with the default constructor of the class.

The Rexx Class for Strings

The Rexx class is one of the basics of NetRexx. The class provides extended string handling compared with Java. The methods of the Rexx class are well known from the classic Rexx language.

A Rexx string is changed only at its creation. Every method or action that would change the string creates a new object. There is only one exception to this rule: Assigning subvalues (see “Indexed Strings” on page 38) to a string does not create a new object.

Parsing a String

A Rexx string can be manipulated with the methods of the Rexx class or parsed with the parse instruction (Figure 14). The parse instruction is one of the most powerful instructions of NetRexx for string manipulation.

```
parse stringExpression template
```

Figure 14. Parse Instruction

stringExpression is any expression with the value of type Rexx (string).

template is more complicated. Technically it consists of variable names, literal strings, numbers, and symbols. In the next paragraphs we show you how to use templates. If you are a experienced Rexx user, you will see that there is no big difference between NetRexx and the classic Rexx equivalent. See the *NetRexx Online Documentation* for more detailed information.

To split a string into words, the template consists of a list of variables. If there are more words than variables, the last variable of the list contains the rest of the string:

```
parse 'NetRexx is an interesting language ' v1 v2 v3

-> Results:  v1 : 'NetRexx'
             v2 : 'is'
             v3 : 'an interesting language '
```

The resulting variables do not contain leading or trailing blanks, except for the last variable of the list.

If you are not interested in one of the words, use a period instead of a variable:

```
parse 'NetRexx is an interesting language ' v1 . v2

-> Results:  v1 : 'NetRexx'
             v2 : 'an interesting language '
```

The second parsing mechanism uses literal strings as a pattern to split up the string:

```
parse 'NetRexx is an interesting language ' v1 'an' v2

-> Results:  v1 : 'NetRexx is '
             v2 : ' interesting language '
```

As you can see, parse no longer separates words delimited by spaces, and v1 and v2 include trailing and leading blanks.

The pattern must not be a literal string; it can be a variable:

```
pattern = 'interesting'
parse 'NetRexx is an interesting language ' v1 . v2 (pattern) v3 .

-> Results:  v1 : 'NetRexx'
             v2 : 'an '
             v3 : ' language'
```

Different methods for positional parsing are available. Please refer to the *NetRexx Online Documentation* for more information.

Built-In Methods

This description of *built-in* methods is brief. Please refer to the *NetRexx Online Documentation* for more details.

Some notes to built-in methods:

- All methods arguments are of type Rexx.
- All results returned are of type Rexx.
- The position of the first character in a string is 1.
- A pad argument, if specified, must be exactly one character long.
- The term *string* in the syntax examples stands for the current Rexx object.

Abbrev

Syntax: *string*.abbrev(info[,length])

Returns 1 if *info* is a leading substring of *string*. *Length* specifies the number of characters that are compared. The comparison is case sensitive.

Abs

Syntax: *string*.abs()

Returns the absolute value of *string*, which must be a number.

B2x

Syntax: *string*.b2x()

Converts the binary *string* to hexadecimal; it must consist of digits 0 or 1.

Center

Syntax: *string*.center(length[,pad])

Returns a string of length *length* with *string* centered in it; the *pad* character is used to fill the string and defaults to blank.

Changestr

Syntax: *string*.changestr(needle,new)

Returns a string where each occurrence of *needle* in *string* is replaced by *new*.

Compare

Syntax: *string*.compare(target[,pad])

Returns 0 if *target* and *string* are the same. If *target* is shorter than *string* it is filled with *pad*, which defaults to blank. The comparison is case sensitive.

Copies

Syntax: *string*.copies(n)

Returns *n* directly concatenated copies of *string*.

Countstr

Syntax: *string*.countstr(needle)

Returns the count of nonoverlapping occurrences of *needle* in *string*.

C2d

Syntax: *string*.c2d()

Returns the decimal value of the character representation of *string*, which must be one character long.

C2x

Syntax: *string*.c2x()

Returns the hexadecimal value of the character representation of *string*, which must be one character long.

Datatype

Syntax: *string*.datatype(option)

Returns 1 if *string* matches the description requested by *option*, 0 otherwise. Only the first character of *option* is used and may be specified in uppercase or lowercase.

Following *option* characters are recognized:

A (alphanumeric)	The *string* contains characters only from the ranges a – z, A – Z, and 0–9.
B (binary)	The *string* contains only characters 0 and 1.
D (digits)	The *string* contains characters only from the 0–9 range.
L (lower case)	The *string* contains characters only from the range a – z.
M (mixed case)	The *string* contains characters only from ranges a – z and A – Z.
N (number)	The *string* is a valid number.
S (symbol)	The *string* contains characters only from ranges a – z, A – Z, and 0–9 or an underscore (_) and does not begin with a digit.
U (upper case)	The *string* contains characters only from the range A – Z.
W (whole number)	The *string* is a whole number (integer).
X (hexadecimal)	The *string* contains characters only from ranges a – f, A – F, and 0–9.

Delstr

Syntax: *string*.delstr(n[,length])

Returns a string with *length* characters of *string* deleted, starting at position *n*. *Length* defaults to the rest of *string*.

Delword

Syntax: *string*.delword(n[,length])

Returns a string with *length* words of *string* deleted, starting at word *n*. *Length* defaults to the remaining words of *string*.

D2c

Syntax: *string*.d2c()

Converts *string* to a single character. *string* must be a whole number and represent a valid character.

D2x

Syntax: *string*.d2x([n])

Converts *string* to a hexadecimal representation. *string* must be a signed integer. The resulting string is padded on the left with zeros to the length *n* (if specified).

Exists

Syntax: *string*.exists(index)

See “Indexed Strings” on page 38.

Format

Syntax: *string*.format([before[,after[,explaces[,exdigits[,exform]]]]])

Formats *string*, which must be a number. For the definition of *before*, *after*, and *explaces*, see Figure 15. If a parameter is skipped (set to *null*), the default is used.

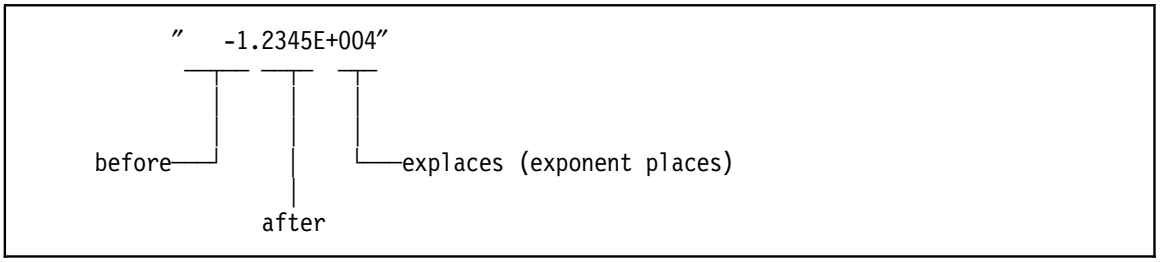


Figure 15. Options of the Format Built-in Method

exdigits sets the trigger point for the usage of the exponential form. *exform* defines the form of exponential notation, either “Scientific” (S) or “Engineering” (E).

Insert

Syntax: *string*.insert(new[,n[,length[,pad]]])

Returns a string where *new* is inserted, padded or truncated to length *length*, at position *n* into *string*.

Lastpos

Syntax: *string*.lastpos(needle[,start])

Returns the position of the last occurrence of the string *needle* in *string* starting from position *start*. Returns 0 if *needle* was not found.

Left

Syntax: *string*.left(length[,pad])

Returns a string of length *length* that contains the left-most characters of *string*. If necessary, the string is padded at the end with the optional character *pad*, which defaults to blank.

Length

Syntax: *string*.length()

Returns the length of *string*.

Lower

Syntax: *string*.lower([n[,length]])

Returns a string where all characters of *string*, starting from position *n* for *length* characters, are changed to lowercase.

Max

Syntax: *string*.max(number)

Returns the larger of *string* and *number*, both of which must be valid numbers.

Min

Syntax: *string*.min(number)

Returns the smaller of *string* and *number*, both of which must be valid numbers.

Overlay

Syntax: *string*.overlay(new[,n[,length[,pad]]])

Returns a string, where *new*, padded or truncated to length *length*, overlays any character of *string* starting from position *n*. *n* defaults to the beginning of *string*, *length* to the length of *new*, and *pad* to blank.

Pos

Syntax: *string*.pos(needle[,n])

Returns the position of the *needle* in *string*, starting at position *n*.

Reverse

Syntax: *string*.reverse()

Returns a string where *string* is swapped end for end.

Right

Syntax: right(length[,pad])

Returns a string of length *length* that contains the right-most characters of *string*. If necessary, the string is padded at the beginning with the optional character *pad*, which defaults to blank.

Sequence

Syntax: *string*.sequence(final)

Returns a string of all characters, in ascending order of encoding, between *string* and *final*, including *string* and *final*, both of which are single characters.

Sign

Syntax: *string*.sign()

Returns a number that indicates the sign of *string*, which must be a valid number. The result is −1 if *string* is negative, 0 if *string* is equal to 0, and 1 if *string* is positive.

Space

Syntax: *string*.space([n[,pad]])

Returns a string where the words in *string* are formatted with *n pad* characters between each word. Leading and trailing blanks are removed. The default for *n* is one; the default for *pad* is blank.

Strip

Syntax: *string*.strip([option[,char]])

Returns a string, where all leading, trailing, or leading and trailing characters *char* of *string* are removed. The *option* specifies *L* for leading, *T* for trailing, or *B* for both (default), and *char* defaults to blank.

Substr

Syntax: *string*.substr(n[,length[,pad]])

Returns a substring of *string*, starting at position *n* with the length *length*, padded with *pad* if necessary.

Subword

Syntax: *string*.subword(n[,length])

Returns a substring of *string* starting with the n^{th} word and up to *length* blank delimited words long.

Translate

Syntax: *string*.translate(oTable,iTable[,pad])

Returns a copy of *string*, where every character in *string* that is found in input table *iTable* is replaced by the matching character (at the same position) of output table *oTable*. The output table is truncated or padded with *pad* to the length of the input table.

Trunc

Syntax: *string*.trunc([n])

Returns a truncated number with *n* decimals. *string* is not rounded and *n* defaults to 0, which simply returns the integer part of *string*.

Upper

Syntax: *string*.upper([n[,length]])

Returns a string where all characters of *string*, starting from position *n* for *length* characters, are changed to uppercase.

Verify

Syntax: *string*.verify(reference[,option[,start]])

Returns the position of the first character of *string*, starting at position *start*, which is not listed in *reference* when *option* is set to *Nomatch* (default). If option is set to *Match*, the position of the first character that is included in *reference* is returned. Returns 0 if no match is found.

Word

Syntax: *string*.word(n)

Returns the n^{th} word of *string*.

Wordindex

Syntax: *string*.wordindex(n)

Returns the character position of the n^{th} word of *string*.

Wordlength

Syntax: *string*.wordlength(string,n)

Returns the length of the n^{th} word in *string*.

Wordpos

Syntax: *string*.wordpos(phrase[,start])

Returns the word number of the first word of *phrase* in *string* starting at word number *start*. Multiple blanks in phrase or *string* are treated as one blank. Returns 0 if no match is found.

Words

Syntax: *string*.words()

Returns the number of words in *string*.

X2b

Syntax: *string*.x2b()

Returns the binary equivalent of *string*, which must be hexadecimal, that is, *string*.datatype('H') must be true.

X2c

Syntax: *string*.x2c()

Returns a single character, the hexadecimal encoding of the single character in *string*.

X2d

Syntax: *string*.x2d([n])

Returns the decimal equivalent of *string*, which must be hexadecimal, that is, *string*.datatype('H') must be true.

Indexed Strings

A NetRexx string that has subvalues is called an indexed string. This function is similar to *stems* in the Rexx language. A subvalue of a NetRexx string is a NetRexx string. Any other type is not supported.

The subvalue of a string is accessed by using square brackets. The opening square bracket must immediately follow the variable name without any interleaving blanks:

```
stringname[expressions]
```

The *expressions*, separated by commas, are called the indexes of the string. Any expression must be a NetRexx string or can be converted to a NetRexx string.

The nonindexed value of the string must have been assigned before indexing is used on it. The nonindexed value is used for a reference to a nonexisting subvalue.

Indexed String Example:

```
phone = 'Sorry, unknown name'   -- default value if name is unknown
phone['Alex'] = '234-4345'      -- set some initial indexed values
phone['Fred'] = '254-2345'      -- phone implements a phone book
phone['Elsa'] = '234-9578'

search='Elsa'
say phone[search]               -- says 234-9578
say phone['Alex']               -- says 234-4345
say phone['Peter']              -- says Sorry, unknown name
```

When multiple indexes, separated by commas, are used, they indicate a hierarchy of strings. A single NetRexx string has a set of indexes and subvalues. The subvalues are also NetRexx strings that may have indexes and subvalues.

Multiple Index Example:

```
x = "?"
x['foo'] = 'Yes'     -- sets the indexed value of x['foo']
x['foo','bar']='Ok'  -- sets the multiple indexed value
say x['foo','bar']   -- Ok

y = x['foo']         -- Returns a string set to 'Yes' with indexed
                     -- value ['bar']
say y                -- Yes
say y['bar']         -- Ok
say y['br']          -- Yes
say x['br']          -- ?
```

Use the *exists(subvalue)* method to determine whether a subvalue exists. The method returns 1 (true) if the subvalue exists, and 0 (false) if not.

Assign null to an indexed reference to drop a subvalue.

The indexes can be retrieved in turn using the over keyword of the loop instruction (see "Repetitive Tasks" on page 43).

Note: The default value of an indexed string cannot be changed. If you assign a new string to the variable, the reference of the variable points to the new string that has no subvalues! This is true even for subvalues that are indexed strings.

Use a hash table (of the *java.util* package) if you need to store objects of a type different from NetRexx strings.

Arrays

In addition to indexed strings, NetRexx also includes the concept of fixed-size arrays, which may be used for indexing values of any type.

Arrays are used in a way similar to indexed strings, but with some important differences:

- An array changes the type of a value to *dimensioned*.
- The index of an array is of type int and starts at 0.
- An array is of fixed size. It must be constructed before use.

The following examples illustrate that an array of Rexx strings is not the same as an indexed string.

Array Constructor:

```
a = String[4]        -- makes an array of 4 Java strings
b = Rexx[7]          -- makes an array of 7 Rexx strings
c = int[10,10]       -- makes an 10×10 array of integer
```

Array Length: An array has a length variable that reflects the size of the array:

```
a = String[4]                -- makes an array of 4 Java strings
say 'Size of a =' a.length   -- says 'Size of a = 4'
```

Array Type Declaration: The type of a variable can be set with an array notation that indicates the dimension of an array without any size:

```
d = int[]       -- one dimensional array of int with any size
e = int[,]      -- two dimensional array of int with any size
```

This notation is very useful when defining method parameters that deal with arrays of unknown size.

Vector: Arrays have a fixed size. If your application has to use dynamically growing arrays, you can use the Vector class (of the *java.util* package) instead.

Control Statements

NetRexx has a few very powerful instructions that enable you to control the program flow.

In a variation to the classic Rexx definition, NetRexx has one instruction for creating groups of instructions, and another instruction for repetitive tasks. In contrast to Java, NetRexx combines its control structures with the exception handling and locking mechanism. As a result programs are well structured and have extended readability.

Do Instruction

The do instruction defines a code block. A code block is a group of instructions. It may have a label, and it may protect an object while the instructions are executed. Exceptions inside the block can be handled. Figure 16 shows the syntax of a code block.

```
do [label labelname] [protect object]
  instruction
    .
    .
[catch [varexp = ] ExceptionClass
  instruction
    .
    .
]...
[finally
  instruction
    .
    .
]
end [labelname]
```

Figure 16. Do Instruction

Options

label The label phrase is used to specify a name for the group. The name can be used with the leave instruction (see “Exit a Control Structure” on page 46) or for better readability of the code if many groups are nested. Extending the readability requires the use of the label name in the end phrase, of course.

protect The protect phrase provides exclusive control over any *object*. The instructions of the do statement cannot be interrupted by another thread that uses the same object. *object* is any expression that results in an object and not in a primitive type.

catch Exceptions thrown by any instruction in the do group may be caught by using one or more catch clauses. Exceptions that are not caught cause the immediate end of the current method (see “Exceptions” on page 30). If an exception is caught, the execution continues with the next statement after the do group (except the finally clause).

finally Instructions in the finally clause are always executed, even if the do block is terminated by an exception, leave, or return instruction. If the exception is caught by a catch clause of the do block, the instructions of the finally clause are executed after the instructions in the catch clause.

Do Instruction Example:

```
do label bigLoop
  i = int
  say 'which number ?'
  do label getInput
    i=ask                  -- gets the input
  catch RuntimeException
    say "This was not a number"
    say "0 is used as default"
    i = 0
  finally
    say "number =" i       -- executed, regardless what happens
  end getInput
  say '5x'i'='5*i
end bigLoop
```

Conditional Instructions

NetRexx has two conditional instructions—if and select—that alter the program flow according to boolean expressions.

If Instruction

The if instruction (Figure 17) is used to conditionally execute one instruction or to select between two alternatives.

```
if booleanExpression then instruction
[else instruction]
```

Figure 17. If Instruction

The *then* and the *else* keyword implicitly insert a semicolon. The implicit semicolon enables you to write the instruction on a separate line or immediately after the *then* or *else* keyword.

A code block must be used if more than one instruction is executed after the *then* or *else* keyword.

The *else* keyword binds to the previous *then* at the same level, when nested if instructions are used.

If Instruction Example:

```
if answer='yes' then say 'fine'
  else                    -- possible because of implicitly semicolon
     say 'Why not'

if a > 4 then
  if b < 2 then
     say 'a is greater 4 and b less 2'
  else
     say 'a is greater 4 and b greater or equal 2'

else do             -- only one instruction allowed -> use do group
  say 'a is less or equal 4'
  say "Don't know anything about b"
end
```

Select Instruction

The select instruction (Figure 18) is used to conditionally execute one or several alternatives. The construct may optionally be given a label, protect an object while the instruction is executed, or catch exceptions.

```
select [label labelname] [protect object]
 when booleanExpression then instruction
[when booleanExpression then instruction]...
[otherwise
  instruction
    .
    .
]
[catch [varexp = ] ExceptionClass
  instruction
    .
    .
]...
[finally
  instruction
    .
    .
]
end [labelname]
```

Figure 18. Select Instruction

Each boolean expression following a *when* is evaluated from top to bottom. If the expression evaluates to 1 (true), the instruction following the *then* is executed and control passes to the *finally* clause. If the *finally* clause is absent, control passes directly to *end*.

If none of the when instructions results in 1, control passes to the *otherwise* instruction. If the *otherwise* instruction is absent, a NoOtherwiseException is thrown.

Options

label The label phrase is used to specify a name for the select statement. The name can be used with the leave instruction (see “Exit a Control Structure” on page 46) or for better readability of the code. Extending the readability requires the use of the label name in the end phrase, of course.

protect The protect phrase provides exclusive control over any *object*. The instructions of the select statement cannot be interrupted by another thread that uses the same object. *object* is any expression that results in an object and not in a primitive type.

catch Exceptions thrown by any instruction in the select statement may be caught using one or more catch clauses. Exceptions that are not caught cause the immediate end of the current method (see “Exceptions” on page 30). If an exception is caught, the execution continues with the next statement after the select instruction (except the finally clause).

finally Instructions in the finally clause are always executed, even if the select instruction is terminated by an exception, leave, or return instruction. If the exception is caught by a catch clause of the select group, the instructions of the finally clause are executed after the instructions in the catch clause.

Select Instruction Example:

```
select
  when x * y < 2000 then say 'You have a small house'
  when x * y < 6000 then say 'You have a large house'
  otherwise             say 'You have a villa'
end
```

Repetitive Tasks

NetRexx has only one instruction for all types of repetitive tasks, the loop instruction.

Loop Instruction

The loop instruction is the most complicated of the NetRexx instructions (Figure 19).

```
loop [label labelname] [protect object] [repetitor] [conditional]
  instruction
     .
     .
 [catch [varexp = ] ExceptionClass
   instruction
     .
     .
 ]...
 [finally
   instruction
     .
     .
 ]
end [labelname]
```

Figure 19. Loop Instruction

The *repetitor* is one of:

- *controlvar* = *iexpression* [to *eexpression*] [by *bexpression*] [for *fexpression*]
- *overvar* over *oterm*
- for *fexpression*
- forever

The *conditional* is either of:

- while *booleanExpression*
- until *booleanExpression*

The instructions after the loop statement are called the *body* of the loop.

Options

label The label phrase is used to specify a name for the loop. The name can be used with the leave instruction (see “Exit a Control Structure” on page 46) or for better readability of the code. Extending the readability requires the use of the label name in the end phrase, of course.

protect The protect phrase provides exclusive control over any *object*. The instructions of the loop cannot be interrupted by another thread that uses the same object. *object* is any expression that results in an object and not in a primitive type.

catch Exceptions thrown by any instruction in the loop may be caught using one or more catch clauses. Exceptions that are not caught cause the immediate end of the current method (see “Exceptions” on page 30). If an exception is caught, the execution continues with the next statement after the loop (except the finally clause).

finally Instructions in the finally clause are always executed, even if the loop is terminated by an exception, leave, or return instruction. If the exception is caught by a catch clause of the loop, the instructions of the finally clause are executed after the instructions in the catch clause.

Indefinite Loops

A indefinite loop is constructed with the *forever* repetitor. The loop ends only when an instruction in the body of the loop causes control to leave the loop.

Example of an indefinite loop:

```
loop forever
  sleep(1000)               -- loop body begins here, wait 1 second
  repaint()                 -- draw something
catch InterruptedException  -- one way to leave the loop
end
```

Bounded Loops

A bounded loop is constructed with the *for* repetitor. The loop is repeated as many times as the *fexpression* specifies. There is no variable that controls how often the loop is repeated. *fexpression* must be a positive integer.

Example of a bounded loop:

```
-- This example displays 7 times  I'm still alive
loop for 7
  say "I'm still alive"
end
```

Controlled Bounded Loops

A controlled bounded loop starts with an assignment to a control variable, *controlvar*. The control variable can be an existing numeric variable in the current method or class or a new variable.

The *by* keyword specifies how the control variable is incremented or decremented after each loop iteration; it defaults to +1.

If the *to* keyword is specified, the loop is terminated when the control variable becomes greater than the *texpression* (or less than the expression if the by value is negative).

If the *for* keyword is used, the loop ends after *fexpression* repetitions.

Examples of controlled bounded loops:

```
loop i=1 to 7
  say i
end
-- displays 1 2 3 4 5 6 7

loop i=1.0 to 4.2 by 0.9
  say i
end
-- displays 1.0 1.9 2.8 3.7

loop i=4 to 3
  say i
end
-- displays nothing because the loop body is never executed

loop i=4.0 by 0.8 for 3
  say i
```

```
end
-- displays 4.0 4.8 5.6
```

Over Loops

An over loop iterates over an indexed string or a hash table (java.util package). The loop takes a snapshot of the indexes in the collection at the start of the loop. For each iteration of the loop, the control variable is set to an index from the snapshot.

The order in which the values are returned is undefined.

Example of an over loop:

```
phone = 'Who ?'
phone['Pete']='2342'; phone['Mary']='3943'; phone['Mike'] = 4643
loop name over phone    -- control variable is name, loops over phone
  say name 'has phone number' phone[name]
end
/* shows
  Pete has phone number 2342
  Mary has phone number 3943
  Mike has phone number 4643 */
```

Conditional Loops

Loops with the *until* or *while* keyword are conditional loops.

A while loop checks the expression before the body of the loop is executed. The loop is executed while the expression is true.

An until loop checks the expression after the body loop has been executed. The loop is repeated until the expression becomes true.

Examples of conditional loops:

```
-- the loop stops when a is greater equal 4
a = 0
loop label whileloop while a < 4
  do
    say "enter any number"
    a = a + ask
    say 'a='a
  catch RuntimeException    -- no number was typed
  end
end whileloop

-- the loop stops when a is greater than 10
-- the loop executes at least once
a = 100
loop until a > 10
  say "enter any number"
  a = ask * 2
  say 'a='a
end
```

Iterate Instruction

The iterate instruction (Figure 20) alters the flow of control within a loop construct. Iterate acts like a jump to the end of the loop body.

```
iterate [labelname]
```

Figure 20. Iterate Instruction

If *labelname* is not specified, iterate just steps to the innermost loop. If *labelname* is specified, iterate steps to the end of the loop body specified. The loop must be an active loop. If a loop does not have a label, the name of the control variable can be used instead.

Iterate Instruction Example:

```
/* language\Prime.nrx */
n = 2000
say 'Prime numbers from 2 to' n':'
loop label outer i=2 to n
   if i//100 = 0 then say
   loop label inner j=2 by 1 while j**2 < n
      if j<i & i//j = 0 then iterate outer
   end
   say i '\-'
end
```

Exit a Control Structure

The leave instruction (Figure 21) is used to leave a control structure, that is, a do block, select instruction, or loop.

```
leave [labelname]
```

Figure 21. Leave Instruction

If *labelname* is not specified, leave just leaves the current control statement. If *labelname* is specified, the leave instruction leaves the control statement with the specified label. The control statement must be in the same method. If a loop does not have a label, the name of the control variable can be used instead.

Leave Instruction Example:

```
-- displays 1 2 3 4 5 6 7
loop x=1
  if x // 7 = 0 then leave
end
```

If the control structure has a finally clause, the leave jumps into the finally clause.

Exit a Method

The return instruction (Figure 22) is used to leave the current method and return control, and optionally a result, to the point of invocation.

```
return [expression]
```

Figure 22. Return Instruction

The expression, if any, is evaluated, finally clauses are executed, and the value of the expression is passed to the caller.

Exit a Program

The exit instruction (Figure 23) ends the program immediately. Finally clauses are not executed.

```
exit [numericExpression]
```

Figure 23. Exit Instruction

The optional *numericExpression* is returned to the program that started the current NetRexx program.

Exit should be used very carefully in a method. The whole NetRexx program terminates, not just the current method.

Trace Instruction

The trace instruction (Figure 24) lets you trace each statement during execution for debugging purposes.

```
trace [methods]
trace [all]
trace [results]
trace [off]
```

Figure 24. Trace Instruction

You can trace the invocation of methods with their parameters, all clauses that are executed (and the methods), or even each expression evaluation and results assigned to variables (in addition to clauses and methods).

The trace option can, however, negate any trace instructions (see “Trace Option” on page 48).

Numeric Instruction

The numeric instruction (Figure 25) enables you to specify both the number of significant digits for calculations in the Rexx class, and the format of exponential notation.

```
numeric digits [n]
numeric form [scientific | engineering]
```

Figure 25. Numeric Instruction

See Figure 5 on page 7 for an example of specifying numeric digits.

Options

The options instruction (Figure 26) is used to pass special requests to the compiler. More than 10 options are available, but only two are of common interest: *binary* and *trace*. The complete list of options is discussed in “Options Keyword” on page 15.

```
options optionList
```

Figure 26. Options Instruction

All options can be used as command line switches when the NetRexx compiler is invoked, but any option set by the options instruction has precedence.

You use the options instruction at any place in the file, but we recommend using it in the first lines of the file, before the first class and method instructions.

Binary Option

The binary option forces the compiler to compile all classes in the current file as binary classes. This is the same as using the binary keyword when defining a class (see “Class Instruction” on page 25).

In a binary class, literals are assigned to primitive data types (see “Data Types” on page 20) or as Java strings. Native binary operators are used where appropriate (see *NetRexx Online Documentation - Binary values and operations*).

In classes that are not binary, terms in expressions are converted to the Rexx string type (see “The Rexx Class for Strings” on page 30).

The use of the binary option or binary keyword in class instructions can speed up the program.

We recommend avoiding the use of the binary keyword as long a possible, because you lose some of the automatic numeric conversions that improve the reliability of programs. If some routines of your program, such as drawing routines, need a boost, you can use the binary keyword for the class implementations. Only minor changes to the source code are necessary when changing to binary classes.

Trace Option

With the trace and notrace option you can control the use of the trace instruction.

Notrace prevents tracing overhead while leaving trace instructions in the program. The value of this function is that you can leave all trace instructions in your code and switch them off at a central point. If the program does not work as you expect, you can switch back the trace option and debug your program without having to recode the trace instructions.

Chapter 5. Using NetRexx As a Scripting Language

In this chapter we discuss NetRexx as a scripting language: that is, we use NetRexx for simple, straightforward programs, without using object-oriented facilities.

We show you how to write subroutines and functions, how to use files, how to call external programs, and we explain the Java code that is generated.

NetRexx programs that use NetRexx as a scripting language are called *script* programs, or *scripts*, in this chapter.

Why Scripts?

You can use NetRexx as a simple scripting language without using any object-oriented features. Scripts can be written very fast. There is no overhead, such as defining a class, and the programs contain only the necessary instructions.

The scripting feature can be used for test purposes. It is an easy and convenient way of entering some statements and testing them.

The scripting feature can also be used for the start sequence of a NetRexx application. When an application is started, the main method of the first public class in a file is called, as described in Chapter 6, “Creating and Using NetRexx Classes” on page 57. In many programs this is inconvenient and not a good style, because the starting sequence is a sequential procedure and not a function of an object or class. The sequence checks the arguments, creates some objects, and starts the main object.

An easy solution is to create a separate program that performs the startup sequence. This program will be short, easy to understand, and easy to maintain. You do not have to scroll through dozens of lines of code to find the main method.

Small helpful command line programs can be written in NetRexx as well. The great advantage of such programs is that they can be used with every operating system that supports Java.

Translating existing Rexx programs to NetRexx is possible if they do not call functions of the operating system.

Straightforward Programs

To use NetRexx as a scripting language, just type in your instructions. There are no restrictions, except for the use of the class and properties instruction.

A good example of a script program is our first program (see “Our First NetRexx Program” on page 7). It is pretty straightforward, takes advantage of special features (exception handling), and does not define any class or other object-oriented facilities.

Use the *say* instruction to print a string and the *ask* keyword to get a response from the user (see “Input and Output” on page 20). Figure 27 shows another example of NetRexx as a scripting language.

```
/* script\Game.nrx

   This is a small game.  The program chooses a number between 0 and 1000
   and you have to find out the number as fast as possible */

say "I'am choosing a number between 0 and 1000"
number = 1000*Math.random() % 1   -- %1 transform result to integer
say 'Found a number'
guess = int                       -- declares guess as variable of type int
loop count = 1 until guess = number
  say count 'try: \-'
  guess = ask
  select
    when guess > number then
      say guess 'is to big'
    when guess < number then
      say guess 'is to small'
    otherwise
      say
      say 'Congratualations. You did it with' count 'tries.'
  end
catch RunTimeException
  say 'Sorry, whole numbers only. You lost the game.'
end
```

Figure 27. A Simple NetRexx Script: Game.nrx

Subroutines and Functions

As your application becomes bigger, you will feel the need for subroutines or functions.

The difference between a subroutine and a function is that a subroutine does not return a result. NetRexx does not provide keywords for subroutines or functions. Both are implemented as methods in a scripting program. If a method returns a value, it is a function, if not, it is a subroutine. Figure 28 shows you how to define a method.

```
method name([parameterlist]) static
```

Figure 28. Methods for NetRexx as a Scripting Language

If you do not specify the *static* keyword, the method cannot be invoked from your program, and the NetRexx compiler complains about the use of the method, not the declaration:

```
24 +++ .... add()
   +++      ^^^
   +++ Error: Cannot refer to a non-static method directly from a static instruction
```

The *parameterlist* defines the parameters the method expects when it is called. See “Method Instruction” on page 28 for more information.

The parentheses can be omitted, if the method does not use any parameters.

Use the return instruction (see “Exit a Method” on page 46) to return the result of your method to the caller. If the method is a subroutine, use return without an expression:

```
return 5*a  -- returns the value of 5*a

return      -- no result value returned
```

Global variables are not available. A method cannot access any variable of the main part of the program or of another method. Each variable must be passed to a method as a parameter.

With this knowledge, we can now improve our game program to calculate the answer in a subroutine (method) and read the input in a function (method), as shown in Figure 29.

```
/* script\Game2.nrx

   This is a small game.  The program chooses a number between 0 and 1000
   and you have to find out the number as fast as possible */

say "I'am choosing a number between 0 and 1000"
number = 1000*Math.random() % 1   -- %1 transform result to integer
say 'Found a number'
guess = int                       -- declares guess as variable of type int
loop count = 1 until guess = number
  say count 'try: \-'
  guess = getNumber()             -- invoke a FUNCTION
  showAnswer(guess,number,count)  -- invoke a SUBROUTINE
end

-- method showAnswer is a subroutine
method showAnswer(guess,number,count) static
  select
    when guess > number then
      say guess 'is to big'
    when guess < number then
      say guess 'is to small'
    otherwise
      say
      say 'Congratualations. You did it with' count 'tries.'
  end

-- method getNumber is a function returning an integer
method getNumber static returns int
 loop forever
   number = ask      -- get the input and check the type
   if number.datatype('W') then return number  -- Ok, return the number

   say 'Sorry, but' number 'is not an integer.'
   say 'Try it again : \-'
 end
```

Figure 29. Using Methods As Subroutines and Functions: Game2.nrx

External Methods

We call a subroutine an external method if it is stored in a separate file. External methods are quite easy, if the files are stored in the same subdirectory.

A program calls an external method, using its file name qualified with the name of the method. Figure 30 shows how our game program calls an external method to read an integer from the user.

```
/* script\Game3.nrx

   This is a small game.  The program chooses a number between 0 and 1000
   and you have to find out the number as fast as possible */

say "I'am choosing a number between 0 and 1000"
number = 1000*Math.random() % 1   -- %1 transform result to integer
say 'Found a number'
guess = int                       -- declares guess as variable of type int
loop count = 1 until guess = number
  say count 'try: \-'
  guess = Input.getNumber()       -- invoke EXTERNAL METHOD
  showAnswer(guess,number,count)
end

-- method showAnswer is a subroutine
method showAnswer(guess,number,count) static
  select
    when guess > number then
      say guess 'is to big'
    when guess < number then
      say guess 'is to small'
    otherwise
      say
      say 'Congratualations. You did it with' count 'tries.'
  end
```

```
/* script\Input.nrx - separate file */

-- method getNumber is a function returning an integer
method getNumber static returns int
 loop forever
   number = ask      -- get the input and check the type
   if number.datatype('W') then return number  -- Ok, return the number

   say 'Sorry, but' number 'is not an integer.'
   say 'Try it again : \-'
 end
```

Figure 30. Calling an External Method: Game3.nrx and Input.nrx

External Methods in a Package

If you want to store your external methods in a different subdirectory, you must build a package:

1. Create a subdirectory in any directory that is listed in the CLASSPATH environment variable.
2. Copy all source files with external methods into this subdirectory.
3. Edit these files and add a package instruction as the first instruction of each file:

   `package` *packagename*

   where *packagename* is the relative name of the directory (starting from the CLASSPATH directory).
4. Edit the script files that use the external methods and add an *import* statement to the beginning of the file:

   `import` *packagename*

Calling Non-Java Programs

You can start external programs with the help of the JDK. The exec method of the Runtime class starts a program in another process and returns a Process object containing information about the process:

```
Runtime.getRuntime().exec( program )
```

Before using the exec method, you must get the current Runtime object, using the Runtime.getRuntime() call. If the specified program cannot be started, an IOException is thrown.

The Process class enables you to communicate with the child process, the external program, through its standard input, output, and error streams and to stop the process and retrieve its exit status. These functions are implemented in methods of the Process class:

getErrorStream Returns an InputStream that is connected to the standard error stream of the object

getInputStream Returns an InputStream that is connected to the standard output stream of the object

getOutputStream Returns an OutputStream that is connected to the standard input stream of the object

destroy Kills the process

exitValue Returns the exit value of the process. If exitValue is called before the end of the process, an IllegalThreadStateException is thrown.

waitFor Waits for the process to terminate.

Please see the *Java Toolkit Online Reference* for more information.

Figure 31 illustrates the use of the process methods.

```
/* script\NonJava.nrx

   This program starts an UNZIP program, redirect it's output,
   parses the output and shows the files stored in the zipfile */

parse arg unzip zipfile .

-- check the arguments - show usage comments
if zipfile = '' then do
   say 'Usage: Process unzipcommand zipfile'
   exit 2
end

do
   say "Files stored in" zipfile
   say "-".left(39,"-") "-".left(39,"-")
   child = Runtime.getRuntime().exec(unzip ' -v' zipfile) -- program start

   -- read input from child process
   in   = BufferedReader(InputStreamReader(child.getInputStream()))
```

Figure 31 (Part 1 of 2). Calling Non-Java Programs from NetRexx: NonJava.nrx

```
      line = in.readline

      start = 0    -- listing of files are not available yet
      count = 0
      loop while line \= null
         parse line sep  program
         if sep = '------' then start = \start
         else
           if start then do
              count = count + 1
              if count // 2 > 0 then say program.word(program.words).left(39) '\-'
                                else say program.word(program.words)
           end
         line = in.readline()
      end

      -- wait for exit of child process and check return code
      child.waitFor()
      if child.exitValue() \= 0 then
         say 'UNZIP return code' child.exitValue()

    catch IOException
       say 'Sorry cannot find' unzip
    catch e2=InterruptedException
       e2.printStackTrace()
   end
```

Figure 31 (Part 2 of 2). Calling Non-Java Programs from NetRexx: NonJava.nrx

Behind the Scenes

NetRexx compiles to Java, which does not have the features of a scripting language.

When you start a Java program, the main method of the class that is started is called. The main method must be static; it belongs to the class and not to an object.

NetRexx takes the file name of the program and creates a class for you. The program statements you coded become the main method of the class (Figure 32).

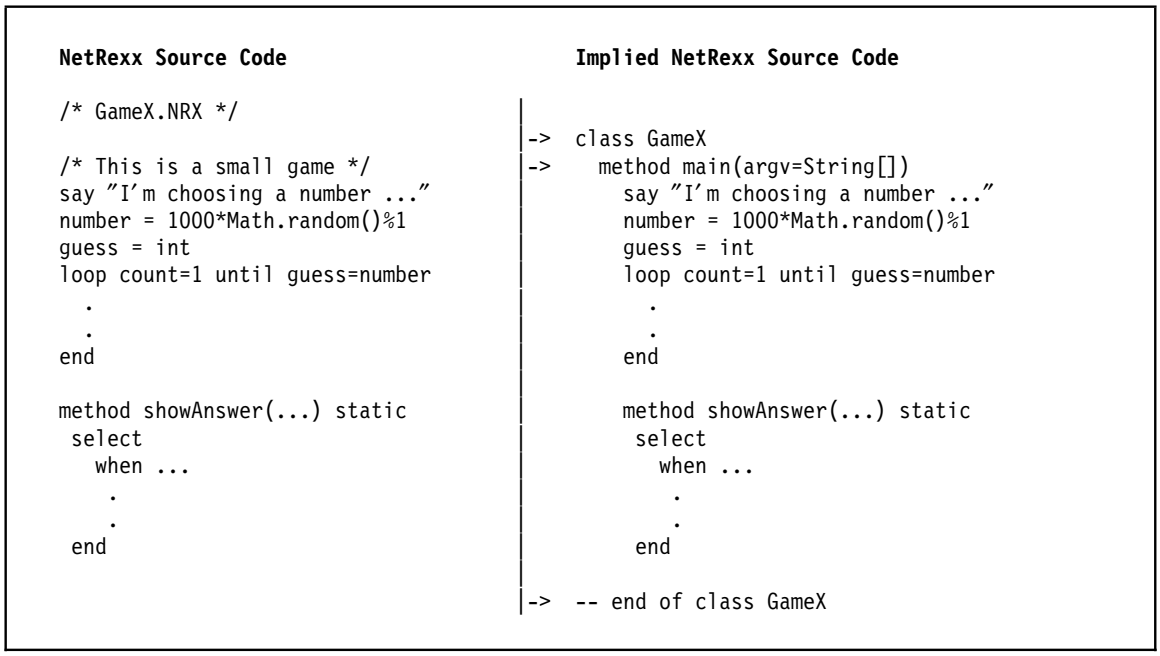


Figure 32. NetRexx As a Scripting Language: Generated Class and Main Method

Methods that you define in your script become additional class methods of the same class. Be sure to specify the *static* keyword.

Class variables and constants cannot be used, because NetRexx does not accept a properties statement as the first statement in a script file. The properties instruction must be used before any method instruction.

Calling external methods is quite easy; the programs are in the same package as long as they are in the same directory. Calling an external method is nothing other than calling a class method of another class.

If the files are stored in another subdirectory, you have to define a package (see "External Methods in a Package" on page 52 and "Packages" on page 70).

Handling Parameters in a NetRexx Script

If you want to pass one or multiple parameters to a NetRexx script program, you must declare the *main* method, before using any other statements (except for the optional class statement):

```
method main(args=String[]) static
   ...
```

Java passes arguments as an array of strings (Java strings). You can test the number of parameters, using the *length* property of the array, and access the parameters, using array indexes starting at 0:

```
method main(args=String[]) static
  say 'Number of arguments:' args.length
     loop i=0 to args.length-1
     say '- Argument number' i':' args[i]
  end
  ...
```

The main method of the first public class must be declared with the array of strings parameter. Java produces an error if the main method is declared differently.

However, NetRexx produces a warning if you declare a variable and never use it. Therefore, if you declare the main method but want to ignore the parameters, add a dummy line to prevent the annoying warning message:

```
method main(args=String[]) static
   args = args
   ...
```

Chapter 6. Creating and Using NetRexx Classes

In this chapter we take a closer look at classes, methods, and inheritance and concentrate on how NetRexx can create and use classes.

Definition of Class

A class is a collection of properties (variables) and methods (procedures) for carrying out operations on the properties.

A class definition is like a building plan for objects, and an object is an instance of a class. For example, Rexx is the String class of NetRexx. You cannot do anything with the class itself. A character string is an object of type Rexx. It has properties, such as the length of the string and its value, and methods, such as lower, associated with objects. You can work with the properties of the object directly, if allowed, or through method calls.

A class can protect the properties of the instances or the class itself by making them private. Every access must then be through methods.

Why Use Classes?

This example illustrates the need for classes:

- A group of three men open a private bank. The bank is a room with a book for each account and a bowl for money.
- If one of the three men wants to add to his account, he puts the money into the bowl and records the deposit in his book.
- If one of the men wants to withdraw money, he takes the money out of the bowl and records the debit in his book.

This method works very well as long there are only three people, who know each other. It is like a conventional programming language. Everybody has full access to the data, and everybody promises to follow the rules for access.

As the bank grows, the old method becomes obsolete. To make it work:

- The bank hires a teller, who is the only one with access to the account books and to the bowl with the money.
- When clients want to withdraw money from or deposit it to their account, they go to the teller, who conducts the transactions for them.

Let us apply this to a program:

- The account books and the bowl are our properties. The account book is an instance property, because every customer has his or her own account, and the bowl is a class property because there is only one.

- The teller is the implementation of the methods. He is the one who has access to the properties and is responsible for the transactions.
- Any discrepancy in an account is the result of the teller's mistake. You do not have to go to each customer (perhaps 10,000) and check what they were doing wrong.

Classes

A class is a construction plan for an object. It describes the data of an object and the methods to access and change the data.

An object is an instance of a class. It encapsulates the object data. The data of an object can be accessed through the methods of an object.

If a new object is constructed from a class, the instance variables (the data of the object) are created and initialized for the object. Every object has its own set of variables. The number of objects is limited only by the system resources.

The instance methods exist only once. They operate on the data of individual objects.

In addition there are class methods and class variables. These methods and variables belong to the class. There is only one set of class variables. Class variables and methods are used most commonly for defining constants or keeping track of all of the objects created in the class.

Properties

The variable definitions of a class are called *properties* in NetRexx. The term *property* includes instance and class variables.

Properties are distinguished by their behavior and visibility. A variable can behave as an instance variable, class variable, constant, or volatile instance variable:

Instance variable — The variable belongs to an instance of the class, and each object has its own variable. The variable is initialized when the object is constructed. It can only be accessed with a reference to an object. This is the default behavior.

Class variable — The variable belongs to the class and is initialized when the class is loaded.

Constant — A constant belongs to the class and cannot be changed.

Volatile instance variable — An instance variable that can be changed asynchronously, outside the control of the current process.

A variable's visibility, that is, how it can be accessed, can be public, inheritable, or private:

Public variable — A public variable can be used by every other class. There is no protection.

Inheritable variable — An inheritable variable can be used by classes that are in the same package (*friends* in C++) or subclasses of the current class.

Private variable — A private variable can be used only by methods of the same class.

An inheritable visibility is the default, if a Properties statement is not used to change it.

We recommend not using public variables, unless they are constants. Even if the use of the variable is not limited now, there is no guarantee that the visibility will not change in the

future. If visibility changes, all code that uses the class must be changed to use methods instead of direct assignments.

Use private variables carefully. A private variable is a strong restriction for subclassing. Use it if changes to the variable have restrictions and you cannot trust the programmers of subclasses.

We do not recommend trusting a user of your classes, but we do recommend trusting the programmers of subclasses, because they have to fix whatever goes wrong with their classes.

Table 5 shows the keywords for behavior and visibility in the properties instruction (see "Properties Instruction" on page 26).

Table 5. Keywords for Behavior and Visibility of Properties

Behavior	Visibility		
	Public	**Private**	**Inheritable**
Instance variable	public	private	inheritable
Class variable	public static	private static	static
Constant	public constant	private constant	constant
Volatile instance variable	public volatile	private volatile	volatile

Methods

A method is a subroutine or function defined in a class.

Three different kinds of methods are defined in NetRexx:

Instance methods An instance method belongs to an object. You can call an instance method only if you have a reference to an object. Instance methods have unlimited access to the instance and class variables of the object.

Class methods A class method belongs to the class. No reference to an object of the class is necessary. Class methods have unlimited access to the class variables but cannot access the data of any object of the class.

Constructor methods A constructor method belongs to the class and constructs an object of the class. Constructor methods do not need a reference to an object, because they create an object. The instance variables of the created object can be accessed from the constructor method. Class variables are also available. The name of a constructor method is the same as the class name.

The methods are distinguished by their behavior and visibility.

Behavior defines whether a method is overridable, final, abstract, or native:

Overridable methods An overridable method can be overridden in a subclass. The method of the subclass must use the same argument list (signature) as the overridden method.

Final methods A final method cannot be overridden in any subclass. A final method can be overridden if the signature is different.

Abstract methods Only the interface (parameter list) of an abstract method is defined. An abstract method must be implemented in subclasses of the class. A class with any abstract method

automatically becomes abstract itself. You cannot create an object in an abstract class.

Native method — A native method is implemented by the environment itself. A native method cannot be overridden.

We recommend avoiding the definition of final methods. Although a final method can be used more efficiently than an overridable method, it restricts the reuseability of the class.

Visibility defines how a method can be called:

Public methods — A public method can be called by any other class. There is no restriction. Public is the default.

Inheritable methods — An inheritable method can be called from subclasses or from classes that are in the same package.

Private methods — A private method can be called only from methods of its own class.

A public method is used as the interface for users of the class.

An inheritable method is used for internal functions of a class and cannot be called by users of the class. Inheritable methods do not check the arguments of the method; they trust the caller.

Table 6 shows the keywords for the behavior and visibility of methods.

Table 6. Keywords for Behavior and Visibility of Methods

Behavior	Visibility		
	Public	**Private**	**Inheritable**
Overridable instance method	(default)	private	inheritable
Final instance method	final	final private	final inheritable
Constructor method	(default)	private	
Overridable class method	static	static private	static inheritable
Abstract instance method	abstract	abstract private	abstract inheritable
Native instance method	native	native private	native inheritable

Signature of Methods

Methods have a list of arguments that are passed to them. If a method does not need any argument, the list can be empty. Method arguments may be optional with a default value. Optional arguments must be at the end of the argument list.

The signature is the name of a method and the types of the arguments in sequence. The return type of a method is part of the method definition but not part of the signature.

Examples of method definitions to illustrate signatures:

```
method a(a=int,b=String,c=int) -- has the signature a(int,String,int)

method c()                     -- has the signature c()

method a(a=int,b)              -- has the signature a(int,Rexx)
```

If a method has optional arguments, a set of signatures is related to that method. The set is built by creating a signature for the method and removing the last optional argument from the argument list (from right to left). This step is repeated as long as the method has optional arguments.

An example of a method with three optional arguments:

```
method b(a=int, b=String, c=int, d=int 5, e='Yes', f=boolean 1)
```

This method has four signatures:

```
b(int,String,int,int,Rexx,boolean)
b(int,String,int,int,Rexx)
b(int,String,int,int)
b(int,String,int)
```

Note: You cannot define two methods with the same signature in the same class.

Overloading Methods

Methods with the same name but different signatures overload each other.

NetRexx builds the signature of the method invocation and invokes the method with the matching signature.

Examples of overloading methods:

```
method a(a=int,b,c=Rexx 'default')      -- M1: signatures a(int,Rexx,Rexx)
  say 'a='a 'b='b 'c='c                 --      and       a(int,Rexx)

method a(b,c=Rexx 'default')            -- M2: signatures a(Rexx,Rexx)
  say 'b='b 'c='c                       --      and       a(Rexx)

method a(a=int)                         -- M3: signature  a(int)
  say 'a='a

a(5)               -- matching signature   M3: a=5
a(5,'Test')        -- matching signature   M1: a=5 b=Test c=default
                   -- the Java string "Test" is converted to a Rexx string
a('Test','Test')   -- matching signature   M2: b=Test c=Test
a('Test')          -- matching signature   M2: b=Test c=default
```

Constructor Methods

A constructor method constructs an object of the class. Constructor methods have the same name as the class. A default constructor is a constructor without any arguments.

The first instruction in such a method must create a new object of the class. The object is created by a call to the constructor of the superclass or to another constructor of the current class. If the first statement of a constructor method is not a call to a constructor of the superclass or the current class, a call to the default constructor of the superclass is added automatically.

After the construction of the object, the *this* keyword refers to the created object. All instance variables of the created object can be accessed.

A constructor method does not have to define the return type of the method. If the return instruction is used, it can be used only without a value or with *this* (*return this*).

Examples of constructor methods:

```
class Example                   -- extends Object per default

  method Example(testobject)    -- constructor with a parameter
    this()                      -- call the default constructor
    b.addElement(testobject)
```

```
method Example()              -- default Constructor
                              -- super() is inserted automatically
  a = Date()
```

The superclass constructor is invoked by a call to *super*([*argument*[,*argument*]...]).

Another constructor of the current class is invoked by a call to *this*([*argument*[,*argument*]...]).

NetRexx automatically defines a default constructor without a parameter list if the user does not define a constructor.

Invoking Methods

There are different ways of invoking methods:

- An instance method is invoked by specifying the object followed by a period and the method's name. The method's name must be followed immediately by a left parenthesis with no blank in between:

  ```
  c = 'abcd'
  a = c.right(6,"-")   -- would set a to "--abcd"
  ```

- If the method has no arguments, the parentheses are not necessary but recommended:

  ```
  c = 'AbCd'
  a = c.lower     -- would set a to "abcd", same as c.lower()
  ```

- If the method is invoked for the current object, the object and the period are not necessary:

  ```
  s = toString()  -- same as s = this.toString()
  ```

- If a method overrides an inherited method, the overridden method can be called with the *super* keyword:

  ```
  class Abc extends Ab
    method className() returns Rexx
      return "Class abc inherits from" super.className()
  ```

- In a constructor method, the constructor of the superclass must be called in the first instruction. If it is omitted, such a call is generated automatically:

  ```
  class Abc extends Ab
    method abc(a)       -- a is of type Rexx
      super(a)          -- invokes the constructor of class ab
      ...
    method abc(a,b)     -- a and b of type Rexx
      super(b)          -- invokes the constructor of class ab
       ...
    method abc(a,b,c)   -- a, b, and c of type Rexx
      this(a,b)         -- invokes the 2nd constructor of class abc
      ....
  ```

- To call a class method, the name of the class is used instead of an object:

  ```
  class Service
    method test() static  -- class method test
      ....
  class XYZ
    Service.test()        -- call the test method of the Service class
  ```

- If the class does not belong to the current package and there is no *import* statement for the package, a class method is invoked by specifying the fully qualified class name:

```
OtherPackage.Utilities.Service.test() -- fully qualified name of Service
```

- The constructor method of a class is invoked by using the name of the class followed by parentheses:

```
a = File('TEMP.TMP')
b = java.io.File('TEMP.TMP') -- same as line above
c = Example()                -- parenthesis are mandatory for constructors
```

Inheritance

We want to use our first example (see "Why Use Classes?" on page 57) to explain the inheritance of classes. The bank wants to open a new branch. The new branch will be the same as the old branch, except the money will be stored in a safe instead of a bowl. The men do not want to establish new procedures for using the bank, but two additional procedures are needed to open and close the safe. The teller needs some new instructions to put the money in the safe instead of the bowl.

If the old branch is a class, we would inherit from that class for the new branch. The safe is a new instance variable that extends the old class. In real life it might be implemented by a database. Two new methods have to be written to open and close the safe. Some of the methods for the teller have to be overridden to use the safe. Overriding means that the signature is still the same, but the implementation is changed.

Definition of Inheritance

In NetRexx a class can inherit all of the public or inheritable variables and methods from another class. The new class is a subclass of the old class, and the old class is the superclass for the new class.

The subclass extends the superclass. The subclass inherits the methods and variables of the superclass. A user of the class does not have to distinguish between methods that are implemented in the current class and methods that are inherited from the superclass. This is also true for properties, but the user should never have direct access to an instance or class variable in a well-designed class.

The subclass can override instance and class methods and instance and class variables (properties).

In the example that follows, class C2 extends (inherits from) class C1 and implements a new method. The use of methods defined in class C1 is transparent for users of class C2:

```
---------------------------------- Definition of class C1
class C1
  Properties
     a = int
     b = int

  method C1(a_ = int, b_ = int)
     a = a_
     b = b_

  method add() returns int
    return a + b

---------------------------------- Definition of class C2
class C2 extends C1
  method C2(a_ = int, b_ = int) -- constructor method for C2
```

```
    super(a_,b_)               -- calls constructor of C1

  method show                  -- new method
    say add()                  -- use of the inherited method add

--------------------------------- Usage of class C2
class XYZ ...
  obj = C2(4,5)                -- new object of type C2
  say obj.add()                -- uses method of C2, implemented in C1
  obj.show()                   -- uses method of C2, implemented in C2
```

Why Use Inheritance?

Data encapsulation and inheritance are the elements of an object-oriented language. Data encapsulation helps to ensure that the data can be changed only by methods that are part of the object definition. If an error occurs, it has to be in the methods of the object and not in thousands of lines of code that are using the object. The consistency and accuracy of the data of the object are guaranteed by the class implementation.

Inheritance is code reuse. Reusing code is one of the main goals of fast program development and reliable programs. The advantage of code reuse is that code is written only once. If there is an error in the code, it has to be corrected only once. If the code is copied instead of inherited, the error must be corrected in every copy. This may be a minor problem in the development cycle of the software, but it is a major problem in the maintenance cycle.

Overriding Methods

A subclass overrides a method of the superclass when it uses the same signature as the superclass.

The old method of the superclass is no longer accessible from users of the class. It is accessible only from method inside the class. Overridden methods of the superclass are called by using the keyword *super* instead of *this*. When calling a method in the same class, *this* is optional and most of the time omitted.

When a method is called on an object, the search for a matching method with the same signature always starts from the class of the object (see Figure 33).

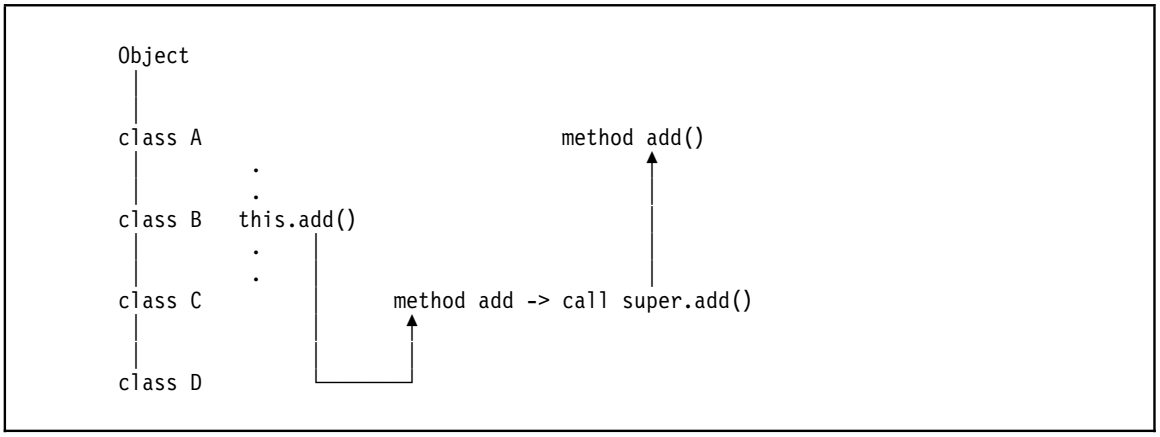


Figure 33. Search for Methods in the Class Chain

Notes:

1. In Figure 33 class A inherits from Object, class B from A, and so forth.
2. The type of the current object is class D.
3. A method in class B calls this.add(), which is the same as add(). NetRexx starts looking for the add method from the bottom of the class chain, that is, from class D. In this example add is implemented by class C.
4. When C calls super.add(), NetRexx starts looking for the method starting with class B. In this example Class A implements the add method, which is overridden by class C.

Overriding and Usage of Property Variables

Property variables can be overridden by a subclass. If a variable of a subclass has the same name as a variable of the superclasses, it is overridden.

A overridden variable in the superclasses can be accessed by using the *super* keyword.

In contrast to the search for methods, NetRexx starts the search for a variable in the class of the executing method.

Note: If the variable is not overridden but belongs to a superclass, we have to use *this* or *super* to access the variable. If *this* or *super* is not used, NetRexx creates a local variable.

This example illustrates the usage and overriding of variables:

```
class C1
   Properties
     a = int                    -- inherited by class C2
     b = int                    -- overridden by class C2
   method C1()                  -- default constructor, no statements
   method C1(a_=int, b_=int)    -- real constructor
     a = a_
     b = a + b_
   method show
     say 'b=' b                 -- accesses b of class C1

class C2 extends C1
     b = Rexx                   -- overrides b of class C1
   method C2(a_=int, b_=int)    -- constructor
     this.a = a_                -- without this a local variable is used
     b = a_ b_                  -- concatenate a_ and b_

------------------------------------ Usage of the class C2
class XYZ
     obj = C2(3,4)              -- construct a C2 object
     obj.show()                 -- shows b = 0 !
```

In this example class C2 overrides the instance variable *b* of C1. The constructor method of C2 assigns the value of *a_* to the inherited instance variable *a* of class C1, and the concatenated value of *a_* and *b_* (automatically converted to strings) to the instance variable *b* of class C2.

The show method is defined in class C1. Every instruction in this method has access to the instance variable defined in C1, or superclasses of C1. If a variable is overridden, it cannot be accessed. The show method shows the value of variable *b* of class C1, which was initialized to 0 when the object was constructed.

Usage or Inheritance

A main question of object-oriented design is whether to use inheritance or only a simple instance of a class, which is known as usage.

The ability to answer this question is one of the goals of object-oriented design; a good design tries to build a model of the real world. If we want to inherit from a class, we have to ask ourselves if the new class is a kind of its superclass. If we can say that the subclass is a kind of its superclass, inheritance is allowed.

Some examples will help you to understand the sentence above:

A Golden Retriever is a dog.

A sea lion is not a lion, even if the names are the same, and both animals share some common behavior.

A sorted list is a list.

A music store is a store.

A train is not a car, but a car is a vehicle, and a train is a vehicle.

A dictionary is not a list.

The last example is difficult to understand. The main characteristic of a dictionary is the relationship between the key and the value of a dictionary element. If we use a key, we can retrieve the related element. The elements of a dictionary are (usually) stored in a list, and some methods exists to browse this list. The direct access to an element through its key remains, however, the primary access. The conclusion is that a dictionary uses a list for its implementation, but it does not inherit from a list.

If the question, Is the new class a kind of its superclass? cannot be answered in the positive, the new class can use the other class but cannot inherit from it.

Abstract Classes

An abstract class cannot construct objects. Abstract classes are used to define a common behavior. A subclass of an abstract class can construct objects.

Abstract classes can optionally include abstract method definitions. If an abstract class includes abstract method definitions, the methods must be implemented by the subclasses.

A class that defines the behavior of graphical elements is a common example of an abstract class. The class defines the size of a rectangle, the origin, and common operations such as translation or rotation.

An instance of the graphical element class would make no sense, because there is no information about the kind of graphical element that should be drawn. The class is an abstract class.

Subclasses of the graphical element class describe real graphical objects such as lines, rectangles, or circles (see Figure 34). Instances of these subclasses can be constructed.

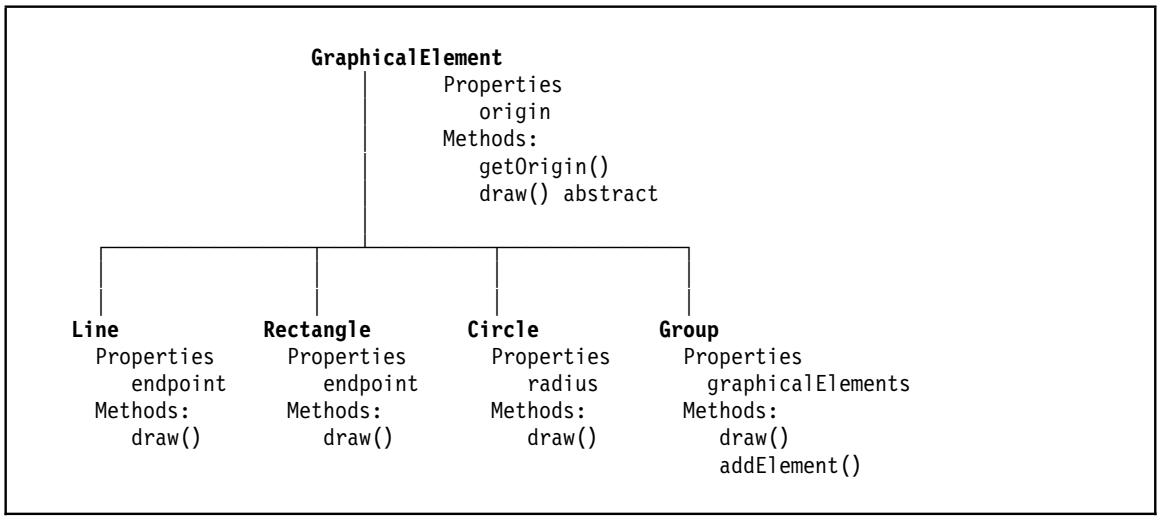


Figure 34. Use of Abstract Classes

Abstract class *GraphicalElement* implements the common behavior (*getOrigin* method) and defines the abstract *draw* method. The subclasses must implement the draw method, as well as the specialized methods that are not common.

The real power of abstract classes is used in the *Group* class. The *graphicalElements* property is a Vector (dynamic array) that stores elements of the GraphicalElement class. The implementation of the Group class is quite easy:

```
class Group extends GraphicalElement

  Properties
    graphicalElements = Vector

  method draw()
    loop i=0 to graphicalElements.size()-1
       obj = GraphicalElement graphicalElements.elementAt(i)
       obj.draw()
     catch ArrayIndexOutOfBoundsException
    end

  method addElement( obj = GraphicalElement )
    graphicalElements.addElement( obj )
```

The Group class uses a *Vector* to store a list of graphical elements. A *Vector* is a class of the JDK that implements dynamic arrays (see the *Java Toolkit Online Reference*).

You cannot construct an object of an abstract class, but you can use the abstract class to access objects of its subclasses. The real type of the objects are Line, Rectangle, Circle, or Group. If a method is called, the search for the method starts at the bottom of the class chain, for example, the class Line or any other of the subclasses of GraphicalElement.

All methods that are defined in the class of the variable, or its superclass, can be called.

If the method is not implemented in the class, only the interface of the method must be defined, for example, *draw()*. The interface definition is an abstract method. The abstract method can now be called, because the subclasses have to implement the method; otherwise it is not possible to create an object of the subclass.

Polymorphism

Figure 34 on page 67 illustrates the concept of polymorphism. The same *draw* method, defined as an abstract method in the superclass, must be implemented in each subclass.

When graphical objects of the subclasses are collected into a Vector of the Group class, the draw method can be invoked on each element of the Vector, independent of the actual type of the element.

The system is smart enough to invoke the correct draw method for each element of the collection. Using the same method name in subclasses and letting the system handle the invocation of the correct method is called *polymorphism*.

Interfaces

An interface class is a class that contains only abstract method definitions and constants.

A class can implement a single interface or multiple interfaces. The interface class acts in this case as a superclass of the class that implements the interface. The relationship between the class that implements interfaces and the interface classes is like multiple inheritance, except for the code reuse aspect of real inheritance.

One difference between interfaces and abstract classes is that abstract classes can implement some of the methods, whereas interfaces define abstract methods only. Another difference is that interfaces give you the function to define methods that accept arguments from classes of different class chains, without the loss of type safety.

This example illustrates the usage of interfaces. We want to implement a dynamic growing array that sorts its elements. We name the class *SortedVector*.

The main problem for the implementation of a sorted Vector is that the elements that are to be added must be compared for sorting.

The SortedVector class will have an *addElement* method that adds a new element to the vector and a *getElement* method that returns the object at a specified position. The addElement method does the sorting.

There are two possible ways of implementing this behavior:

- We define an abstract class, SortedVector, which implements all of the methods of the class, except for the addElement method. For every class we want to use with the sorted Vector, we define a new subclass that implements the addElement method. The type and methods of the elements are well known, and the sorting can be done in the addElement method. The getElement method returns an object of the generic Object type or is overridden in the subclasses.
- We define an interface class, called *Comparable*, with a method to compare two objects of the interface class. The SortedVector class accepts objects of type Comparable for the addElement method. The getElement method returns objects of type Comparable.

  Every class that will be used with the SortedVector class must implement the Comparable interface:

```
-- Interface Comparable which defines a method for comparing
class Comparable interface
  method greaterEqual(obj = Comparable) returns boolean abstract

-- sorted vector class uses the Comparable interface
class SortedVector
  Properties
    list = Vector()                -- Java Toolkit class (java.util.Vector)

  method addElement(obj = Comparable) returns int
```

```
        size = list.size()
        loop pos=0 to size-1
           objinlist = list.elementAt(pos)
           if objinlist.greaterEqual(obj) then do
              list.insertElementAt(obj,pos)   -- insert at current position
              return pos                      -- and return position
           end
         catch ArrayIndexOutOfBoundsException -- not possible
        end
        list.addElement(obj)            -- obj is greater as any object in list
        return size                     -- return position ( 0 based )
         .
         .

      -- Book Class implements the Comparable interface
      class Book implement Comparable
        Properties
          author                        -- lets sort by author
          ...
        method greaterEqual(obj = Comparable) returns boolean
          return author >= (Book obj).author
           .
           .
```

The subclassing approach is type safe. The disadvantage of the approach is that the sorting has to be implemented in every subclass. You can imagine that the sorting algorithm has a good chance of making mistakes, unless the very slow approach in the example is used. Every subclass must be tested very carefully.

The Comparable interface approach implements the sorting algorithm only once. It must be tested only once. The disadvantage of this approach is that we have to use type casting when comparing and retrieving objects. An instance of the sorted vector must contain objects of one class only.

Classes can implement multiple Interfaces (see “Class Instruction” on page 25). If a class does not implement all of the methods of all interfaces, the class must be an abstract class.

Interfaces can implement other interfaces. If an interface implements another interface, it is real multiple inheritance, because an interface can inherit (implement) from multiple interface classes:

```
class Equality interface
  method equality(obj=Equality) returns boolean abstract

class Comparable interface
  method greaterEqual(obj = Comparable) returns boolean abstract

class Compare interface implements Comparable, Equality
  method equality(obj=Equality) returns boolean
  ...                       -- must implement
  method greaterEqual(obj = Comparable) returns boolean
  ...                       -- must implement
```

Hint for Java Programmers

In Java interfaces are extended (inheritance) and not implemented by other interfaces.

NetRexx syntax:

```
class Compare interface implements Comparable, Equality
```

Java syntax:

```
public interface Compare extends Comparable, Equality
```

Class Libraries

A class library is a collection of classes. The classes are written for reuse. A class library is called a Package in NetRexx (and Java, of course).

One of biggest advantage of an object-oriented language is the reusability of the written code. Every time you write a new class, you should ask yourself “Is this class of common interest?” If the answer is yes, you should search the Internet for a class that has the same or at least similar function. There is a good chance that the work has already been done for you.

If you write the class on your own, you should make it possible to reuse the class. Spend a little more work to make it reusable, and it will pay back very fast.

Packages

A package is a class library. A class that belongs to a package includes a package instruction (Figure 35) before the first class definition.

```
package packagename
```

Figure 35. Package Instruction

The *packagename* identifies the package to which the classes of the source file belong.

Every compiled class is stored in a separate file, a class file. The name of the class file is the class name with the extension *.class*.

The *packagename* depends on the directory structure where the class files are stored. A package must be stored in a subdirectory of a directory in the CLASSPATH (see “Installation” on page 2). The package name is the path of the subdirectories. The subdirectories are separated with a period. An example illustrates this:

```
SET CLASSPATH=C:\JAVAOS2\lib\NetRexxC.zip;C:\MYCLASSES;...

Package:  util.container

Classes:  C:\MYCLASSES\util\container\SortedVector.class
          C:\MYCLASSES\util\container\Comparable.class
```

Class names can be short class names or qualified class names. The short class name is the name used in the class instruction. For your and our convenience, we call the short class name only class name. The qualified class name is the package name combined with the short class name:

```
util.container.SortedVector
util.container.Comparable
```

To use a class of a package in a different subdirectory, we have to use the qualified class name:

```
class Book implements util.container.Comparable
```

Because this is inconvenient, NetRexx provides the import instruction (Figure 36) to declare the usage of packages.

```
import importname
```

Figure 36. Import Instruction

The *importname* must be one of:

- A qualified class name
- A package name. To distinguish a package name from a class name, the package name has a trailing period after the last letter of the name. All classes in the package are imported.

An imported class can be referenced by the short name of the class:

```
import util.container.             -- all classes of package
import util.container.Comparable   -- single clas import

class Book implements Comparable   -- use short name after import
```

NetRexx automatically imports the NetRexx and JDK packages:

```
import netrexx.lang.
import java.lang.
import java.io.
import java.util.
import java.net.
import java.awt.
import java.applet.
```

If two packages include a class with the same short name, the qualified name must be used to access the desired class.

Packages in Zip Files

Zip files can be used to store packages. Zipped packages can be used on machines with file systems that do not support long file names, or to limit the number of files required for large class libraries.

Use the Info-Zip utility to compress the whole package path and include the zip file in the CLASSPATH.

Note: You must run the zip utility without compression. NetRexx and Java do not support compressed zip files.

The JDK packages and the NetRexx packages are distributed as zip files:

```
d:\...java_home...\lib\classes.zip
d:\...java_home...\lib\NetRexxC.zip
c:\MYCLASSES\itso.zip
```

Zip files must be referenced explicitly in the CLASSPATH environment variable:

```
SET CLASSPATH=.;d:\...\lib\classes.zip;d:\...\lib\NetRexxC.zip;c:\MYCLASSES\itso.zip
```

Globally Unique Package Names

The designers of Java have proposed a naming scheme for packages to avoid name conflicts. The package name should consists of three parts:

1. The first part is the organization to which the programmer belongs. This is the name of the domain, starting from right to left.
2. The second part identifies the developer. You can use the Internet ID for this part.
3. The third part is an identifier for the package. The programmer specifies this part.

Figure 37 shows an example of the naming scheme.

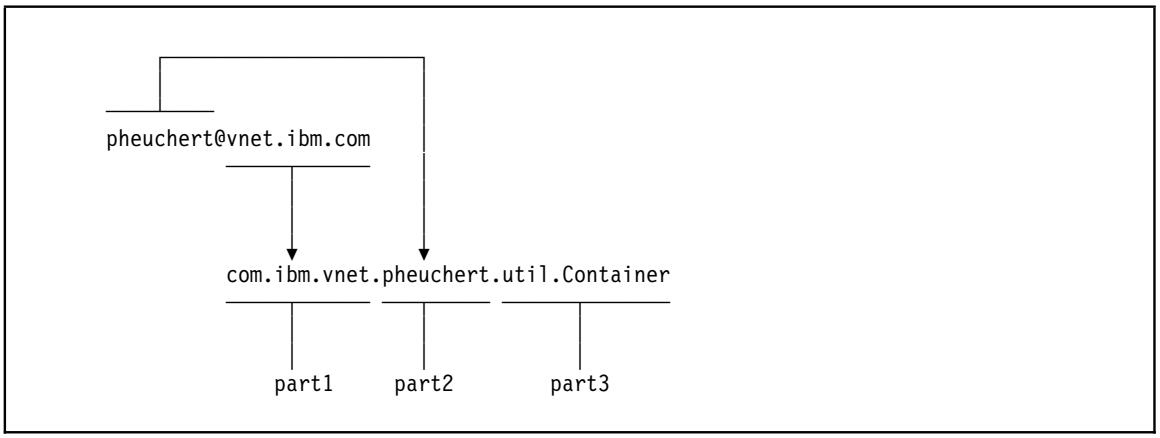


Figure 37. Global Naming Scheme for Packages

Using Java Classes

Every Java class can be used with NetRexx, because NetRexx compiles to Java classes. There is no difference between compiled Java and NetRexx classes.

In our example in Figure 31 on page 53 Java classes are used intensively.

Most of the Java classes you will use are part of packages. You must import the classes or packages, although NetRexx imports many basic packages by default.

When using Java classes and methods it is good practice to use the exact spelling of the Java names. NetRexx attempts to find Java classes and methods even if the spelling differs in case, but we recommend always using the Java mixed case spelling in your NetRexx programs.

Java Class Libraries

The JDK provides a rich set of packages:

java.lang	Includes the Java wrapper classes for primitive data types, the string classes, the system-related classes (such as process and toolkit), and the exception classes
java.util	Includes utility classes like dynamic arrays (Vector) and dictionaries (Hashtable) and string parsing functions
java.io	Includes the classes for file and stream handling
java.net	Includes the classes for TCP/IP connections
java.awt	Includes the classes for the graphical user interface
java.awt.image	Includes the classes to work with images
java.applet	Extends the graphical user interface classes to support applets

This list is not complete but includes the most important packages.

Using NetRexx Classes from Java

Using a NetRexx class from Java is the same as using a Java class from NetRexx. NetRexx compiles to Java classes that can be used by Java programs.

You should import the *netrexx.lang* package to use the short class name for the Rexx (NetRexx string) class.

A NetRexx method without a *returns* keyword can return nothing, which is the void type in Java, or a Rexx string. Examine your code carefully. A good programming practice is to declare the return type for every method, except if you are not returning a result in all return instructions.

NetRexx is case independent; Java is case dependent. NetRexx generates the Java code with the case used in the class and method instructions. For example, if you named your class *Game* in the NetRexx source file, the resulting Java class file is *Game.class*.

We recommend that the public class name in your source program match the NetRexx source file name. For example, if your source file is *FACTOR.NRX*, and your class is *Factor*, NetRexx generates a warning and changes the class name to *FACTOR* to match the file name. A Java program using the class name *Factor* would not find the generated class, because its name is *FACTOR.class*.

If you have problems, compile your NetRexx program with the options *-keep -format* (see “Compile Options” on page 14). Look at the *java.keep* file for the correct spelling style and method parameters.

Chapter 7. Creating Graphical User Interfaces

In this chapter we explore how NetRexx can be used to develop graphical user interfaces (GUIs) for applications and applets.

The JDK includes many classes to create a GUI. The interface is platform independent and can be used on every operating system that implements Java.

The GUI is currently not state of the art. It implements a basic set of controls and includes some basic drawing capabilities. Most of the more advanced controls such as notebooks and visual containers are missing. The problem is well known and there is a lot of work in progress to extend the GUI in a future JDK. A state of the art GUI is expected by the end of 1997.

The GUI is not part of the NetRexx language. All classes that are used to define a GUI are Java classes provided by the JDK.

The event handling changed completely between JDK 1.0.2 and 1.1.1. We used the JDK 1.1.1, which should be available for all platforms when this book is published.

Applets and Applications

Java GUI programs can be executed in two different environments: applets and applications.

Applets are Java programs that are executed in a Java-enabled Web browser. Applets are always part of an HTML page. Even the stand-alone applet viewer, which is part of the JDK, needs an HTML file to load an applet.

Applications are stand-alone programs, started from the command line and executed by the Java interpreter.

Because applets are provided by other programmers, they are untrusted code and restricted in there functions. Java provides built-in security functions to control the actions of every program executed by the Java interpreter.

Applications do not have any security restrictions. The restrictions that apply to applets depend on the implementation of the Web browser or applet viewer. Local applets, stored on your machine, should not have any restrictions. However, if you execute a local applet with a Netscape browser, the same restrictions as for remote loaded applets apply.

The main restrictions are that remote loaded applets do not have access to file I/O and can establish TCP/IP connections only to the Web server from which they are loaded.

The security restrictions are defined by a security manager, which is a Java class.

Applets

Applets are invoked by a Web browser when an HTML page includes an applet tag. When such a page is formatted, the applet tag causes it to send a request to the server for the applet code, which is the compiled class file, and starts running it. An applet viewer can be used instead of a Web browser. In this case the contents of the HTML page is skipped and only the applet is shown.

If a Web browser is used, the Java implementation of the Web browser runs the applet. Netscape, for example, has its own Java implementation on Windows. If you install a newer JDK, it has no effect on your applets running in the Netscape browser.

Note: When this book was written, only the Sun HotJava Web browser provided JDK 1.1.1 support. We believe that this will change in a few months. If you have the need to support Web browsers with JDK 1.0.2, you should consult the *Java JDK 1.0.2* documentation.

The Applet Tag

The applet tag (Figure 38) defines the class file used for the applet.

```
<applet code="appletClassFile"
     [codebase=codebaseURL]
     [object=serializedApplet]
     [alt=alternateText]
     [name=appletInstanceName]
     width=pixel height=pixel
     [align=alignment]
     [vspace=pixels] [hspace=pixels]
 >
     [<param name=parameterName value=value > ] ...
     [HTML text for browser without Java Support]
</applet>
```

Figure 38. HTML Applet Tag

The options can appear in any order:

code Defines the class file that contains the applet. The file name is applied relative to the code base directory. Absolute path names are not allowed.

codebase Defines the base URL to the directory of the applet code and defaults to the directory of the HTML file if omitted. The code base URL can point to a server other than the Web server. An applet can communicate only with the Web server sending the code (codebase). The codebase value can be accessed by the applet with the getCodeBase method.

object Defines the file holding the applet in serialized format. A serialized applet is stored with the values of all nontransient variables. This specification is mutually exclusive with the *code* option.

alt If the Web browser understands the applet tag but does not support or allow Java applets, the *alternateText* is shown.

name Defines a name for the loaded applet. A named applet can be found by other applets, which is necessary for communication between applets.

width, height Defines the initial width and height of the applet display area. The values can be redefined by the applet at run time.

align Defines the alignment of the applet. The possible values for this attribute are *left, right, top, texttop, middle, absmiddle, baseline, bottom,* and *absbottom*.

vspace Defines the space above and below the applet.

hspace Defines the space left and right of the applet.

param Defines a parameter that can be accessed with the getParameter method in the applet. A parameter is a keyword-value pair. You can define multiple param tags for the applet.

You can access a parameter with the getParameter method:

```
x = getParameter('COLOR')
```

This would return the value of a parameter with the COLOR keyword.

If a parameter does not exist, a null value is returned. Do not assign the return value of the getParameter method to a variable of type Rexx, because the program would signal a NullPointerException if the parameter is not set. (Note that this is fixed in NetRexx Version 1.1.)

Structure of an Applet

Any applet must be a subclass of the Applet class provided by the JDK. The Applet class defines six methods that are invoked by the system:

init() Is invoked when a document containing the applet is opened. The method is used to initialize the applet.

start() Is invoked when the document containing the applet is shown. The start method is always invoked after the init method. The start is also invoked when the applet has been stopped and the document is shown again.

stop() Is invoked when the document is no longer displayed. This method is always invoked before the destroy method is invoked.

destroy() Is invoked when the applet is "unloaded" from the applet viewer or Web browser. This method is used to clean up resources.

update(g=Graphics)
Is responsible for redrawing the applet. The default implementation redraws the background and calls the paint method.

paint(g=Graphics)
The paint method draws the components.

The applet class provides default implementations of the methods.

The start and stop methods may be invoked by the system multiple times throughout the applet's life. When an applet is stopped and restarted depends on the applet viewer or Web browser. Most of the currently available browsers stop the applet when another page is loaded and the current page is still available through the *back* command of the browser.

The start and stop methods are useful for starting and stopping animations when the applet becomes visible or invisible.

After the destroy method is invoked, the start method is never invoked again.

Figure 39 illustrates the life cycle of an applet through its methods.

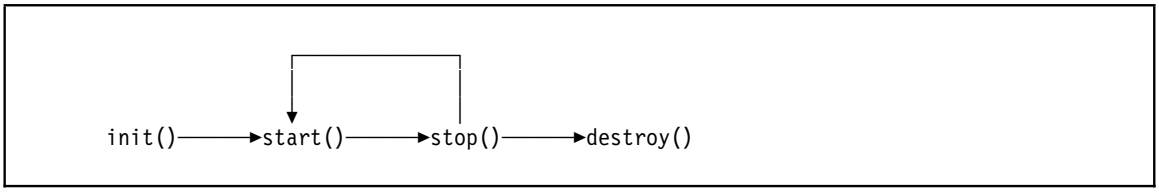


Figure 39. Life Cycle of an Applet

Figure 40 shows a simple applet.

```
/* gui\guifirst\GuiFirst.nrx

   First Simple NetRexx Applet */

import java.text.               -- Needed for the SimpleDateFormat class

class GuiFirst extends Applet
  Properties inheritable
    initDate  = Date
    startDate = Date
    stopDate  = Date

  method init()
    initDate = Date()

  method start()
    startDate = Date()

  method stop()
    stopDate = Date()

  method paint(g=Graphics)
    -- Format the result String
    f = SimpleDateFormat("H:mm:ss")     -- Formats hours:minutes:seconds
    result = "Init:" f.format(initDate) " Start:" f.format(startDate)

    if stopDate \= null then result = result " Stop:" f.format(stopDate)

    -- draw the string
    g.drawString(result,25,25)
```

Figure 40. First Simple Applet: GuiFirst.nrx

The HTML file that loads the applet in Figure 40 is shown in Figure 41.

```
<HTML>
<HEAD>
<TITLE>First Simple Applet Program</TITLE>
</HEAD>
<BODY>
<H1>First Simple Applet Program</H1>
<applet code=GuiFirst.class width=300 height=40
 alt='Please enable Java to see the applet'>
Sorry but your browser does not support Java applets.
</applet>
</HTML>
```

Figure 41. First Simple Applet HTML File: GuiFirst.htm

Start the applet viewer in the same directory where the HTML and .class files are stored:

```
appletviewer GuiFirst.htm   <== Windows 95/NT

applet GuiFirst.htm         <== OS/2
```

If you start this small applet with the applet viewer, and stop and start the applet with the viewer, you should see the window shown in Figure 42.

Figure 42. First Simple Applet in the Applet Viewer

The applet was initially started at the Init time. Start and Init time were the same. The applet was then stopped and restarted using the stop and start functions of the applet viewer.

Applications

An application is started with the Java interpreter from the command line. The environment of the installed JDK is used for the application. This environment can be different from the Java environment of your Web browser.

Only the main method is invoked automatically by the system. The main method is a public class method:

```
method main( ARGS = String[] ) public static
```

The main method typically creates a frame window and makes this window visible. Figure 43 shows a small application.

You do not have to define a main method. NetRexx can do this for you (see Chapter 5, “Using NetRexx As a Scripting Language” on page 49 and “Behind the Scenes” on page 54).

```
/* gui\guiapp\GuiApp.nrx

   First Simple NetRexx Application */

import java.text.               -- Needed for the SimpleDateFormat class

class GuiApp
  Properties inheritable
    window = Frame

  method main( args=String[] ) public static
    GuiApp()  -- Creates an object of the GuiApp class and shows the window

  method GuiApp()
    -- Create a frame window
    window = Frame('First GUI Application')

    -- Set the size of the window
    window.setSize(210,100)
```

Figure 43 (Part 1 of 2). First Simple GUI Application: GuiApp.nrx

```
      -- Set the window position to the middle of the screen
      d = window.getToolkit().getScreenSize()
      s = window.getSize()
      window.setLocation((d.width - s.width) % 2,(d.height - s.height)%2)

      -- Add a label to the window. The label text is centered
      f = SimpleDateFormat("H:mm:ss")     -- Formats hours:minutes:seconds
      text = Label("Started at:" f.format(Date()),Label.CENTER)
      window.add("Center",text)           -- Add the label to the window

      -- add the window event listener to the window for close window events
      window.addWindowListener( CloseWindowAdapter() )

      -- show the window
      window.setVisible(1)

  /*-----------------------------------------------------------------------------
    The CloseWindowAdapter exits the application when the window is closed.
    WindowAdapter is an abstract class which implements a WindowListener interface.

    The windowClosing() method is called when the window is closed.
    -----------------------------------------------------------------------------*/

  class CloseWindowAdapter extends WindowAdapter
    method windowClosing( e=WindowEvent )
      exit 0
```

Figure 43 (Part 2 of 2). First Simple GUI Application: GuiApp.nrx

The code of the program requires some explanation:

- The main method creates an object of the GuiApp class, which is our application. The constructor method of the class creates a Frame object (which is a frame window), initializes the window, and shows it on the screen.
- The window would be visible in the upper left corner of your desktop. This is inconvenient, so we ask the current Toolkit for the screen size and center the window in the middle of the screen:

  ```
  -- Set the window position to the middle of the screen
  d = window.getToolkit().getScreenSize()
  s = window.getSize()
  window.setLocation((d.width - s.width) % 2,(d.height - s.height)%2)
  ```

- The absence of the paint method is the most visible difference between an application and an applet. The application uses a component that does the drawing on its own. Most of the currently available applets are drawing graphics and require a paint method. Most applications are constructed using components (see “User Interface Controls” on page 82) and do not need to draw the window.
- The Java frame windows enables programmers to control the close event, when the window is closed by the frame controls. The default behavior is to ignore the event. We add a WindowListener to the window, which listens to the close event and exits the application:

  ```
  -- add the window event listener to the window for close window events
  window.addWindowListener( CloseWindowAdapter() )
  ```

  See “Event Handling” on page 111 for more details about event processing.

Figure 44 shows the result of the program.

Figure 44. The First GUI Application

Running As an Applet or an Application

A NetRexx GUI application can be coded so that it can run as either an applet or an application.

The class is coded as a subclass of the Applet class, with a main method that is executed when run as an application. The main method allocates the instance object (the applet), creates the frame, adds the applet to the frame, and then calls the init method to lay out the frame.

When run as an applet, the frame is allocated by the system, and the init method is called automatically.

Figure 45 shows the NetRexx code that can run as an applet or an application.

```
/* gui\guiapp\GuiApplt.nrx

   First Simple NetRexx Application or Applet using same code */

import java.text.               -- Needed for the SimpleDateFormat class

class GuiApplt extends Applet

  Properties inheritable static
    guiobj = GuiApplt                       -- the instance
    applic = byte 0                         -- applet

  Properties inheritable
    text1  = Label                          -- label text init/stop
    text2  = Label                          -- label text start

  method GuiApplt()                         -- constructor
    super()

  method main(args=String[]) public static  -- APPLICATION ONLY
    applic = 1
    guiobj = GuiApplt()                     -- creates the instance
    window = Frame('Application or Applet') -- create a Frame window
    window.setSize(210,100)                 -- set the size of the window
    d = window.getToolkit().getScreenSize() -- center the window
    s = window.getSize()
```

Figure 45 (Part 1 of 2). GUI Application or Applet: GuiApplt.nrx

```
      window.setLocation((d.width - s.width) % 2,(d.height - s.height)%2)
      window.add("Center",guiobj)             -- add Applet to Frame
      guiobj.init()                           -- init Applet
      window.addWindowListener( CloseWindowAdapter() )  -- close event
      window.setVisible(1)                    -- make window visible

    method init()                             -- APPLET and APPLICATION
      super.init()
      this.setLayout(null)
      f = SimpleDateFormat("H:mm:ss")         -- formats hours:minutes:seconds
      text1 = Label("Init at:" f.format(Date()))
      text2 = Label(" ")
      this.add(text1)                         -- add the labels to the window
      this.add(text2)
      text1.setBounds(40,30,120,15)           -- and size them
      text2.setBounds(40,50,120,15)

    method start()                            -- APPLET ONLY
      f = SimpleDateFormat("H:mm:ss")         -- change label text
      text2.setText("Started at:" f.format(Date()))
      super.start()

    method stop                               -- APPLET ONLY
      f = SimpleDateFormat("H:mm:ss")         -- change label text
      text1.setText("Stopped at:" f.format(Date()))
      super.stop()

    method destroy                            -- APPLET and APPLICATION
      super.destroy()
      if applic = 1 then exit 0               -- end application

  /*---------------------------------------------------------------------------
    The CloseWindowAdapter exits the application when the window is closed.
    WindowAdapter is an abstract class which implements a WindowListener interface.

    The windowClosing() method is called when the window is closed.
    ---------------------------------------------------------------------------*/

  class CloseWindowAdapter extends WindowAdapter
    method windowClosing( e=WindowEvent )
      GuiApplt.guiobj.destroy()
```

Figure 45 (Part 2 of 2). GUI Application or Applet: GuiApplt.nrx

User Interface Controls

In this section we describe the user interface controls, such as menus, buttons, and entry fields, that are included in the JDK.

The user interface controls are derived from the Component class. Table 7 summarizes the available component classes.

Table 7. Component Classes of the GUI	
Class	**Description**
Label	Component that shows static text
TextField	Entry field for text
TextArea	Multicolumn entry field for text
Button	A textual button
Checkbox	A check box is like a state button. It can be toggled between the on and off state. Grouped check boxes can be used as radio buttons.
List	List box that presents a list of selectable strings
Choice	Drop-down list box
Scrollbar	A scroll bar is used to specify an integer value in a range of values.
Menubar	The menu bar that is attached to frame windows
Menu	List of menu items shown from the menu bar
MenuItem	Choice in a menu
CheckboxMenuItem	A menu item that represents a choice in a menu
PopUpMenu	Menu that is independent of a menu bar

Each of these classes has a series of methods that manipulate the contents or appearance of the objects. User actions on these objects resolve in low-level and semantic events that have to be handled by event handling methods (see “Event Handling” on page 111).

Label

A label is a component that displays a single line of text. The text of the label can be modified by the program, but not by the user.

Constants, constructor, and methods of interest:

Constants

CENTER Alignment: center the text in the label

LEFT Alignment: text is left justified

RIGHT Alignment: text is right justified

Constructor

Label(s=String ’’,alignment=int LEFT)
Creates a Label object

Methods

getText() returns String
Retrieves the text of the label

setText(s=String)
Sets the text of the label

setFont(f=Font)
Sets the font of the label

TextField

The TextField component is an entry field used to edit a single line of text.

Constructors, methods, and events of interest:

Constructors

TextField()
: Creates a TextField object

TextField(s=String)
: Creates a TextField object and sets the text of the object

TextField(chars=int)
: Creates a TextField object and sets the visible size of the object to the specified value. The text of the object is not limited by the size parameter.

TextField(s=String,chars=int)
: Combination of the previous two constructors

Methods

getText() returns String
: Retrieves the text of the entry field

setText(s=String)
: Sets the text of the entry field

setEchoChar(c=char)
: Sets the echo character for the entry field. The entry field still maintains the original characters but shows only the echo character. This method is useful for a password entry field.

setFont(f=Font)
: Sets the font of the label

Semantic Events

ActionEvent
: Fired when the *Return* key is pressed

TextEvent Fired when the text in the entry field changes

Low-Level Events

KeyEvent To change the key entered, use the setKeyChar method of the event in the keyTyped method of the listener. Use the consume method of the event to delete the key stroke.

FocusEvent
: Fired when the component gets or loses the focus

The following example shows an entry field that uses a key listener which allows alphabetic characters only and changes the characters to uppercase:

```
...
field = TextField()
field.addKeyListener(KeyTester())
...

class KeyTester implements KeyAdapter
  method keyTyped( e=KeyEvent )
    key = Rexx e.getKeyChar()
    if key.datatype('M') then e.setKeyChar(key.upper())
                         else e.consume()
```

Refer to “KeyCheck Class” on page 265 for a complete implementation of a key listener.

TextArea

The TextArea component is an entry field used to edit multiple lines of text.

Constants, constructors, methods, and events of interest:

Constants

SCROLLBARS_BOTH
: Create and display both vertical and horizontal scroll bars

SCROLLBARS_HORIZONTAL_ONLY
: Create and display horizontal scroll bar only

SCROLLBARS_VERTICAL_ONLY
: Create and display vertical scroll bar only

SCROLLBARS_NONE
: Do not create or display any scroll bars for the text area

Constructors

TextArea(rows=int,columns=int)
: Creates a TextArea object with the specified rows and columns visible

TextArea(s=String,rows=int,columns=int)
: Creates a TextArea object like the constructor above and sets the text to the string

TextArea(s=String,rows=int,columns=int,scrollbar=int)
: Creates a TextArea object like the constructor above and shows only the specified scroll bars

Methods

getText() returns String
: Retrieves the text of the entry field

setText(s=String)
: Sets the text of the entry field

append(s=String)
: Adds a string to the end of the text in the entry field

insert(s=String,pos=int)
: Inserts a string at a given position to the text in the entry field

replaceRange(s=String,start=int,end=int)
: Replaces the text of a given range with a new string. The character at the end position is not included in the range.

setFont(f=Font)
: Sets the font of the label

Semantic Events

TextEvent Fired when the text in the entry field changes

Button

The Button component is a textual push button.

Constructor, methods, and events of interest:

Constructor

Button(label=String)
: Creates a Button object with the specified label

Methods

setEnabled(b=Boolean)
: Enables (1) or disables (0) the button

getLabel() returns String
: Retrieves the label of the button

setLabel(s=String)
: Sets the label of the button

setFont(f=Font)
: Sets the font of the label

Semantic Events

ActionEvent
: Fired when the push button is pressed

Low-Level Events

FocusEvent
: Fired when the component gets or loses the focus

Checkbox

The Checkbox component maintains and visualizes a boolean variable. The variable can be true (checked) or false (not checked).

The state of a check box has no influence on the state of other check boxes in the same window, unless the check boxes are grouped by a CheckboxGroup.

A CheckboxGroup is not a visible class of the JDK. If check boxes are grouped with a CheckboxGroup, they act like radio buttons. Only one check box of a group can be set. If another check box is set, the first one is cleared.

Methods and events of interest:

Methods

getState() returns Boolean
: Returns the state of the check box. The check box is set if the state is 1 (true).

setState(b=Boolean)
: Sets the state of the check box

setEnabled(b=Boolean)
: Enables (1) or disables (0) the check box

getLabel() returns String
: Retrieves the label of the check box

setLabel(s=String)
Sets the label of the check box

setCheckboxGroup(cb=CheckboxGroup)
Sets the CheckboxGroup to which the check box belongs. Typically the group is set with the constructor of the check box.

setFont(f=Font)
Sets the font of the label

Semantic Events

ItemEvent Fired when the state of the check box is changed

Low-Level Events

FocusEvent
Fired when the component gets or loses the focus

Figure 46 shows an applet with four check boxes. The first two check boxes are members of a CheckboxGroup; the last two check boxes are not members of a CheckboxGroup.

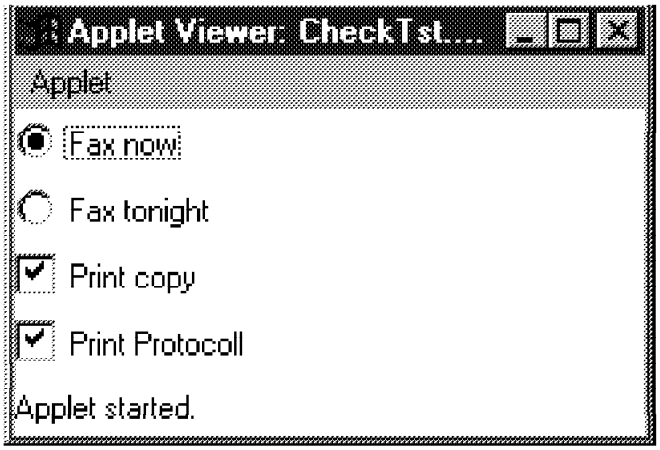


Figure 46. Applet with Check Boxes

Figure 47 shows the applet code.

```
/* gui\checkbox\CheckTst.nrx

   Applet with check boxes and a CheckboxGroup*/

class CheckTst extends Applet

method init()
  setLayout(GridLayout(0,1))        -- new layoutmanager with one column
  cg = CheckboxGroup()              -- CheckboxGroup for the first checkboxes
  add(Checkbox("Fax now",cg,0))     -- First  checkbox member of cg
  add(Checkbox("Fax tonight",cg,1)) -- Second checkbox member of cg
  add(Checkbox("Print copy",1))     -- Third  checkbox
  add(Checkbox("Print Protocoll",1))-- Fourth checkbox
```

Figure 47. Check Box Example: CheckTst.nrx

List

The List class implements a list box. The list box presents a list of strings. Depending on the state of the list box only, one or multiple strings can be selected.

The currently available List class does not support horizontal scroll bars or the use of objects instead of strings.

Methods and events of interest:

Methods

add(s=String,pos=int -1)
: Adds a string to the list. If *pos* is omitted (or -1), the string is added to the end of the list.

getItemCount() returns int
: Returns the number of items in the list

remove(pos=int)
: Removes a string at the specified position of the list

remove(s=String)
: Removes the specified string from the list

removeAll()
: Empties the list

select(pos=int)
: Select the item at the specified position

deselect(pos=int)
: Deselects the item at the specified position

getSelectedIndex() returns int
: Returns the index of the selected item. If no item is selected, -1 is returned.

getSelectedItem() returns String
: Returns the selected item. If no item is selected, *null* is returned.

getSelectedIndexes() returns int[]
: Returns an array with the indexes of the selected items

getSelectedItems() returns String[]
: Returns an array of strings of the selected items

Semantic Events

ItemEvent Fired when an item is selected or deselected. The getItem method of the ItemEvent class returns an Integer object for ItemEvents of List objects. Integer is a Java wrapper class for integers.

ActionEvent
: Fired when a double-click on an item occurs. The string returned from the getActionCommand method of the ActionEvent class is the string of the item where the double-click was performed.

Low-Level Events

FocusEvent
: Fired when the component gets or loses the focus

The add and remove methods are also available as addItem and delItem. Currently the item has to be a string, so there is no difference between these methods.

We suspect that these functions will accept objects in future releases, so we recommend using add and remove only, for compatibility.

Note: The getSelectedItem function returns a *null* value if no item is selected. If you assign the return value of the function to a variable of type Rexx (NetRexx string), and no item is selected, a NullPointerException is signaled:

```
s = Rexx
s = listbox.getSelectedItem() -- unsafe because the method can return null
```

Make sure that an item is selected before invoking the getSelectedItem method, or use a variable of the Java String class:

```
s = String
s = listbox.getSelectedItem() -- safe
```

This behavior has been fixed in NetRexx 1.1 and you can use a Rexx string as well.

Choice

The Choice class implements a drop-down list box. It is similar to a list, except that it does not need as much space.

In its normal state, the Choice class displays the currently selected string and a small image button. If the button is pressed, the entire list of choices is displayed. After the selection of an item, the list shrinks back to a single line.

One item of a Choice class is always selected.

Methods and events of interest:

Methods

add(s=String,pos=int -1)
: Adds a string to the list. If *pos* is omitted (or -1), the string is added to the end of the list.

getItemCount() returns int
: Returns the number of items in the list

remove(pos=int)
: Removes a string at the specified position of the list

remove(s=String)
: Removes the specified string from the list

removeAll()
: Empties the list

select(pos=int)
: Selects the item at the specified position

getSelectedIndex() returns int
: Returns the index of the selected item

getSelectedItem() returns String
: Returns the selected item (string)

Semantic Events

ItemEvent Fired when an item is selected. The getItem method of the ItemEvent class returns the string of the selected item for ItemEvents of Choice objects. Integer is a Java wrapper class for integers.

Low-Level Events

FocusEvent
: Fired when the component gets or loses the focus

Scrollbar

The Scrollbar class is not very useful. Use the ScrollPane class, which implements a container with scroll bars, instead.

Menus

Two types of menus are available, pop-up menus and menu bars.

A pop-up menu usually is not visible to the user but pops up through an action of the user. The action depends on the operating system; in many cases it is the right mouse button. Java provides a platform-independent abstraction of the pop-up menu trigger event.

A menu bar is attached to a frame. The menu bar is located directly under the title bar of the owning frame, except on some platforms where the menu bar appears at the top of the screen. The menu bar contains menus. Usually only the labels of the menus are visible. If the user clicks on a menu label in a menu bar, the menu associated with the label appears, as shown in Figure 48.

Menu items can optionally own a shortcut key. The definition of a shortcut key is operating system dependent and is handled by the JDK.

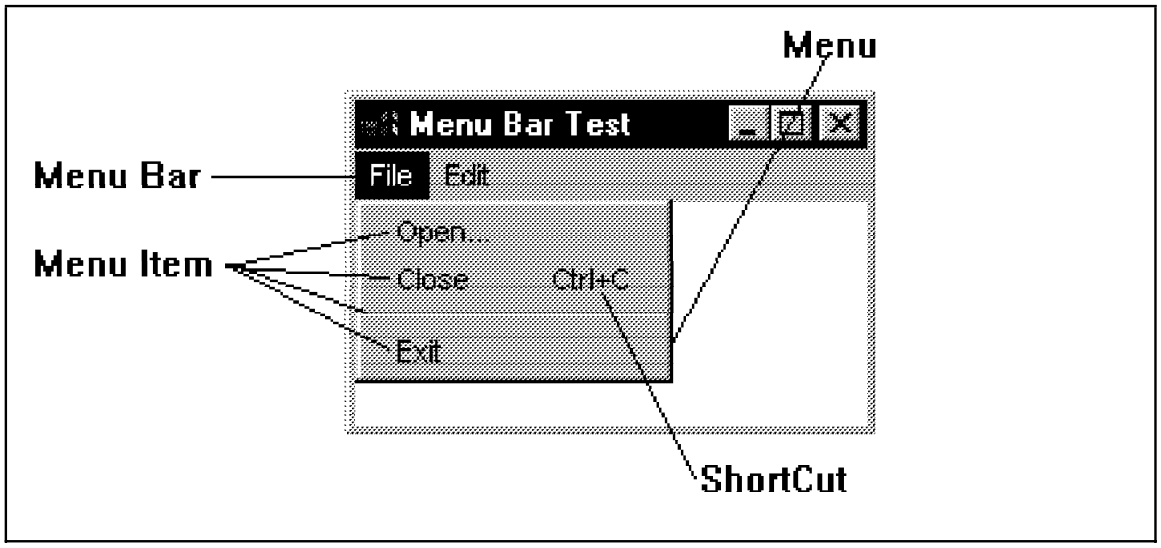


Figure 48. Menu Bar Sample Application

Figure 49 shows the source code for the menu bar sample application.

```
/* gui\menubar\MenuBarX.nrx

   This program shows the usage of a menu bar */

---------------------------------------- MenuBarX class (implicit)
   MenuBarTest()

---------------------------------------- MenuBarTest class
```

Figure 49 (Part 1 of 2). Menu Bar Sample Application: MenuBarX.nrx

```
class MenuBarTest
  Properties inheritable
    status = Label()

  method MenuBarTest()
    win = Frame("Menu Bar Test")      -- creates the window

    mb = MenuBar()                    -- creates the menubar (Java)

    m  = Menu("File")                 -- first menu in the menubar
    mb.add(m)                         -- add menu to menubar

    mi = MenuItem("Open...")          -- menu item for menu File
    m.add(mi)                         -- add menu item to menu File
    mi.addActionListener(MenuAction(status,"Open")) -- menu action

    -- next menu item for menu File with a shortcut Ctrl-c
    mi = MenuItem("Close",MenuShortcut(KeyEvent.VK_C))
    m.add(mi);
    mi.addActionListener(MenuAction(status,"Close")) -- menu action

    m.addSeparator();                 -- add a separator to the menu

    mi = MenuItem("Exit")             -- next menu item for menu File
    m.add(mi);
    mi.addActionListener(CloseMenu()) -- menu action

    m = Menu("Edit")                  -- next menu in menubar
    mb.add(m)

    win.setMenuBar(mb)                -- add the menubar to the frame window

    win.add("South",status)           -- add the label to the window

                                      -- set window position to the middle of the screen
    win.setSize(200,130)
    d = win.getToolkit().getScreenSize()
    s = win.getSize()
    win.setLocation((d.width - s.width) % 2,(d.height - s.height)%2)
    win.setVisible(1)

---------------------------------------- MenuAction class
class MenuAction implements ActionListener
  Properties inheritable
    text        = String
    field       = Label

  method actionPerformed(e = ActionEvent)
    field.setText('Menu' text 'selected')

  method MenuAction(aLabel = Label, aString = String)
    field       = aLabel
    text        = aString

---------------------------------------- CloseMenu class
class CloseMenu implements ActionListener
  method actionPerformed(e = ActionEvent)
    exit 0
```

Figure 49 (Part 2 of 2). Menu Bar Sample Application: MenuBarX.nrx

MenuBar

A MenuBar object must be attached to a frame window:

```
framewindow.setMenuBar(mb)
```

An applet cannot contain a menu bar, because there is no frame window.

Methods of interest:

Methods

add(m=Menu)
: Adds a menu to the menu bar

remove(m=MenuComponent)
: Removes a menu from the menu bar

setHelpMenu(m=Menu)
: Adds a menu as a help menu. On some systems help menus are on the right side of the menu bar. Only one help menu can exist in a menu bar. If a second help menu is added, the first one is removed.

Figure 49 on page 90 shows the usage of a menu bar.

Menu

A menu object contains menu items and has a label. A menu object is inserted into a menu bar.

Menus can be created with a tear-off attribute. If a tear-off menu is supported by the operating system, the menu can be cloned and used as a top-level window, when torn off.

Methods and events of interest:

Methods

add(m=MenuItem)
: Adds a menu item to the menu

addSeparator()
: Creates a menu item as a separator and adds it to the menu

remove(m=MenuComponent)
: Removes a menu item from the menu

Semantic Events

ActionEvent
: Fired when any menu item of the menu is selected. The getCommand method of the ActionEvent class returns the label of the selected menu item, unless a shortcut key fired the event. If a shortcut key fired the event the getCommand method returns a null string (see note in “List” on page 88).

  We do not recommend attaching an action listener to a menu. Attach an action listener to a menu item instead.

Figure 49 on page 90 shows the usage of menus.

MenuItem

A menu item is an entry in a menu or pop-up menu. A menu item has a label and optionally a shortcut key.

A shortcut key is an object of the MenuShortcut class. The key is defined using the key definitions of the KeyEvent Class:

```
shortcut = MenuShortcut(KeyEvent.VK_C)     -- Shortcut for c key.
```

For Windows 95, Windows NT, and OS/2 systems, shortcuts are invoked with the Ctrl key, for example, Ctrl-c.

If a boolean variable with the value 1 is added as a parameter to the MenuShortcut constructor, the short cut is set to Shift-*key*.

Only numbers and letters are allowed for MenuShortcuts; function keys cannot be used.

Methods and events of interest:

Methods

setEnabled(b=Boolean)
: Enables (1) or disables (0) the menu item

setLabel(s=String)
: Changes the label of the menu item

Semantic Events

ActionEvent
: Fired when any menu item of the menu is selected. The getCommand method of the ActionEvent class returns the label of the selected menu item, unless a shortcut key fired the event. If a shortcut key fired the event, the getCommand method returns a null string (see note in “List” on page 88).

Figure 49 on page 90 shows the usage of a menu items.

Pop-Up Menus

Pop-up menus, which are also known as *context menus*, are shown when the user invokes the pop-up menu with the mouse.

The pop-up menu must be owned by a parent component, which should be the applet or frame window. Only one parent at one time is possible.

To show the pop-up menu you must catch the mouse event when the user requests the pop-up menu.

The mouse action differs among operation systems:

- In Windows and OS/2 systems, a menu pops up when the right mouse button is released.
- In Motif systems a menu pops up when the right mouse button is clicked.

The trigger definition is encapsulated by the JDK. The MouseEvent class provides the isPopupTrigger method to detect a trigger event for a pop-up menu.

The pop-up menu is shown when the show method of the menu is invoked:

```
show(origin=Component, x=int, y=int)
```

The parameters for the show method are provided with the mouse event:

```
-- e     is an object of MouseEvent
-- popup is an object of PopupMenu
popup.show(e.getComponent(),e.getX(),e.getY()
```

Note: In *JDK 1.1 - AWT Enhancements*, the mouse events are captured by using the enableEvents method of an applet. There are some problems if you catch mouse events in this way:

- The enableEvents method is protected. You have to subclass the component to use the method.
- If the applet or frame window owns some components, the mouse events are not propagated to the frame or applet. You have to subclass each component again to get the mouse events.

We used a mouse event listener (see Figure 51 on page 95), which can be attached easily to any component. The drawback of this approach is that you must catch the mouse pressed and mouse release event, because the isPopupTrigger method is operating system dependent and can be true for the mouse pressed or mouse released event.

Methods and events of interest:

Methods

show() Pops up the menu

add(m=MenuItem)
: Adds a menu item to the menu

addSeparator()
: Creates a menu item as a separator and adds it to the menu

remove(m=MenuComponent)
: Removes a menu item from the menu

Semantic Events

ActionEvent
: Fired when any menu item of the menu is selected. The getCommand method of the ActionEvent class returns the label of the selected menu item, unless a shortcut key fired the event. If a shortcut key fired the event, the getCommand method returns a null string (see note in “List” on page 88).

  We do not recommend attaching an action listener to a menu. Attach an action listener to a menu item instead.

Figure 50 shows a pop-up menu and Figure 51 shows the matching source code.

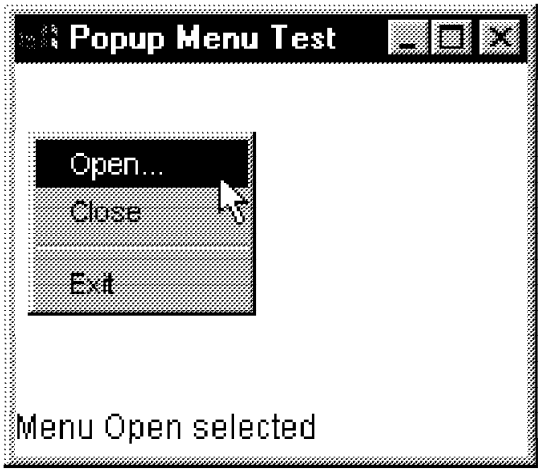


Figure 50. Pop-up Menu Sample Application

```
/* gui\popupmenu\Popup.nrx

   This program shows the usage of a popup menu */

---------------------------------------- Popup class (implicit)
   MenuBarX()

---------------------------------------- MenuBarX class
class MenuBarX
  Properties inheritable
    status = Label('Press the right mouse button')

  method MenuBarX()
    win = Frame("Popup Menu Test")    -- creates the window

    m  = PopupMenu("File")            -- first menu in the menubar

    mi = MenuItem("Open...")          -- menu item for menu File
    m.add(mi)                         -- add menu item to menu File
    mi.addActionListener(MenuAction(status,"Open")) -- menu action

    mi = MenuItem("Close")            -- next menu item for menu File
    m.add(mi);
    mi.addActionListener(MenuAction(status,"Close")) -- menu action

    m.addSeparator();                 -- add a separator to the menu

    mi = MenuItem("Exit")             -- next menu item for menu File
    m.add(mi);
    mi.addActionListener(CloseMenu()) -- menu action

    win.add(m)                        -- add the popup menu to the window

                                      -- add the mouse event listener
    status.addMouseListener(PopUpMouseListener(m))
    win.addMouseListener(PopUpMouseListener(m))
```

Figure 51 (Part 1 of 2). Pop-up Menu Sample Application: Popup.nrx

```
    win.add("South",status)          -- add the label to the window

                                     -- set window position to the middle
    win.setSize(200,130)
    d = win.getToolkit().getScreenSize()
    s = win.getSize()
    win.setLocation((d.width - s.width) % 2,(d.height - s.height)%2)

    win.setVisible(1)                -- show the window

------------------------------------- PopUpMouseListener class
class PopUpMouseListener extends MouseAdapter
  Properties
    mpopup = PopupMenu
  method PopUpMouseListener(p=PopupMenu)
    mpopup = p

  -- look for popup trigger event when a mouse button is pressed
  method mousePressed(e=MouseEvent)
    if e.isPopupTrigger() then
      mpopup.show(e.getComponent(),e.getX,e.getY)

  -- look for popup trigger event when a mouse button is released
  method mouseReleased(e=MouseEvent)
    if e.isPopupTrigger() then
      mpopup.show(e.getComponent(),e.getX,e.getY)

------------------------------------- MenuAction class
class MenuAction implements ActionListener
  Properties inheritable
    text        = String
    field       = Label

  method actionPerformed(e = ActionEvent)
    field.setText('Menu' text 'selected')

  method MenuAction(aLabel = Label, aString = String)
    field       = aLabel
    text        = aString

------------------------------------- CloseMenu class
class CloseMenu implements ActionListener
  method actionPerformed(e = ActionEvent)
    exit 0
```

Figure 51 (Part 2 of 2). Pop-up Menu Sample Application: Popup.nrx

Layout Manager

A layout manager is a class that arranges the child components of a container. Containers are components which hold and include other components. Frame windows, applets, and dialog windows are containers.

The layout manager resizes or moves the components according to the layout rule of the manager, to fit the components into the available space.

A layout manager resizes and rearranges your components when the window is resized by the user, or when the pack method (of the Window class) is invoked. There is no longer a need to create fixed-size windows when using a layout manager.

Using the setLayoutManager method, you can change the layout manager of a container:

```
window.setLayoutManager(BorderLayout())
```

Every container has a default layout manager, windows defaulting to the BorderLayout manager, and panels to the FlowLayout manager. You can get the current layout manager with the getLayout method.

Panels are components and containers, so you can add panels to panels to any depth. Panels are very well suited if another layout manager is appropriate for a part of the window frame.

A layout manager uses two properties of the components within a container to arrange its layout: the minimum size and the preferred size. The minimum size of a component specifies the minimum amount of space the component needs. The preferred size of a component can reflect the size when a component looks good.

If one of your components changes its size on its own, or if you change the layout manager at run time, you must invoke the invalidate method and then the validate method of the container.

FlowLayout

The FlowLayout manager places its components from left to right and top to bottom. If there is no space in the current row, the component is placed in the next row. The FlowLayout manager uses the preferred size of the components to calculate the space needed. Figure 52 shows an applet with a FlowLayout manager in two sizes.

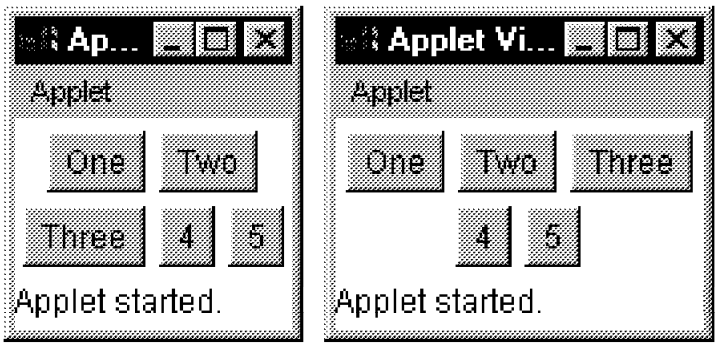


Figure 52. FlowLayout Manager

The FlowLayout manager has an alignment property that specifies how the remaining space of a row is used. If the alignment is set to LEFT or RIGHT, the remaining space is moved to the right or left end of a row. If the alignment is set to CENTER the remaining space is divided by 2 and distributed to the left and right end of the row.

The default alignment of a FlowLayout manager is CENTER.

The components are separated by a horizontal and a vertical gap. The gap is also added to the borders of the container. The default for the gaps is 5 pixels.

Constants, constructors, and methods of interest:

Constants

LEFT Used for left alignment

RIGHT Used for right alignment

CENTER Used for center alignment

Constructors

FlowLayout(align=int CENTER)
Creates a FlowLayout manager with horizontal and vertical gaps set to five pixels.

FlowLayout(align=int, hgap=int, vgap=int)
Creates a FlowLayout manager with the specified parameters

Methods

setAlignment(align=int)
Sets the alignment of the FlowLayout manager

setHgap(hgap=int)
Sets the horizontal gap between the columns

setVgap(vgap=int)
Sets the vertical gap between the columns

Figure 53 shows the code for the applet shown in Figure 52 on page 97. The buttons are added to the applet's default FlowLayout manager.

```
/* gui\flowlayout\FlowLay.nrx

   Sample Applet to illustrate the FlowLayout manager */

class FlowLay extends Applet

method init()
  setLayout(FlowLayout(FlowLayout.CENTER))  -- not necessary (default for applets)
  add( Button('One'))
  add( Button('Two'))
  add( Button('Three'))
  add( Button('4'))
  add( Button('5'))
```

Figure 53. FlowLayout Manager Sample: FlowLay.nrx

BorderLayout

The BorderLayout manager places its components in five places. Figure 54 shows an applet with a BorderLayout manager.

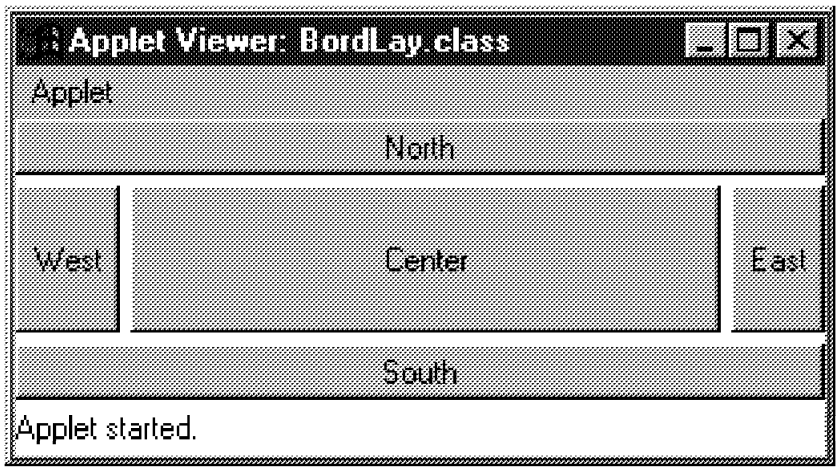


Figure 54. BorderLayout Manager

The places are named *North*, *West*, *Center*, *East*, and *South*. When a component is added to the container, one of these five names must be used:

```
window.add('South',Button('button-text'))
```

The *North* and *South* components are stretched by the BorderLayout manager in a horizontal direction. The vertical size of the components is not changed.

The *West* and *East* components are stretched in a vertical direction. The horizontal size of the components is not changed.

The *Center* is stretched in both directions to fill the remaining space.

If a location is not filled by a component, or the component is invisible during the layout calculation, the space is used by the other components.

The horizontal gap specifies the space between the *West*, *East*, and *Center* components. The vertical gap specifies the space added to the bottom of the *North* and to the top of the *South* component. The default value for the gap is 0.

Constructors and methods of interest:

Constructors

BorderLayout()
: Creates a default BorderLayout manager with horizontal and vertical gaps set to 0 pixels

FlowLayout(hgap=int, vgap=int)
: Creates a BorderLayout manager with gaps set to the specified parameters

Methods

setHgap(hgap=int)
: Sets the horizontal gap between the components

setVgap(vgap=int)
: Sets the vertical gap between the components

Figure 55 shows the code for the applet shown in Figure 54 on page 98.

```
/* gui\borderlaypout\BordLaynrx

   Sample Applet to illustrate the BorderLayout manager */

class BordLay extends Applet

method init()
  setLayout(BorderLayout(4,4))
  add( 'North',  Button('North') )
  add( 'South',  Button('South') )
  add( 'West',   Button('West') )
  add( 'East',   Button('East') )
  add( 'Center',Button('Center') )
```

Figure 55. BorderLayout Manager Sample: BordLay.nrx

Note: The BorderLayout is very useful for the top panel of a frame with subpanels placed into the available positions. The subpanels use their own layout manager to arrange individual components.

GridLayout

The GridLayout manager places its components into a number of fixed rows and columns. The components are resized to fit in the resulting areas; their minimum and preferred sizes are ignored. Figure 56 on page 100 shows an applet with two GridLayout managers.

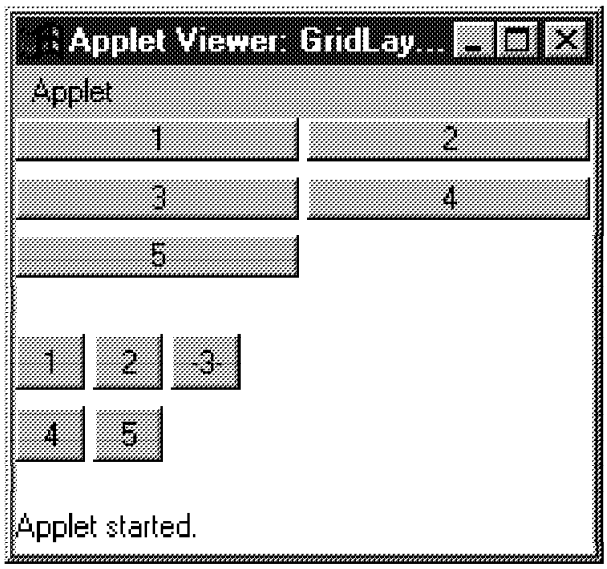


Figure 56. GridLayout Manager

The applet is divided in two parts. The upper part consists of a GridLayout manager that uses all of the available space. The lower part consists of a FlowLayout manager with a panel with a GridLayout manager. This configuration forces a GridLayout manager to size to its preferred size.

Note: If a GridLayout manager is set to its preferred size, all of its components are sized equal to the largest component. This is very useful for creating buttons of equal size, according to the button with the longest text.

Remember that a panel with a FlowLayout manager sets the components to their preferred size.

The GridLayout manager works with rows and columns. The components are placed from left to right and top to bottom.

If the number of rows is greater than zero, any column specification is ignored. The number of components is divided by the rows and rounded to the next greater number. A GridLayout manager with seven components set in three rows will have three columns, with two empty places, of course.

If the number of rows is set to 0, the specified number of columns will be used. The number of components is divided by the columns and rounded to the next greater number.

If the minimum count of elements is not available, the space remains free.

Horizontal and vertical gaps can be used to add additional space between the columns or rows. The gap is set to 0 by default.

Constructors and methods of interest:

Constructors

GridLayout()
: Creates a default GridLayout manager with one row. The horizontal and vertical gaps are set to 0 pixels.

GridLayout(row=int, column=int)
Creates a GridLayout manager with the defined gaps

GridLayout(row=int,column=int,hgap=int,vgap=int)
Creates a GridLayout manager with the specified parameters

Methods

setHgap(hgap=int)
Sets the horizontal gap between the columns

setVgap(vgap=int)
Sets the vertical gap between the columns

Figure 57 shows the code for the applet shown in Figure 56 on page 100.

```
/* gui\gridlayout\GridLay.nrx

   Sample Applet to illustrate the GridLayout manager */

class GridLay extends Applet

method init()
  -- divide the space of the applet in two rows with 20 pixels space between
  setLayout(GridLayout(2,0,0,20))

  -- add a panel with a GridLayout manager to row 1
  -- this layout manager resizes the components to use the available space
  p=Panel()    -- new panel
  add(p)       -- add panel
  p.setLayout(GridLayout(3,0,3,6)) -- 3 rows hgap=3 vgap=6
  p.add(Button('1'))
  p.add(Button('2'))
  p.add(Button('3'))       -- next row
  p.add(Button('4'))
  p.add(Button('5'))       -- next row

  -- Add a panel with a FlowLayout manger to row two to avoid resizing
  pFlow=Panel()
  pFlow.setLayout(FlowLayout(FlowLayout.LEFT,0,0))  -- left alignment no gaps
  add(pFlow)

  -- Add a panel with a GridLayout manager to the panel with the flowlayout
  p=Panel()    -- new Panel
  pFlow.add(p) -- add Panel

  p.setLayout(GridLayout(0,3,3,6)) -- 3 columns hgap=3 vgap=6
  p.add(Button('1'))
  p.add(Button('2'))
  p.add(Button('-3-')) -- this button it's the largest, sets the size for all
  p.add(Button('4'))
  p.add(Button('5'))
```

Figure 57. GridLayout Manager Sample: GridLay.nrx

GridBagLayout

The GridBagLayout manager is the most complicated layout manager of the JDK. The GridBagLayout manager arranges its components in a grid of cells. Each cell in a column has the same width, and each cell in a row has the same height. Columns or rows do not have to be equal in size, which is different from the GridLayout manager.

Figure 58 shows an example of an applet using a GridBagLayout manager.

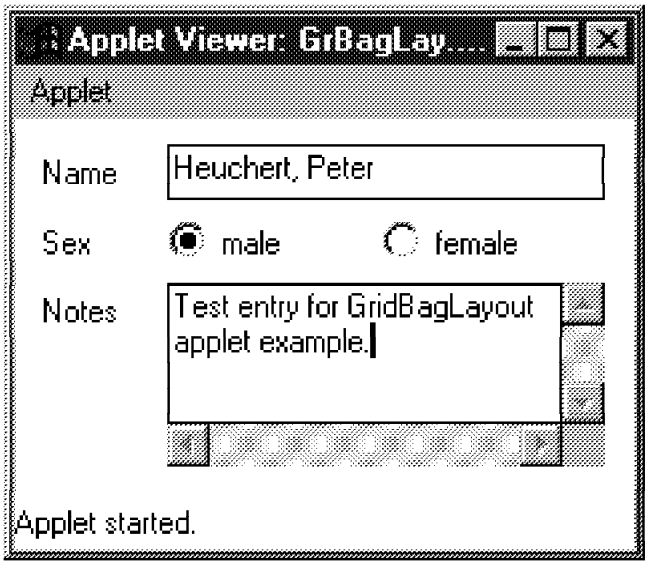


Figure 58. GridBagLayout Manager

The cells in the grid are addressed by their x and y indexes. The origin of the grid is in the upper-left corner of the container and starts with cell (0,0). The size of the grid is automatically expanded. If you add a cell to (9,7), the grid size becomes 10 rows and 8 columns.

A component can be added to any cell of the grid and can span over multiple rows and columns. The area a component occupies is called the *component display area*.

Not every cell must be occupied.

The component's location and other information are defined by a GridBagConstraints object. Every component in a GridBayLayout manager is associated with a GridBagConstraints object.

The GridBagConstraints class defines seven constraints. The constraints are public instance variables of a GridBagContraints object:

Position The *gridx* and *gridy* constraints define the cell that the component occupies. If a component occupies more than one cell, *gridx* and *gridy* define the upper-left corner in the grid. Remember that *gridx* and *gridy* are 0-based.

Size The *gridwidth* constraint defines the number of cells a component occupies in a horizontal direction (from left to right).

The *gridheight* constraint defines the number of cells a component occupies in a vertical direction (from top to bottom).

Fill The *fill* constraint defines if and in which direction a component is stretched to the size of the component display area:

- NONE (default)
- BOTH
- VERTICAL

- HORIZONTAL

Anchor If a component is not stretched to the full size of the component display area, the *anchor* defines the component's position as shown in Figure 59.

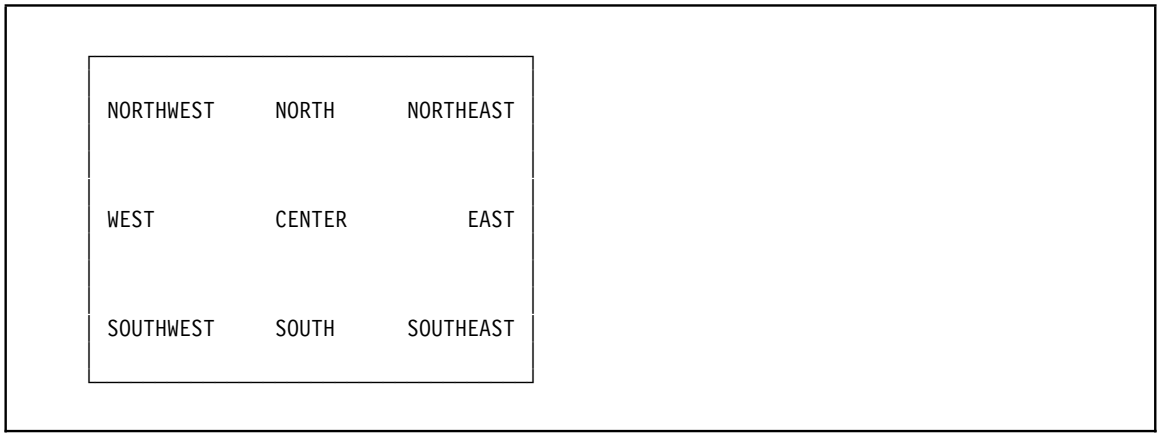


Figure 59. GridBagLayout: Anchor Constraint

CENTER is the default.

Insets The *insets* constraint affects the size of the component's display area. A positive value reduces the size and a negative value increases the size of the component display area.

The *insets* constraint is an object of type Insets. When a GridBagConstraints object is constructed, an Insets object is created and assigned to the insets constraint.

An Insets object has four public instance variables: top, left, bottom, and right. All are set to 0 as the default. It is not necessary to create a new Insets object, all values can be easily modified:

```
-- gbc is a GridBagConstraints object
gbc.insets.top = 5 -- reduces the component display area about 5 pixels
gbc.insets.left = 2
```

Weights The *weightx* and *weighty* constraints control how additional space is distributed to the cells.

When the container is larger than the preferred size of the GridBagLayout manager, which is based on the preferred size of its components, the additional space is distributed to the cells.

The GridBagLayout manager calculates the sum of all weights of a row. The additional space is divided by the sum of the weights and then distributed to the cells in proportion to the weights.

If a cell has a weight of 0, no additional space is added to the cell. The additional space is added to the whole row or whole column. If cells in the same row have different vertical weights, the largest weight is used for the whole row. The default value of the weights is 0.

The grid is centered in the container, if the sum of the weights is 0.

Internal Padding

When the GridBagLayout manager calculates the minimum or preferred size of the container, it invokes the getMinimumSize or getPreferredSize method of the components. The internal Padding constraints, *ipadx* and *ipady*, are added to the returned size of the component. Both can be positive or negative and default to 0 pixels.

How to Use a GridBagLayout Manager

Follow these simple instructions to use the GridBagLayout manager:

1. Sketch the layout of your applet or frame on a piece of paper as in Figure 60.

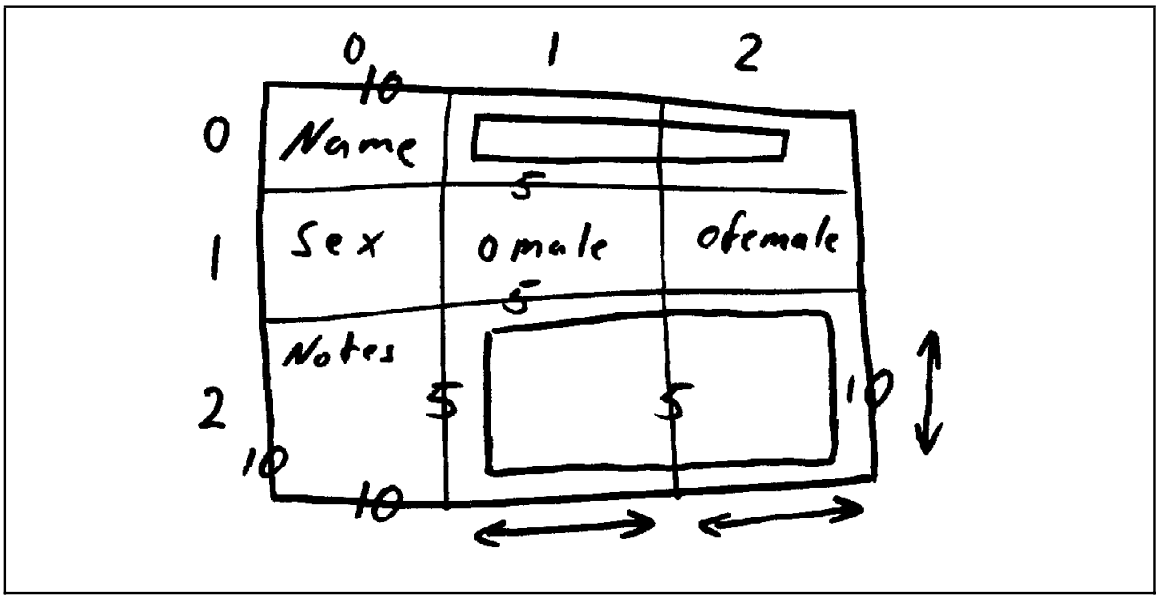


Figure 60. Sketch for GridBagLayout Manager Example

2. Number the rows and columns and mark the rows and columns that should receive additional space.
3. Give every row or column that should be expandable the weight of 1.
4. Use multiples of 1 for the weight to define the proportion of expansion of multiple expandable rows or columns.
5. Decide whether a component should be expanded (fill constraint) and mark the anchor if the fill constraint is not set to BOTH.
6. If you want some space between the components, define the insets for the rows and columns. Be consistent with the insets. Write the insets on the lines of your grid.
7. Start coding; add a GridBagLayout manager to the container:

```
-- window is a frame window and the container
gbl = GridBagLayout()
window.setLayout(gbl)
```

8. Create a GridBagConstraints object and preset it with your common constraints:

```
tgbc = GridBagConstraints()             -- template object for later use
tgbc.anchor = GridBagConstraints.NORTHWEST
tgbc.insets.top = 5
tgbc.insets.left = 4
...
```

If you add a *uses GridBagConstraints* phrase to your class instruction (see “Class Instruction” on page 25), you can use all of the constants without the class name:

```
class MyGui uses GridBagConstraints
  ...
  tgbc.anchor = NORTHWEST
  ...
```

9. Perform the following three steps for every component:

   a. Clone the previously created GridBagConstraints object and change the constraints that are different:

      ```
      -- tgbc is the template GridBagConstraints object
      gbc = GridBagConstraints tgbc.clone()
      gbc.gridx = 2;  gbc.gridy = 0
      ...
      ```

      The cast to type GridBagConstraints is necessary because the clone method returns a type of void.

   b. Add the component to the container.

      ```
      -- window is a frame window and the container
      -- comp is the component
      window.add(comp)
      ```

   c. Set the GridBagConstraints in the GridBagLayout manager:

      ```
      -- gbl is a GridBagLayout manager object
      -- gbc is a GridBagConstraints object
      gbl.setConstraints(comp, gbc)
      ```

Figure 61 shows the code for the GridBayLayout applet shown in Figure 58 on page 102.

```
/* gui\gridbaglayout\GrBagLay.nrx

   Sample Applet to illustrate the GridBagLayout manager */

class GrBagLay extends Applet uses GridBagConstraints

method init()
  gbl  = GridBagLayout()
  t1gbc = GridBagConstraints()   -- for labels
  t2gbc = GridBagConstraints()   -- for the other components
  setLayout(gbl)

  -- Preset t1gbc  used for the labels
  t1gbc.anchor=NORTHWEST
  t1gbc.insets.top=5; t1gbc.insets.left=10; t1gbc.insets.right=5
  t1gbc.gridx = 0

  -- Preset t2gbc  used for the other components
  t2gbc.fill = BOTH;
  t2gbc.insets.top=5; t2gbc.insets.right=10
  t2gbc.weightx = 1;  t2gbc.gridx = 1

  -- Add the labels
  l = Label('Name')
  gbc=GridBagConstraints t1gbc.clone(); gbc.gridy=0; ; gbc.insets.top=10
  add(l); gbl.setConstraints(l,gbc)

  l = Label('Sex')
  gbc=GridBagConstraints t1gbc.clone(); gbc.gridy=1;
  add(l); gbl.setConstraints(l,gbc)
```

Figure 61 (Part 1 of 2). GridBagLayout Manager Sample: GrBagLay.nrx

```
l = Label('Notes')
gbc=GridBagConstraints t1gbc.clone(); gbc.gridy=2; gbc.insets.bottom=10
add(l); gbl.setConstraints(l,gbc)

-- Add the other components
name = TextField()
gbc=GridBagConstraints t2gbc.clone(); gbc.gridy=0; gbc.gridwidth=REMAINDER
gbc.insets.top=10
add(name); gbl.setConstraints(name,gbc)

cbg  = CheckboxGroup()  -- Checkbox group because exclusive choices
cb   = Checkbox('male',cbg,1)
gbc=GridBagConstraints t2gbc.clone(); gbc.gridy=1; gbc.insets.right=5
add(cb); gbl.setConstraints(cb,gbc)

cb   = Checkbox('female',cbg,0)
gbc=GridBagConstraints t2gbc.clone(); gbc.gridy=1; gbc.gridx=2;
add(cb); gbl.setConstraints(cb,gbc)

area = TextArea(5,20) -- 5 rows and 20 columns visible
gbc=GridBagConstraints t2gbc.clone(); gbc.gridy=2; gbc.gridwidth=REMAINDER
gbc.weighty = 1
gbc.insets.bottom=10
add(area); gbl.setConstraints(area,gbc)
```

Figure 61 (Part 2 of 2). GridBagLayout Manager Sample: GrBagLay.nrx

The usage of the GridBagLayout manager requires a lot of coding. The subclass of the GridBagLayout manager shown in Figure 62 reduces the coding by providing methods designed for convenient usage.

```
/* redbook\gui\SimpleGridBagLayout.nrx

   Simple GridBagLayout class for easy coding */

package Redbook

class SimpleGridBagLayout extends GridBagLayout uses GridBagConstraints
  Properties inheritable
    theContainer = Container       -- Container used with this layout manager
    dInsets      = Insets(0,0,0,0) -- Template insets
    dAnchor      = int NORTHWEST

  method SimpleGridBagLayout( aContainer = Container )
    super()
    theContainer = aContainer
    aContainer.setLayout(this)

  method setInsets(top=int,left=int,bottom=int,right=int)
    dInsets.top    = top
    dInsets.left   = left
    dInsets.bottom = bottom
    dInsets.right  = right

  method setAnchor(newAnchor=int)
    dAnchor=newAnchor
```

Figure 62 (Part 1 of 2). SimpleGridBagLayout Manager Class: SimpleGridBagLayout.nrx

```
  method newConstraints(x=int,y=int,sizex=int,sizey=int,fill=int,-
                        anchor=int,weightx=double,weighty=double)-
         returns GridBagConstraints
    gbc = GridBagConstraints()
    gbc.gridx = x; gbc.gridy = y;
    gbc.gridwidth = sizex; gbc.gridheight=sizey
    gbc.fill = fill; gbc.anchor = anchor;
    gbc.weightx = weightx; gbc.weighty = weighty
    return gbc

  method addFixSize(comp=Component,x=int,y=int) returns Component
    addVarSize(comp,x,y,Insets dInsets.clone(),0.0,0.0,1,1,NONE,dAnchor)
    return comp

  method addFixSize(comp=Component,x=int,y=int,newInsets=Insets,-
              sizex=int 1,sizey=int 1,-
              fill=int NONE, anchor=int NORTHWEST) returns Component
    addVarSize(comp,x,y,newInsets,0.0,0.0,sizex,sizey,fill,anchor)
    return comp

  method addVarSize(comp=Component,x=int,y=int,newInsets=Insets,weightx=double,-
              weighty=double,sizex=int 1,sizey=int 1,-
              fill=int BOTH, anchor=int NORTHWEST) returns Component
    gbc = newConstraints(x,y,sizex,sizey,fill,anchor,weightx,weighty)
    gbc.insets = Insets newInsets
    theContainer.add(comp)
    setConstraints(comp,gbc)
    return comp

  method addVarSize(comp=Component,x=int,y=int,weightx=double,weighty=double,-
              sizex=int 1,sizey=int 1,-
              fill=int BOTH, anchor=int NORTHWEST) returns Component
    addVarSize(comp,x,y,Insets dInsets.clone(),weightx,weighty,-
                sizex,sizey,fill,anchor)
    return comp
```

Figure 62 (Part 2 of 2). SimpleGridBagLayout Manager Class: SimpleGridBagLayout.nrx

Note that the class is part of the *Redbook* package. It must be stored in a subdirectory named “Redbook,” which is part of a subdirectory listed in the *CLASSPATH*.

With this class, the sample applet can be simplified quite a bit (see Figure 63).

```
/* gui\simplegridbaglayout\GrBagLa2.nrx

   Sample Applet to illustrate the GridBagLayout manager */

import Redbook.

class GrBagLa2 extends Applet uses GridBagConstraints

method init()
  sgbl  = SimpleGridBagLayout(this)

  -- Add the labels
  sgbl.addFixSize(Label('Name'),0,0,Insets(10,10,5,5))
  sgbl.addFixSize(Label('Sex'),0,1,Insets(0,10,5,5))
```

Figure 63 (Part 1 of 2). GridBagLayout Manager Sample—Simplified: GrBagLa2.nrx

```
    sgbl.addFixSize(Label('Notes'),0,2,Insets(0,10,10,5))

    -- Add the other components
    cbg  = CheckboxGroup()  -- Checkbox group because exclusive choices
    sgbl.addVarSize(TextField(),1,0,Insets(10,0,5,10),1.0,0.0,REMAINDER)
    sgbl.addVarSize(Checkbox('male',cbg,1),1,1,Insets(0,0,5,5),1.0,0.0)
    sgbl.addVarSize(Checkbox('female',cbg,1),2,1,Insets(0,0,5,10),1.0,0.0)
    sgbl.addVarSize(TextArea(5,20),1,2,Insets(0,0,10,10),1.0,1.0,REMAINDER)
```

Figure 63 (Part 2 of 2). GridBagLayout Manager Sample—Simplified: GrBagLa2.nrx

CardLayout

The CardLayout manager shows only one component at a time. The visible component is resized to the size of the container. The other components are hidden. The visible components are usually panels that themselves contain more components.

The CardLayout manager can be used to implement user interfaces such as the notebook in OS/2 or Windows.

Figure 64 shows an applet with a CardLayout manager.

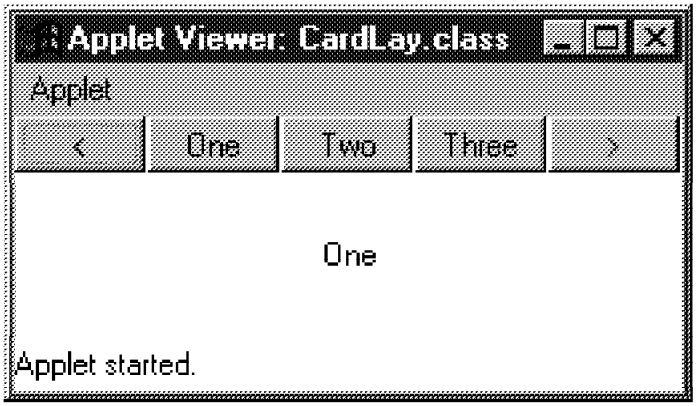


Figure 64. CardLayout Manager

The CardLayout manager controls the visibility of the components added to it. Several methods are available to change the current component, that is, the visible component.

When the components are added to the container, a label must be used to identify the component. The label can be used with the CardLayout manager to make the component current.

Use the following method to add the components to the container:

```
add(LABEL=String,COMP=Component)
```

The horizontal and vertical gaps reduce the size of the components managed by the CardLayout manager.

Constructors and methods of interest:

Constructors

CardLayout()
: Creates a CardLayout manager with the gaps set to 0

Methods

setHgap(hgap=int)
: Sets the horizontal gap between the columns

setVgap(vgap=int)
: Sets the vertical gap between the columns

Figure 65 shows the code for the CardLayout manager applet shown in Figure 64 on page 108.

```
/* gui\cardlayout\CardLay.nrx

   Sample Applet to illustrate the CardLayout manager */

class CardLay extends Applet

method init()
  setLayout(BorderLayout())  -- To show the buttons and the CardLayout manager

  -- GridLayout to store the button list
  pgrid = Panel(); grid = GridLayout(1,0,1,0)
  pgrid.setLayout(grid)

  -- This is the CardLayout manager
  pcard = Panel();
  pcard.setLayout(CardLayout())

  -- create the buttons and add the action listener
  b= Button('<');     b.addActionListener(BList('prev',pcard)); pgrid.add(b)
  b= Button('One');   b.addActionListener(BList('One',pcard));  pgrid.add(b)
  b= Button('Two');   b.addActionListener(BList('Two',pcard));  pgrid.add(b)
  b= Button('Three'); b.addActionListener(BList('Three',pcard));pgrid.add(b)
  b= Button('>');     b.addActionListener(BList('next',pcard)); pgrid.add(b)

  -- add the panels to the applet
  add("North",pgrid)
  add("Center",pcard)

  -- add the components to the CardLayout manager
  pcard.add('One',Label('One',Label.CENTER))
  pcard.add('Two',Label('Two',Label.CENTER))
  pcard.add('Three',Label('Three',Label.CENTER))

-- The BList class is needed to switch the panels
class BList implements ActionListener
  Properties inheritable
    cont  = Container
    aLabel = String

  method BList(theLabel=String,aContainer=Container)
    cont = aContainer; aLabel = theLabel

  method actionPerformed(e=ActionEvent)
    select
      when aLabel='next' then
        (CardLayout cont.getLayout()).next(cont)
      when aLabel='prev' then
        (CardLayout cont.getLayout()).previous(cont)
```

Figure 65 (Part 1 of 2). CardLayout Manager Sample: CardLay.nrx

```
        otherwise
          (CardLayout cont.getLayout()).show(cont,aLabel)
      end
```

Figure 65 (Part 2 of 2). CardLayout Manager Sample: CardLay.nrx

A second version of the same program, CardLay2.nrx, disables the button that matches the current page.

Frame and Dialog Windows

Frame and dialog windows do not only apply to Java applications. Any applet can also create a frame or a dialog window. Frames and dialogs are subclasses of the Window class. The only difference is that a warning string is added to any window an applet creates. The reason for the warning string is that an applet could present you with a window like a login window and ask you to enter a userid and password. With the warning string in the window, anyone can see that the window shown is not a standard login window.

Every window, dialog or frame, is invisible when constructed. You must use the setVisible method to make them visible. Before making a window visible, you have to set the size of the window. There are two ways of setting the size:

- Set the size to a fixed value with the setSize method:

  ```
  win.setSize(200,100)
  ```

  The first parameter is the width, and the second is the height of the window in pixels.

- Calculate the preferred size with the pack method. The pack method cannot return proper results if you did not set the visible columns for TextFields, or the number of rows in lists, for example. The advantage of the pack method is that you are no longer dependent on the fonts of different operating systems.

Frame Windows

A frame window has a title bar that includes the system menu, and it may have a menu bar. The Frame class implements frame windows:

```
Frame(title = String '')
```

The z-order of frame windows is not set, because no information about the parent window is available.[1] Every frame window can be set to the top, unless it has dialog windows.

Dialog Windows

A dialog window is used to get input from the user. Dialog windows have a title bar, but no system menu, and they cannot include menu bars.

A dialog window needs a frame window as the parent for the constructor:

```
Dialog(parent=Frame, modal=boolean)
Dialog(parent=Frame, title = String '', modal=boolean 0)
```

[1] The z-order is the third dimension of your screen. The window with the highest z-order is on top of all the other windows. The window with the lowest z-order is your desktop.

A dialog window always stays on top of its parent window.

If the dialog window is modal, the parent window cannot be accessed as long as the dialog window exists. It is not possible to set the visibility of a dialog window to false; the dialog window must be destroyed.

Tabbing Support

Frame and dialog windows support tabbing. The tabbing sequence is directly related to the sequence in which components are added to the window. Tabbing works in every depth, regardless of how many panels and layout managers are used.

Any component that returns 1 (true) when the isTabbable method is invoked can accept keyboard focus.

Event Handling

Event handling changed dramatically with JDK 1.1. With the new event handling, *delegation event handling*, subclassing of components is no longer necessary in most situations.

Delegation event handling has many advantages:

- Only the events you are really interested in must be handled by your code
- A clear separation between the GUI and controlling code is possible
- It prevents programming errors.
- It is easy to learn.

The old event handling of JDK 1.0 is still supported but must be separated from the new event handling at the component level.

The delegation event handling model encapsulates all events in classes that are subclasses of the EventObject class. The events are propagated from the components (event sources) to event listeners. The event source *fires* events, and the listener *receives* events.

An event listener implements an interface specific for the event it will receive. The listener interface defines one or more methods, which are invoked by the event sources.

The event source has a list of registered listeners for every event type. If an event occurs, the registered event listeners are called. The listeners are called synchronously, so that the next listener is called when the previous listener returns from the event handling method. The sequence in which the event listeners are called is not specified.

The only exception to the rule above is that event listeners can consume events. If an event is consumed by a listener, no further event handling occurs. An event is consumed by calling the consume method, which is defined in the AWTEvent class. Consuming events is useful when you are writing keyboard handlers or creating GUI builders.

Events

The events are represented by a hierarchy of event classes. Each class provides the data related to the event.

The event classes do not define any public instance variable. The data of an event can be accessed through *getAttribute* and possibly *setAttribute* methods.

You can expand the event hierarchy by defining your own event types.

The JDK distinguishes between two event types: low-level events and semantic events.

Low-Level Events

Low-level events represent low-level input and actions on a visual component of the GUI. The following low-level event classes are available:

ComponentEvent
Fired when a component is shown, hidden, moved, or resized

FocusEvent
Fired when a component gets or loses the focus

KeyEvent Fired from a component, which owns the keyboard focus, when a key is pressed, released, or typed (combination of key pressed and released)

MouseEvent
Fired when the mouse is moved or dragged or the mouse buttons are pressed

ContainerEvent
Fired when a component is added or removed from a container

WindowEvent
Fired when a window is opened, closed, iconified, deiconified, activated, or deactivated

Low-Level events have multiple event types per event class.

Semantic Events

Semantic events encapsulate the semantics of the GUI. Event sources of semantic events do not have to be components. A nonvisual time class, for example, can also fire an ActionEvent, as can buttons or list boxes.

The main difference between low-level events and semantic events is that a semantic event describes a single action that is on a higher level than simple input events.

The following semantic event classes are available:

ActionEvent
Fired when a command is given in buttons, lists, choices, text fields, and text areas

AdjustmentEvent
Fired when an adjustable value has changed, for example, in scroll bars

ItemEvent Fired when a selection of an item occurs in lists, choices, and check boxes

TextEvent Fired when a monitored text changes in text fields and text areas

Semantic events have only one event type per event class.

Event Listener Interface

An event listener interface is defined for every event class. The interface defines a separate method for every distinct event type the event class represents.

The event listener interfaces define a balance between providing a separate method for any single event and one method for all event types.

Use the following method to add an event listener to a component:

addEventListener(*eventListenerObject*)

EventListener is a place holder for the name of an event listener interface. For example, to add a WindowListener object to a window use the following command:

```
-- win     is an object of class Frame
-- winlist is an object of class WindowListener
win.addWindowListener(winlist)
```

An object of class WindowListener indicates that the class of the object implements the WindowListener interface.

Low-Level Listener Interfaces

The methods of the low-level event interfaces are:

ComponentListener

componentHidden(e=ComponentEvent)
: Invoked when component has been hidden (with setVisibility(0), for example)

componentMoved(e=ComponentEvent)
: Invoked when component has been moved

componentResized(e=ComponentEvent)
: Invoked when component has been resized

componentShown(e=ComponentEvent)
: Invoked when component has been shown (with setVisibility(1), for example)

ContainerListener

componentAdded(e=ContainerEvent)
: Invoked when a component has been added to the container

componentRemoved(e=ContainerEvent)
: Invoked when a component has been removed from the container

FocusListener

focusGained(e=FocusEvent)
: Invoked when a component gains the keyboard focus

focusLost(e=FocusEvent)
: Invoked when a component loses the keyboard focus

KeyListener

keyPressed(e=KeyEvent)
: Invoked when a key has been pressed

keyReleased(e=KeyEvent)
: Invoked when a key has been released

keyTyped(e=KeyEvent)
: Invoked when a key has been typed. This event occurs when a key press is followed by a key release.

MouseListener

The mouse events are split into two listeners. The splitting enables an application to react to some mouse events, without the need to receive messages from mouse movement.

mouseClicked(e=MouseEvent)
: Invoked when the mouse has been clicked on a component

mousePressed(e=MouseEvent)
: Invoked when a mouse button has been pressed on a component

mouseReleased(e=MouseEvent)
: Invoked when a mouse button has been released on a component

mouseEntered(e=MouseEvent)
: Invoked when the mouse enters a component

mouseExited(e=MouseEvent)
: Invoked when the mouse exits a component

MouseMotionListener

mouseDragged(e=MouseEvent)
: Invoked when a mouse button is pressed on a component and then dragged. Mouse drag events continue to be delivered to the component where the drag originated until the mouse button is released (regardless of whether the mouse position is within the bounds of the component).

mouseMoved(e=MouseEvent)
: Invoked when the mouse has been moved on a component (with no buttons pressed)

WindowListener

windowActivated(e=WindowEvent)
: Invoked when a window is activated, or the focus returned to the window or any of its components

windowClosed(e=WindowEvent)
: Invoked when a window has been closed. The event is delivered only when the window was destroyed with the dispose method.

windowClosing(e=WindowEvent)
: Invoked when a window is in the process of being closed. The event is delivered when the user selects **Close** from the window's system menu. If the program does not explicitly hide or destroy the window as a result of this event, the window close operation is canceled.

windowDeactivated(e=WindowEvent)
: Invoked when a window is deactivated. A window is deactivated when the focus goes to another window. This is like a focus lost event at the window level.

windowDeiconified(e=WindowEvent)
: Invoked when a window is deiconified

windowIconified(e=WindowEvent)
: Invoked when a window is iconified

windowOpened(e=WindowEvent)
: Invoked when a window has been opened. The event is delivered when a window is shown for the first time.

Semantic Listener Interfaces

The semantic listener interfaces are characterized by defining only one method per interface:

ActionListener

actionPerformed(e=ActionEvent)
: Invoked when an action occurs, that is, when a user clicks a button, double-clicks on an item in a list box, or presses the *Enter* key in a text field.

AdjustmentListener

adjustmentValueChanged(e=AdjustmentEvent)
: Invoked when the adjustable value has changed

ItemListener

itemStateChanged(e=ItemEvent)
: Invoked when an item's state has been changed. If a list box allows multiple selection, the method is invoked for selection and deselection of an item. Otherwise the method is invoked for selection of items only.

TextListener

textValueChanged(e=TextEvent)
: Invoked when the text of a TextField or TextArea object has changed

Adapters

An adapter is an implementation of a low-level listener interface. All methods defined in the interface are implemented. The default implementation in the adapter methods is to return without an action.

Adapter classes are abstract classes.

Adapters are convenient programmer shortcuts. You can easily subclass an adapter class and overwrite only the methods you are interested in, without having to implement all of the methods defined by the interface.

The following adapters are available:

- ComponentAdapter
- FocusAdapter
- KeyAdapter
- MouseAdapter
- MouseMotionAdapter
- WindowAdapter

Event and Component Cross Reference

Table 8 shows which events are fired from the GUI components.

Table 8. Event and Component Cross Reference

Class	Low-Level Events							Semantic Events			
	ComponentEvent	ContainerEvent	FocusEvent	KeyEvent	MouseEvent	MouseMotionEvent	WindowEvent	ActionEvent	AdjustmentEvent	ItemEvent	TextEvent
Component	X		X	X	X	X					
Container	X	X	X	X	X	X					
Dialog	X	X	X	X	X	X	X				
Frame	X	X	X	X	X	X	X				
Button	X		X	X	X	X		X			
Choice	X		X	X	X	X				X	
Checkbox	X		X	X	X	X				X	
CheckboxMenuItem	X		X	X	X	X				X	
List	X		X	X	X	X				X	
MenuItem	X		X	X	X	X		X			
Scrollbar	X		X	X	X	X			X		
Textarea	X		X	X	X	X					X
Textfield	X		X	X	X	X		X			X

Fonts

Java defines a platform-independent interface to work with fonts. Fonts are used when painting text on a graphics context (see “Images” on page 118) or as a property for components.

A font has three properties: font name, font style, and point size.

The font name must be chosen from (currently) six logical font names that are mapped to the system fonts on each operating system:

- Helvetica
- TimesRoman
- Courier
- Dialog
- DialogInput
- Symbol

If you specify an invalid name, an operating-system-dependent default is used.

To create a font object, you must use the constructor of the Font class:

```
aFont = Font(fontname = String, style = int, size = int)
```

A font object cannot be changed after it has been created.

Font Styles

The font style defines the thickness and the slant of a font. Table 9 shows the available font styles and the corresponding constant.

Table 9. Font Styles	
Style	**Constant**
Plain or roman	Font.PLAIN
Italic	Font.ITALIC
Bold	Font.BOLD
Bolditalic	Font.BOLD + FONT.ITALIC

Font Attributes

A font can be sized to every point size. One point is approximately 1/72 of an inch. There is no guarantee that you get a font with exactly the size you requested. You should always use the FontMetrics class to determine the exact size of the font that was created.

The FontMetrics class contains information about the visual attributes of a font (see Figure 66).

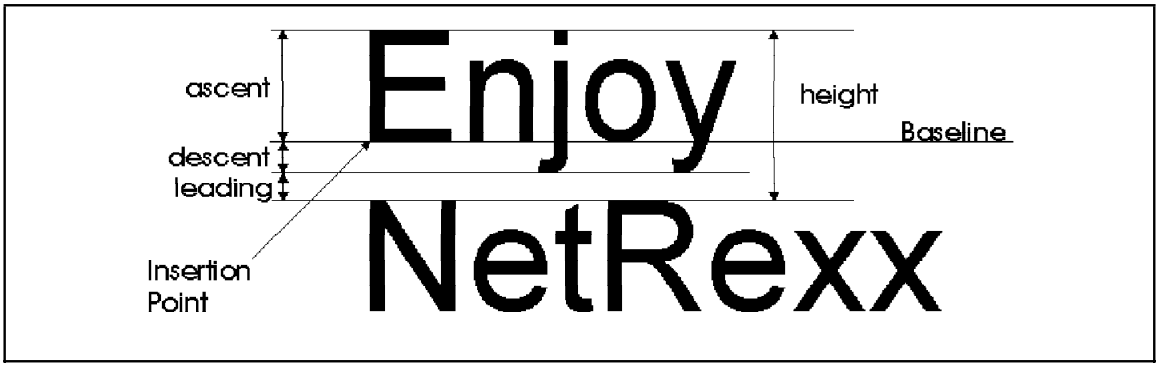


Figure 66. Font Attributes

The FontMetrics class is an abstract class, so you cannot directly create an instance of it. Use the getFontMetrics method, which is implemented for the Component, Graphics, and Toolkit classes:

```
-- obj   is an object of the component, graphics, or Toolkit class
-- aFont is an object of the class Font
fm = FontMetrics
fm = obj.getFontMetrics(aFont)
```

Methods of the FontMetrics class:

getAscent() returns int	Returns the font's ascent in pixels
getDescent() returns int	Returns the font's descent in pixels
getHeight() returns int	Returns the font's height in pixels
getLeading() returns int	Returns the font's leading in pixels
stringWidth(s = String) returns int	Returns the width of the string in pixels.

For an example of how to use fonts, see "Extended Label Component" on page 134.

Images

The classes available for working with images are spread across the java.applet, java.awt, and java.awt.image packages.

An image is represented by an object of the java.awt.Image class. You can load and display an image using the getImage method of the Applet or Toolkit class. Currently GIF and JPEG images are supported.

The Image class is an abstract class. When an image is loaded, an object of a subclass of Image is returned to the caller. The exact class type is not of importance to the programmer.

See “Image Component” on page 139 for an example of working with images.

Loading an Image

If the image data is available as a GIF or JPEG file, the getImage method of the Applet or Toolkit class can be used to load the image.

The getImage method returns immediately, without even checking whether the image exists, and before the image is loaded. The image is actually loaded when the program draws the image the first time.

Applet: For an applet, the getImage methods of the Applet class can be used:

getImage(anUrl = URL) returns Image
: Returns an Image object that will load the image from the given URL

getImage(anUrl = URL, filename = String) returns Image
: Returns an Image object that will load the image from an URL created by the given URL and file name

These methods cannot be used in a constructor of an applet. However, they can be used in the init method of an applet.

Note: URLs are discussed in more detail in “URL Handling” on page 212.

Example of using the getImage methods of Applet:

```
class MyApplet extends Applet
  method init()
    img1 = getImage(imageURL)  -- imageURL is an instance variable
    img2 = this.getImage(getCodeBase(),"picture.gif")
```

Application: For applications you must use the getImage methods of the Toolkit class. Use Toolkit.getDefaultToolkit to retrieve a Toolkit object. The Toolkit class defines two getImage methods:

getImage(anUrl = URL) returns Image
: Returns an Image object that will load the image from the given URL

getImage(filename = String) returns Image
: Returns an Image object that will load the image from the given file

Example of using the getImage methods of the Toolkit class:

```
tk   = Toolkit.getDefaultToolkit()  -- gets the Toolkit

-- create the first image object from an URL
do
   anUrl = URL('http://java.sun.com/graphics/people.gif')
   img1  = tk.getImage(anURL)
```

```
 catch MalformedURLException        -- Oops, wrong URL
end

-- create the second image object from a file
img2 = tk.getImage('KIDS.GIF')
```

You can also use the getImage method of Toolkit in an applet. This is useful when you write a component that will be used by applications and applets (see "Image Component" on page 139).

Loading an Image Locally or from the Web

The *getResource* method of the Java Class class enables you to load images locally or from the Web with relative URLs. When you use this method, the image is loaded from a file in the CLASSPATH when you run an applet locally, and from the Web server when you run the applet in a Web browser.

Example of using the getResource method:

```
tk     = Toolkit.getDefaultToolkit()  -- gets the Toolkit

imgUrl = this.getClass().getResource('/redbook/KIDS.GIF')
img    = tk.getImage(imgUrl)
...
```

If your class is in a package, you have to prefix the file name with the package name directory:

```
imgUrl = this.getClass().getResource('/nrxbeans/lab/LED1.GIF')
img    = Toolkit.getDefaultToolkit().getImage(imgUrl)
```

MediaTracker

The image is loaded the first time the drawImage method is called. To load it in advance, use the MediaTracker class, which loads an image or a group of images in a background thread, gives you status information, and enables you to wait for an image or an image group to be loaded.

When adding an image to a MediaTracker object, you must specify an ID. The ID reflects the priority for loading images and can be used to get information about the load process. You can group the images by using the same ID.

The constructor of the MediaTracker class requires a component as a parameter:

```
MediaTracker(aComponent=Component)
```

Useful methods of the MediaTracker class are:

addImage(anImage=Image,id=int)
: Adds an image to the MediaTracker object with the given ID

checkAll(load=boolean 0) returns boolean
: Checks whether all load processes are finished. If *load* is true, the loading process is started.

checkID(id = int, load = boolean 0) returns boolean
: Checks whether the image or the group of images with the given ID is loaded. If *load* is true, the loading process is started.

isErrorAny() returns boolean
: Returns true if any image had an error while loading

waitForAll() throws InterruptedException
: Starts loading for all images. Returns when all images are loaded. Use the isErrorAny method to check for errors while loading.

waitForAll(timeout=long) returns boolean throws InterruptedException
: Starts loading for all images. Returns when all images are loaded or the specified timeout (in ms) occurs. Returns true if all images are successfully loaded. Use the isErrorAny method to check for errors while loading. If *timeout* is 0, the waitForAll method waits forever.

waitForID(id=int) throws InterruptedException
: Starts loading for all images. Returns when the image or the group of images specified by the given ID is loaded. Use the isErrorAny method to check for errors while loading.

waitForID(id=int,timeout=long) returns boolean throws InterruptedException
: Starts loading for all images. Returns when the image or the group of images specified by the given ID is loaded or the specified timeout (in ms) occurs. Returns true if all images are successfully loaded. Use the isErrorAny method to check for errors while loading. If *timeout* is 0, the waitForID method waits forever.

The following example loads one background image and a group of images for animation:

```
class Animator extends Applet implements Runnable
 Properties constant
   images = 4                                    -- 4 images for animation
 Properties inheritable
   background = Image
   animator   = Image[images]                    -- array for animation

 method init()                                   -- init the applet
   tracker = MediaTracker(this)                  -- create the mediatracker
   background = getImage(getCodeBase(),"bkg.gif") -- background image
   tracker.addImage(background,1)                -- track background image id=1
   loop i=0 for images                           -- load animator images
      animator[i] = getImage(getCodeBase(),"anim"i".gif")  -- anim1.gif, etc
      tracker.addImage(animator[i],2)            -- track the image group id=2
   end
   do                                            -- load and show images
      tracker.waitForID(1)                       -- load background
      repaint()                                  -- show the background
      tracker.waitForID(2)                       -- wait for the animations
      start()                                    -- start animation
    catch InterruptedException
   end
```

This example is not complete. There is no code for error handling, and the painting of the images and the animation part are missing. See Figure 67 on page 122 for a complete example.

Drawing an Image

To draw an image, use the drawImage method of an object of the Graphics class.

Typically the drawImage method is used in the paint or update method. You have to subclass a window, applet, or component to get access to these methods.

Several overloaded versions of the drawImage method are defined by the Graphics class:

drawImage(img=Image, x=int, y=int, observer=ImageObserver)
: Draws image *img* at point (*x*,*y*). Uses the image observer, *observer*, if the image must be loaded.

drawImage(img=Image, x=int, y=int, width=int, height=int, observer=ImageObserver)
Same as above, except that the image is scaled to the given width and height.

drawImage(img=Image, x=int, y=int, bgcolor=Color,observer=ImageObserver)
Draws image *img* at point (*x*,*y*). Uses the image observer, *observer*, if the image must be loaded. Uses color *bgcolor* for transparent parts of the image.

drawImage(img=Image, x=int, y=int, width=int, height=int, bgcolor=Color, observer=ImageObserver)
Same as above, except that the image is scaled to the given width and height.

The image is loaded when it is drawn the first time, and if it was not loaded in advance by a MediaTracker object. Loading an image, especially if it is loaded from the Internet, takes some time. To avoid a blocked interface, the image loading is an asynchronous process carried out in the background. The object that processes the background task is an ImageProducer.

One parameter of the drawImage method is ImageObserver, which is an interface that defines the imageUpdate method. The imageUpdate method is called from the ImageProducer a few times while the image is loaded.

The ImageObserver interface is implemented by the Component class. The implementation calls the repaint method every time the imageUpdate method is invoked, and repaint invokes the paint method of the applet.

A typical paint method for an applet drawing an image is:

```
method paint(g = Graphics)
  -- img is an instance variable of type Image
  d = getSize()                              -- gets the size of the component
  g.drawImage(img,0,0,d.width,d.height,this) -- draw the image
```

This paint method scales the image to the size of the applet. The same method would work for a component or a window.

Animated Images

Applets with animated images are currently the most fashionable Java programs. These applets start a thread that continuously triggers the repaint method of the applet.

One problem with animation is flickering. Flickering occurs when the screen is refreshed by the operating system during a paint action. Two points in the repainting process of an applet are responsible for flickering:

- The update method erases the background and calls the paint method after that. You must change this method to call the paint method without erasing the background:

  ```
  method update(g = Graphics)
    paint(g)
  ```

- The paint method draws (as a default) to a graphics context object that is visible on the screen. Use an offscreen graphics context instead, and copy the resulting image with one call to the applet when the drawing is complete. The usage of offscreen images is also known as *double buffering*.

  A sample paint method that creates an offscreen image for painting is:

  ```
  method paint(g = Graphics)
    offScreenImage = createImage( getSize().width, getSize().height )
    gContext       = offScreenImage.getGraphics()

    -- do the drawing using the gContext object
    gContext.drawString(....)
    ...
  ```

```
    -- draw the offScreenImage to the screen
    g.drawImage(offScreenImage,0,0,this)
    gContext.dispose()                          -- release the resources
```

Figure 67 shows the implementation of an animated applet. The applet creates an offscreen image of a string ("NetRexx") and moves this image from right to left. When the image reaches the left border of the applet, the image is squeezed (compressed) and changes color. The applet then unsqueezes the image and moves it right to the center with more changes in color.

```
/* gui\animator\Animator.nrx

   Animated applet.  Creates an image from a string and moves the images over the screen */

class Animator extends Applet implements Runnable

  Properties inheritable
    netrexx     = String "NetRexx"              -- image string
    stringImage = Image                         -- image of the drawn string
    gi          = Graphics                      -- graphics object of image
    imagepos    = int                           -- where to draw image text (y)
    x           = int 200                       -- current image position (x)
    height      = int                           -- height of the image
    imgwidth    = int                           -- width of the image
    width       = int                           -- width of the image squeezing

  --------------------------- applet INIT
  method init()
    x = 200; height = 0; width = 0              -- re-init
    f  = Font('Helvetica',Font.BOLD,30)         -- choose the font
    fm = getFontMetrics(f)                      -- get the metrics of the font
    imagepos = fm.getAscent()                   -- where to draw text
                                                -- create an image of the string NetRexx
    stringImage = createImage(fm.stringWidth(netrexx)+20,fm.getHeight())
    gi = stringImage.getGraphics()              -- get graphics object
    gi.setFont(f)
    drawText(Color.black)                       -- start with black
    imgwidth = stringImage.getWidth(this)       -- image size
    height   = stringImage.getHeight(this)
    Thread(this,'Animator Thread').start()      -- start animation

  --------------------------- draw the "NetRexx" string into the image
  method drawText(c=Color) private
    gi.setColor(c)                              -- set the color
    gi.drawString(netrexx,0,imagepos)           -- draw the string

  --------------------------- calculates the x Position of the image when moving
  method calculatePosition(i=int) private
    x = 200-i

  --------------------------- calculates the width of the image when squeezing
  method calculateSize(i=int) private
    width = imgwidth-i

  --------------------------- applet UPDATE (dont erase background to avoid flicker)
  method update(g=Graphics)
    paint(g)
```

Figure 67 (Part 1 of 2). An Animated Applet: Animator.nrx

```
--------------------------- applet PAINT (draw the image)
method paint(g=Graphics)
  g.setClip(x,10,stringImage.getWidth(this)+5,stringImage.getHeight(this)+10)
  g.drawImage(stringImage,x,10,width,height+10,this)

--------------------------- applet RUN: animation: move, squeeze, unsqueeze, move
method run()
  width = imgwidth
  ct = Thread.currentThread()     -- get thread for sleep times
  do
    loop i=1 to 200 by 2          -- move the string to the left border
       calculatePosition(i)
       repaint()
       ct.sleep(10)
    end
    drawText(Color.red)           -- change color
    loop w=1 to imgwidth-25 by 2  -- squeeze the string at the border
       calculateSize(w)
       repaint()
       ct.sleep(10)
    end
    ct.sleep(150)                 -- wait a moment
    drawText(Color.green)         -- change color
    loop while w > 0              -- unsqueeze the image
       calculateSize(w)
       repaint()
       ct.sleep(10)
       w = w - 1
    end
    drawText(Color.blue)          -- change color
    loop i=200 to 150 by -1       -- move the image back right
       calculatePosition(i)
       repaint()
       ct.sleep(220 - i)          -- slow down every move
    end
    ct.sleep(2000)                -- let for 2 seconds
    drawText(Color.white)         -- change color
    repaint()
   catch InterruptedException
  end
```

Figure 67 (Part 2 of 2). An Animated Applet: Animator.nrx

We use this animated applet to create a JavaBean for VisualAge for Java (“Creating an Animated JavaBean” on page 255). Figure 140 on page 256 shows a snapshot of the running applet.

Lightweight Components

In the beginning of 1997 Sun started the *Swing* project to implement a set of lightweight components, and with the JDK 1.1 Sun introduced a set of lightweight components. See *http://java.sun.com/products/jdk/awt/swing* for more details.

Lightweight components do not have a native peer.

Every other component consists of the class you specify and another class that is called the *peer*. Peer components are native components if they use the control windows of the operating system. Because every operating system has a different look and feel, Java and NetRexx programs act a bit differently on different platforms.

Because there is no native peer for the lightweight components, you have to draw the components yourself. As a result, the components have the exact same look and feel on every platform.

Another advantage of lightweight components is that you can change the look of the component by writing subclasses. If the drawing is done by a native peer, such changes are impossible.

To create a lightweight component, your new class must be a direct subclass of Component, Container, or any existing lightweight component class. Override the following methods with your new implementation:

paint(g=Graphics)
: Draws the component

getMinimumSize() returns Dimension
: Returns the minimum size of the component

getPreferredSize() returns Dimension
: Returns the preferred size of the component. The preferred size can be different from the minimum size. For example, if the component includes a margin, the minimum size can return the size of the component without the margin. If you do this, you must write code in the paint method to draw the component in a different way when the size of the component is smaller than the preferred size.

The update method does not have to be implemented, because it is not called for lightweight components when the components are repainted.

If your component needs to erase the background, you must implement this in the paint method.

See “Extended Label Component” on page 134 and “Image Component” on page 139 for examples of lightweight components.

Problem Solutions and Examples

In this section we describe solutions for some standard problems, such as the closing event of windows and translating keystrokes. We also present examples of how to build reusable classes, which are a big advantage of the event delegation model.

Closing Windows

When a frame or dialog window is closed by using the frame context menu or the *Alt-F4* shortcut, a window event is fired, which corresponds to the closingWindow method of the WindowListener interface.

If a listener has not been added to the window, the event is ignored. Therefore every window has to add a WindowListener to catch this event.

The first simple WindowListener implementation just exits the whole application:

```
CloseWindow extends WindowAdapter
  method closingWindow(e=WindowEvent)
    exit 0
```

The CloseWindow class inherits from WindowAdapter instead of implementing the complete WindowListener interface, which defines seven different methods, of which we are interested in only one.

If we want to destroy a window but not exit the whole application, we can easily write another class with this behavior. A better idea, however, is to make the class configurable as shown in Figure 68.

```
/* gui\closewindow\CloseWindowA.nrx

   Implements a reusable WindowListener which closes Windows */

class CloseWindowA extends WindowAdapter
  Properties public constant
    HIDE     = int 0                   -- hide the window          (setVisible(0))
    DESTROY  = int 1                   -- destroy the window       (dispose())
    SHUTDOWN = int 2                   -- shutdown the application (exit)

  Properties inheritable
    behaviour = int
    parent    = Window
                                       -- constructor: default destroy window
  method CloseWindowA(cWindow = Window, theBehaviour=int DESTROY)
    behaviour = theBehaviour
    parent    = Window cWindow.getParent()   -- save the parent (to bring in front)

  method windowClosing(e=WindowEvent)        -- called when window is closed
    select
      when behaviour = HIDE then             -- hide the window
          e.getWindow().setVisible(0)
      when behaviour = DESTROY then          -- destroy the window
          e.getWindow().dispose()
      otherwise exit 0                       -- exit the application
    end
    if parent \= null then                   -- put the parent in front
       parent.toFront()                      -- (it is not automatic)
```

Figure 68. Simple Close Window Event Listener: CloseWindowA.nrx

This version of our CloseWindowA class can be configured and fix the problem that the parent window does not appear on top when the child window is destroyed.

Notes:

1. When the parent is null, we assume that the main window of the application has been closed. If the main window is closed, we exit the application.
2. There are three different constructors: one that accepts a dialog window, one that accepts a frame and parent window, and one that accepts a frame window only.
3. You can get the information about the parent window from a dialog window, so it is not necessary to set the parent window.
4. A frame window always returns null for the getParent method. Therefore, the parent window is retrieved in the constructor.
5. If only a frame window is used in the constructor, the behavior is set to SHUTDOWN because we assume that the frame window is the main window of the application and the application exits when the frame window is closed.
6. A modal dialog window cannot be hidden. The init method changes the behavior parameter to DESTROY if the window is a modal dialog, and the behavior is requested as HIDE.

The CloseWindowA class can be tested with a simple test program:

```
/* gui\closewindow\CloseTst.nrx   Test the CloseWindowA class */
win = Frame('TestWindow')
win.addWindowListener(CloseWindowA(win,CloseWindowA.SHUTDOWN))
win.setSize(100,100)
win.setVisible(1)
```

Action Events from Menus and Buttons

The previous example implemented a listener that closes a window when the user uses the system menu. Now we want to extend the CloseWindow class to react to ActionEvents. First, we have to change the class definition, because we implemented the ActionListener interface:

```
class CloseWindow extends WindowAdapter implements ActionListener
```

In addition, we have to implement the actionPerformed method that is defined by the ActionListener interface:

```
method actionPerformed(e = ActionEvent)
  closeTheWindow()
```

We invoke the closeTheWindow method that closes the window. This method is also used by the windowClosing method:

```
method windowClosing(e=WindowEvent)  -- called when window is closed
  closeTheWindow()
```

The ActionEvent object does not have any reference to the window where the event occurred. Therefore, we cannot use the getWindow method. We need a new instance variable that stores the window reference. The reference can be used by the closeTheWindow method. Figure 69 shows the complete class.

```
/* redbook\gui\CloseWindow.nrx

   Implements a reusable WindowListener which closes Windows */

package Redbook

class CloseWindow extends WindowAdapter implements ActionListener
  Properties public constant
    HIDE     = int 0         -- hide the window          (setVisible(0))
    DESTROY  = int 1         -- destroy the window       (dispose())
    SHUTDOWN = int 2         -- shutdown the application (exit)

  Properties inheritable
    behaviour = int
    theWindow = Window
    parent    = Window

  -- constructor for dialog windows
  method CloseWindow(cWindow = Dialog, theBehaviour=int DESTROY)
    init(Window cWindow.getParent(),cWindow,theBehaviour)

  -- constructor for frame windows
  method CloseWindow(parentWindow = Window, cWindow = Window,-
                     theBehaviour=int DESTROY)
     init(parentWindow,cWindow,theBehaviour)

  -- constructor for main application frame windows (set parent = null)
  method CloseWindow(cWindow = Frame)
     init(null,cWindow,SHUTDOWN)
```

Figure 69 (Part 1 of 2). Close Window Event Listener: CloseWindow.nrx

```
-- method to set a new behaviour
method setCloseBehaviour(newBehaviour = int)
  behaviour = newBehaviour

-- method to init - private because not type safe
method init(parentWindow = Window, cWindow = Window, -
            theBehaviour = int) private
  parent    = parentWindow            -- save the parent
  theWindow = cWindow                 -- save the window reference
  behaviour = theBehaviour            -- save behaviour
  if theWindow <= Dialog then         -- is it a dialog?
    if (Dialog theWindow).isModal() & -
        behaviour == HIDE then        -- modal dialogs cannot hide
      behaviour = DESTROY             -- they must be destroyed

method windowClosing(e=WindowEvent)   -- called when window is closed
  closeTheWindow()

method actionPerformed(e=ActionEvent) -- called from button or menus
  closeTheWindow()

method closeTheWindow() inheritable
  select
    when behaviour = HIDE then        -- hide the window
        theWindow.setVisible(0)
    when behaviour = DESTROY then     -- destroy the window
        theWindow.dispose()
    otherwise exit 0                  -- exit the application
  end
  if parent \= null then do           -- parent in front
     RedbookUtil.sleep(220)
     parent.toFront()
  end
```

Figure 69 (Part 2 of 2). Close Window Event Listener: CloseWindow.nrx

The CloseWindow class is a member of the Redbook package.

Setting the Focus in Windows

A window gives the focus to the first component that can receive the keyboard focus when the window is opened for the first time. When the window is destroyed or hidden and shown again, the focus remains on the window. If the window is deactivated (the window and all of its components lost the focus) and activated by a click on the title bar of the window, the focus is set to the window and not to any component of the window.

The expected behavior is that when a window is shown, the focus is set to a specified component. If the window is deactivated and activated by a click on the window title bar, the focus is set to the component that owned the focus at deactivation time. To implement the desired behavior, we use a FocusListener that sets the focus to a component of the window, whenever the window by itself receives the focus.

When the window is deactivated the component that owns the focus is stored. When the window is activated again, the focus is set to the stored component. The corresponding methods are windowDeactivated and windowActivated.

When the window is destroyed, the specified component is stored to receive the focus at the next activation. This is implemented by receiving the windowClosed event.

Figure 70 shows the code that implements our first approach for a WindowFocus class.

```
/* redbook\gui\WindowFocus.nrx

   Sets the focus to a defined component when the window is shown,
   saves focus component when deactivated, and resets the focus when reactivated */

package Redbook

class WindowFocus extends WindowAdapter
  Properties
    theWindow        = Window
    defaultrecipient = Component       -- default when window was destroyed
           recipient = Component       -- the component used at activation time

  -- constructor, sets default recipient
  method WindowFocus(aWindow = Window, focusRecipient = component)
    defaultrecipient = focusRecipient     -- default
    recipient        = focusRecipient
    theWindow        = aWindow
    aWindow.addWindowListener(this)

  -- Window is activated: set the focus to the recipient
  method windowActivated(e=WindowEvent)
    recipient.requestFocus()

  -- stores the focus when the window is deactivated
  method windowDeactivated(e=WindowEvent)
    recipient = theWindow.getFocusOwner()

  -- window was destroyed, next time defaultrecipient
  method windowClosed(e=WindowEvent)
    recipient = defaultrecipient
```

Figure 70. WindowFocus Class: WindowFocus.nrx

Notice that the constructor adds the object itself to the window. This is done to prevent errors, because most people do not expect that a focus handler is a window listener.

A small test program, FocusTst.nrx, is provided in the gui\windowfocus subdirectory.

The WindowFocus class is a member of the Redbook package.

Automatic Selection in TextField Objects

When the focus is set to a TextField object, the current contents should be selected, so that a user can enter a new value that destroys the current text.

When text in a TextField object is selected and the focus leaves the object, any selection should be reset.

A focus listener added to a TextField object is the solution for this problem. Figure 71 shows the code of the FieldSelect class.

```
/* redbook\gui\FieldSelect.nrx

   Selects the contents of a TextField object when the object gets the focus.
   Remove any selection when the object looses the focus */

package Redbook

class FieldSelect implements FocusListener

  -- select the text in the TextField
  method focusGained(e=FocusEvent)
    if \e.isTemporary() then (TextField e.getComponent()).selectAll()

  -- remove any selection
  method focusLost(e=FocusEvent)
    if \e.isTemporary() then (TextField e.getComponent()).select(0,0)
```

Figure 71. FieldSelect Class: FieldSelect.nrx

A small test program, FieldTst.nrx, is provided in the gui\fieldselect subdirectory.

The FieldSelect class is a member of the Redbook package.

Adding Listeners Automatically

In the previous example, an object of the FieldSelect class is added to every text field in the window. This can be a lot of work, and we can expect that some programmers do not add the necessary listener to some text fields.

An interesting approach is to write a class that inherits from the WindowsFocus class and automatically adds an object of our FieldSelect class to every TextField object of the window. If we combine this approach with the CloseWindow class, only one object is needed per window, and that is easier to control.

To automatically add a FieldSelect object to text fields, we have to think about when to look for text fields. If we look when the window is opened, we cannot be sure that fields will not be added at run time. We have to scan all components and add a component listener to our window, which informs us about all changes in the structure of the window. When new text fields are added, we can attach our FieldSelect object and remove it when text fields are removed from the window.

We implement the new class step by step:

1. Our new class has to implement the ContainerListener interface:

   ```
   class WindowSupport extends WindowFocus implements ContainerListener
   ```

2. We define a method that scans all components for text fields and adds a focus listener to the fields:

   ```
   Properties inheritable
    fieldSelectObj = FieldSelect()   -- FocusListener used for TextFields

   method scanComponents()
    components = this.theWindow.getComponents() -- returns an array of components
    loop i=0 to components.length-1             -- 0 based arrays
       if components[i] <= TextField then
          components[i].addFocusListener(fieldSelectObj)
    end
   ```

3. We have to support the events when components are added to the window:

```
   method componentAdded(e=ContainerEvent)
      comp = e.getChild()              -- returns the added component
      if comp <= TextField then
        comp.addFocusListener(fieldSelectObj)
```

Figure 72 shows the code of the WindowSupport class.

```
/* redbook\gui\WindowSupport.nrx

   Full support for a Window:   closingWindow events are handled
                                focus switches are supported
                                text selection in text fields is supported
                                (null for a focus recipient is allowed) */

package Redbook

class WindowSupport extends WindowFocus implements ContainerListener
 Properties inheritable
   fieldSelectObj = FieldSelect()   -- FocusListener used for TextFields
   closeListener  = CloseWindow

 -- constructor when a parent window is given
 method WindowSupport(parentWindow = Window, currentWindow = Window,-
                      focusRecipient = Component null,-
                      closeBehaviour=int CloseWindow.DESTROY)
   super(currentWindow,focusRecipient)
   closeListener = CloseWindow(parentWindow,currentWindow,closeBehaviour)
   init(currentWindow)

 -- constructor when the window is the main application frame window
 method WindowSupport(aFrameWindow = Frame, focusRecipient = Component null)
   super(aFrameWindow,focusRecipient)
   closeListener = CloseWindow(aFrameWindow)
   init(aFrameWindow)

 -- constructor when the window is a dialog window
 method WindowSupport(aDialog = Dialog, focusRecipient = Component null,-
                      closeBehaviour=int CloseWindow.DESTROY)
   super(aDialog,focusRecipient)
   closeListener = CloseWindow(aDialog,closeBehaviour)
   init(aDialog)

 -- return the closeWindow object
 -- used to add the closeWindow object to pushbuttons or menus
 method getCloseWindow() returns CloseWindow
   return closeListener

 -- sets a new focus recipient
 -- change the current recipient if equal null
 method setFocusRecipient(aComponent = Component)
   super.defaultrecipient = aComponent
   if super.recipient == null then super.recipient = aComponent

 -- sets a new close behaviour for the closewindow Listener
 method setCloseBehaviour(newBehaviour=int)
   closeListener.setCloseBehaviour(newBehaviour)

 -- overrides the windowActivated method from WindowFocus to handle null
 method windowActivated(e=WindowEvent)
```

Figure 72 (Part 1 of 2). WindowSupport Class: WindowSupport.nrx

```
    if super.recipient \= null then super.windowActivated(e)

  -- scans all components of the window for TextField objects
  -- handles nested containers
  method scanComponents(components = Component[]) inheritable
    loop i=0 to components.length-1          -- 0 based arrays
       if components[i] <= TextField then
          components[i].addFocusListener(fieldSelectObj)
       if components[i] <= Container then do
          (Container components[i]).addContainerListener(this)
          scanComponents((Container components[i]).getComponents())
       end
    end

  -- called from the constructors do init the class
  method init(aWindow = Window) inheritable
    aWindow.addWindowListener(closeListener)
    aWindow.addContainerListener(this)
    scanComponents(this.theWindow.getComponents())

  -- invoked when a new component is added to any container of the window
  method componentAdded(e=ContainerEvent)
    comp = e.getChild()              -- returns the added component
    if comp <= TextField then
       comp.addFocusListener(fieldSelectObj)
    if comp <= Container then        -- handles nested containers
      (Container comp).addContainerListener(this)

  -- invoked when a component is removed from the window
  method componentRemoved(e=ContainerEvent)
    comp = e.getChild()              -- returns the added component
    if comp <= TextField then
       comp.removeFocusListener(fieldSelectObj)
    if comp <= Container then        -- handles nested containers
       (Container comp).removeContainerListener(this)
```

Figure 72 (Part 2 of 2). WindowSupport Class: WindowSupport.nrx

A small test program, SuppTest.nrx, is provided in the gui\windowsupport subdirectory.

The WindowSupport class is a member of the Redbook package.

Controlling Keyboard Input

A typical problem for text fields is to translate keystrokes from lowercase to uppercase and to allow only a limited character set, such as numbers.

You can add a KeyListener to a component to control the input. The keyTyped method is invoked every time a new key is pressed.

If you want to change a typed key, you can use the getKeyChar and setKeyChar methods to access and change the key of a key event.

If you do not want to allow a key type, you have to consume the event, using the consume method, which is implemented in the AWTEvent class. The consume method stops further processing of the event. The event is *consumed* by your event listener.

Here is a simple key listener that allows only hexadecimal input and translates lowercase letters to uppercase:

```
class OnlyHexadecimal extends KeyAdapter
  Properties constant
    keySet = Rexx '0123456789ABCDEF'

  method keyTyped(e=Keyevent)
    key = Rexx e.getKeyChar()  -       - make key of type Rexx for further use
    if key.c2d() == KeyEvent.VK_BACK_SPACE then return
    key = key.upper()
    if keyset.pos(key) == 0 then
      e.consume
    else
      e.setKeyChar(key)
```

Figure 73 shows a more complete class.

```
/* redbook\gui\KeyCheck.nrx

   KeyListener implementation which compares the key with a
   lookup string and translate the key to uppercase if necessary */

package Redbook

class KeyCheck extends KeyAdapter

  Properties constant public
    ALL         = Rexx null
    NUMERIC     = Rexx '0123456789'
    ALPHA       = Rexx ' abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
    ALPHANUM    = Rexx ALPHA||NUMERIC
    HEXADECIMAL = Rexx '0123456789ABCDEF'

  Properties inheritable
    mode        = Rexx
    translate   = boolean 0

  method KeyCheck(modeString=Rexx ALL, toUpperCase = boolean 1)
    setMode(modeString,toUpperCase)

  method keyTyped(e=KeyEvent)
    key = Rexx e.getKeyChar()

    if key.c2d() == KeyEvent.VK_BACK_SPACE then return

    if translate then do
       key = key.upper()
       e.setKeyChar(key)
    end

    if mode \= null then
       if mode.pos(key) == 0 then do
          e.consume
          Toolkit.getDefaultToolkit().beep()
       end

  -- methods to change the current mode

  -- change the lookup string
  method setMode(newMode = Rexx)
    mode = newMode
```

Figure 73 (Part 1 of 2). Check and Manipulate Key Events: KeyCheck.nrx

```
  -- change the translation
  method setMode(toUpperCase = boolean)
    translate = toUpperCase

  -- change the lookup string and the translation
  method setMode(newMode = Rexx, toUpperCase = boolean)
    mode = newMode
    translate = toUpperCase
```

Figure 73 (Part 2 of 2). Check and Manipulate Key Events: KeyCheck.nrx

The KeyCheck class is a member of the Redbook package.

Limiting the Length of a TextField

If you want to limit the length of a TextField to a given number of characters, you can use the TextListener interface as shown in Figure 74.

```
/* redbook\gui\LimitTextField.nrx

   Limits the length of a TextField to a number of characters */

package Redbook

class LimitTextField implements TextListener
   Properties inheritable
     maxlen = int
     field  = TextField

   method LimitTextField(afield = TextField, limit = int)
     field = afield
     maxlen= limit

   method textValueChanged(e=TextEvent)
     s = Rexx field.getText()           -- get the TextField contents
     if s.length() > maxlen then do     -- check if to long
       pos = field.getCaretPosition()   -- remember the cursor position
       s = s.left(maxlen)               -- truncate the string
       field.setText(s)                 -- back to TextField
       field.setCaretPosition(pos)      -- reset the cursor position
       field.getToolkit().beep()        -- make noise
     end
```

Figure 74. Limit the Length of a TextField: LimitTextField.nrx

The LimitTextField class is a member of the Redbook package.

Using Buttons of the Same Size

When you lay out a window, most of the time you have an area where push buttons are used. If the buttons are in one row, they can have different sizes; if the buttons are in one column, they should all have the same size.

To create buttons of the same size you can use a GridLayout manager. Everything in the container is sized to the same size. The drawback of the GridLayout manager is that it resizes all included components to the full size of the container.

The solution to the problem is to create a panel, using the FlowLayout manager (default), and add another panel, using a GridLayout manager. The FlowLayout manager forces the GridLayout manager to resize to its preferred size. The preferred size is evaluated by checking the components and using the largest component size for all.

Figure 75 shows a subclass of Panel that implements same-sized buttons by using two layout managers.

```
/* redbook\gui\EqualSizePanel.nrx

   Implements a panel that equalizes the size of its components */

package Redbook

class EqualSizePanel extends Panel
  Properties constant public
    HORIZONTAL = int 0
    VERTICAL   = int 1

  Properties inheritable
    fLayout = FlowLayout(FlowLayout.LEFT,0,0)
    gPanel  = Panel()

  method EqualSizePanel(alignment = int HORIZONTAL,gap = int 5)
    fLayout = FlowLayout(FlowLayout.LEFT,0,0)
    setLayout(fLayout)
    super.add(gPanel)                          -- add the other panel
    if alignment = HORIZONTAL then             -- create the GridLayout manager
      gPanel.SetLayout(GridLayout(1,0,gap,0))  -- one row, hgap=gap
    else
      gPanel.SetLayout(GridLayout(0,1,0,gap))  -- one column, vgap=gap

  method add(comp = Component) returns Component
    gPanel.add(comp)
    return comp

  method getFlowLayout() returns FlowLayout
    return fLayout

  method setGaps(hgap = int, vgap = int)
    fLayout.setHgap(hgap)
    fLayout.setVgap(vgap)
```

Figure 75. Panel with Same-Sized Buttons: EqualSizePanel.nrx

A small test program, EqualTst.nrx, is provided in the gui\equalsizepanel subdirectory.

The EqualSizePanel class is a member of the Redbook package.

Extended Label Component

The extended label component is a lightweight component (see “Lightweight Components” on page 123) that implements a label, similar to the *Label* class of the JDK. The extended label formats the text in multiple lines and with vertical alignment.

The text given to an ExtendedLabel object is separated in lines by a separator character. The default for the separator character is ′\n′.

Figure 76 shows the code for the ExtendedLabel class.

```
/* redbook\gui\ExtendedLabel.nrx

   An extended Label has the ability for multiple lines with
   vertical and horizontal alignment.
   The text of the extended Label is separated into lines.
   The default separator character is '\n'.
   The ExtendedLabel class is a 'lightweight' component */

package Redbook

class ExtendedLabel extends Component
  Properties constant public
    LEFT       = 0               -- alignment constants
    RIGHT      = 1
    CENTER     = 2
    TOP        = 3
    BOTTOM     = 4
  Properties inheritable
    text       = Rexx            -- indexed Rexx string for the lines
    lines      = int 0           -- number of lines
    halign     = int             -- horizontal alignment
    valign     = int             -- vertical alignment
    inset      = Insets(0,0,0, 0) -- Insets for the component
    separator  = char '\n'       -- line separator
    maxlength  = int 0           -- max. length of the component in pixel
    lineLength = Rexx '0'        -- indexed Rexx string for length of the line
    lineHeight = int             -- height of one line
    lineDescent= int
    lineAscent = int             -- ascent of the font
    prefSize   = Dimension       -- preferred size of the label
    doUpdate   = boolean 1       -- controls repainting

  -- constructs the component with given insets (margins)
  -- The lines are separated by '\n' characters in the string
  method ExtendedLabel(ltext=Rexx,aInsets=Insets,-
                       hor_align=int LEFT, ver_align=int TOP)
    this(ltext,hor_align,ver_align)
    inset = Insets aInsets.clone()

  -- constructs the component without any insets
  method ExtendedLabel(ltext=Rexx,hor_align=int LEFT, ver_align=int TOP)
    text      = ltext
    halign    = hor_align
    valign    = ver_align
    parseText()

  -- sets the text of the label
  method setText(ltext = String)
    text = ltext
    parseText()
    calculateMetrics()
    repaint()

  -- sets the separator and reruns the separation of the text
  method setSeparator(sep = char)
    separator = sep
    setText(text)           -- start the separation with the old string
```

Figure 76 (Part 1 of 4). Extended Label Class: ExtendedLabel.nrx

```
  -- sets the horizontal alignment method
  method setHorizontalAlignment(align = int)
    halign = align
    repaint()

  -- sets the vertical alignment method
  method setVerticalAlignment(align = int)
    valign = align
    repaint()

  -- returns the horizontal alignment method
  method getHorizontalAlignment() returns int
    return halign

  -- returns the vertical alignment method
  method getVerticalAlignment() returns int
    return valign

  -- sets the insets (margins) of the component
  method setInsets( newInsets = Insets)
    inset = Insets newInsets.clone()
    repaint()

  -- returns the insets (margins) of the component
  method getInsets() returns Insets
    return inset

  -- invoked from layout managers to figure out the preferred size
  method getPreferredSize() returns Dimension
    return Dimension(maxlength + inset.left + inset.right,-
                     lines * lineHeight + inset.top + inset.bottom)

  -- invoked from layout managers to figure out the minimum size
  method getMinimumSize() returns Dimension
    return Dimension(maxlength, lines * lineHeight)

  -- invoked when the component is created
  -- after calling the super method font metrics are available
  method addNotify()
    super.addNotify()
    calculateMetrics()

  -- Sets the font and calculates the new metrics
  method setFont(f = Font)
    super.setFont(f)
    calculateMetrics()
    repaint()

  -- Sets the text color, background has to be set by parent
  method setForeground(c = Color)
    super.setForeground(c)
    repaint()

  -- Prevents repainting if set to false
  method setUpdate(b = boolean)
    doUpdate = b
    repaint()
```

Figure 76 (Part 2 of 4). Extended Label Class: ExtendedLabel.nrx

```
-- internal method: extract the lines from the text string
method parseText() inheritable
  ltext = text
  loop line = 1 while ltext \= ''
     parse ltext temp (separator) ltext
     text[line] = temp
  end
  lines = line-1

-- internal method: calculate the current metrics of the component
method calculateMetrics() inheritable
  fm = this.getFontMetrics(this.getFont())
  if fm == null then return     -- no font metrics available
  lineHeight = fm.getHeight()
  lineDescent = fm.getDescent()
  lineAscent = fm.getAscent()
  maxLength = 0
  loop line=1 for lines
     lineLength[line] = fm.stringWidth(text[line])
     if maxLength < lineLength[line] then maxLength = lineLength[line]
  end
  prefSize = getPreferredSize()

-- draw the entire extended label
method paint(g=Graphics)
  if \doUpdate then return   -- no painting when forbidden
  off = Insets inset.clone() -- offset for margins
  d   = getSize()            -- returns a Dimension
                             -- Check the size and remove the margins when necessary
  if d.width < prefSize.width then do
    off.left = 0; off.right = 0
  end
  if d.height < prefSize.height then do
    off.top = 0; off.bottom=0
  end
                             -- calculate the y coordinate for the starting point
  d.height = d.height - off.top - off.bottom
  select
    when valign = TOP    then ypos = lineAscent
    when valign = BOTTOM then ypos = d.height - lineHeight * lines + lineAscent
    otherwise
      ypos = (d.height - lineHeight * lines) % 2 + lineAscent
  end
  ypos = ypos + off.top
                             -- set the font and the color
  g.setColor(getForeground())
  g.setFont(getFont())
                             -- draw the lines
  loop i=1 for lines
    select
      when halign = LEFT then xpos = off.left
      when halign = RIGHT then
        xpos =  d.width - off.right - lineLength[i]
      otherwise -- CENTER
        xpos = (d.width - off.left - off.right - lineLength[i])%2 + -
               off.left
    end
    g.drawString(text[i],xpos,ypos)
```

Figure 76 (Part 3 of 4). Extended Label Class: ExtendedLabel.nrx

```
        ypos = ypos + lineHeight
      end
```

Figure 76 (Part 4 of 4). Extended Label Class: ExtendedLabel.nrx

The ExtendedLabel class is a member of the Redbook package.

Repainting of the component can be suppressed with the setUpdate(allowRepaint) method. If *allowRepaint* is false, the component does not repaint. This method is useful if multiple changes, such as changing the alignment and the text, are made.

A sample application (Figure 77) illustrates the usage of the ExtendedLabel class. The application changes the font, color, and horizontal alignment with every mouse click on the label. The vertical alignment changes with every third mouse click.

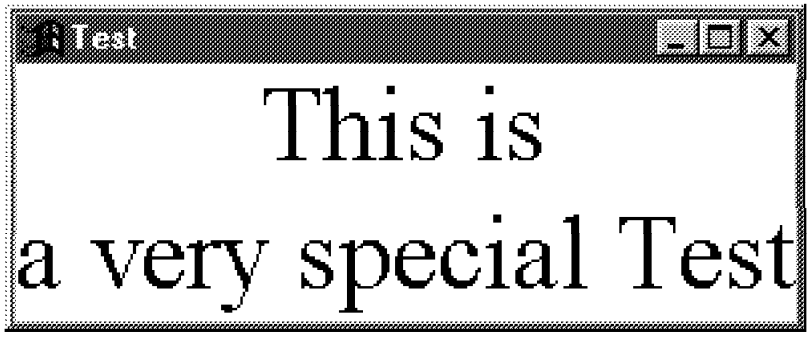


Figure 77. Extended Label Test Application

Figure 78 shows the code of the extended label test application.

```
/* gui\extendedlabel\ExtTest.nrx

   Test program for the ExtendedLabel class.
   Click with the left mouse button on the label */

import Redbook.                -- use the Redbook package

win = Frame('Test')            -- the main frame window

--inset = Insets(5,5,5,5)        -- margin for the ExtendendLabel object
inset = Insets(0,0,0,0)        -- margin for the ExtendendLabel object
ex = ExtendedLabel win.add('Center',-
                           ExtendedLabel('This is\na very special Test',inset,-
                           ExtendedLabel.CENTER,ExtendedLabel.CENTER ))

ex.setFont(Font("TimesRoman",Font.PLAIN,40)) -- change the font make it big

ex.addMouseListener(Mouse(ex))  -- used to get the mouse click on the label
win.pack                        -- set window client area to the preffered size of
                                -- the ExtendendLabel object

WindowSupport(win,null)         -- needed for closeWindow Support
RedbookUtil.positionWindow(win) -- center the window on the screen
win.setVisible(1)               -- show the window
```

Figure 78 (Part 1 of 2). Extended Label Test Application: ExtTest.nrx

```
-- class used to get the mouse events
class Mouse extends MouseAdapter  uses ExtendedLabel
  ex   = ExtendedLabel
  i   = 0                         -- internal counter

method Mouse(al = ExtendedLabel)  -- constructor
  ex   = al

method MouseClicked(e = MouseEvent)
  ex.setUpdate(0)
  if i // 2 = 0 then do                  -- change the color and font every mouse click
    ex.setForeground(Color.RED)
    ex.setFont(Font("Dialog",Font.BOLD,16))
  end
  else do
    ex.setForeground(Color.BLACK)
    ex.setFont(Font("TimesRoman",Font.PLAIN,26))
  end

  select                 -- change the horizontal alignment every mouse click
    when i // 3 = 0 then ex.setHorizontalAlignment(LEFT)
    when i // 3 = 1 then ex.setHorizontalAlignment(CENTER)
    otherwise ex.setHorizontalAlignment(RIGHT)
  end

  select                 -- change the vertical alignment every third mouse click
    when (i%3) // 3 = 0 then ex.setVerticalAlignment(TOP)
    when (i%3) // 3 = 1 then ex.setVerticalAlignment(CENTER)
    otherwise ex.setVerticalAlignment(BOTTOM)
  end

  ex.setUpdate(1)
  i = i + 1
```

Figure 78 (Part 2 of 2). Extended Label Test Application: ExtTest.nrx

Image Component

To draw an image in an application or an applet, you must subclass the paint method. This is very uncomfortable and does not allow the usage of a layout manager.

The image component is a lightweight component (see “Lightweight Components” on page 123) that loads and draws an image. This component can be used with a layout manager.

The image component can scale the image to the size of the component. The aspect ratio of the image is not changed when scaling occurs, so some empty space may remain in the component. The component has alignment attributes that define the alignment of the image within the component.

An Insets object can be used to define a margin for the component.

The getMinimumSize method returns the original size of the image, and the getPreferredSize method returns the original size, with the margin added.

The loading of the image is done by a MediaTracker object (see “Loading an Image” on page 118). The image is shown only when it is completely loaded.

The component accepts a file name, a URL, or an image as a parameter for the image. If an error occurs while loading the image, an exception (LoadImageException see “Exceptions” on page 271) is signaled.

Figure 79 shows the code of the ImagePanel class.

See “Photograph Album Sample Application” on page 150 for a sample application using the ImagePanel class.

```
/* redbook\gui\ImagePanel.nrx

   Implements a LightWeight component which shows an image .
   The component accepts an Insets object for a margin around the image.
   When scaling is on, the image always keeps its aspect ratio.
   Alignment can be specified; default is LEFT TOP */

package Redbook

class ImagePanel extends Component
  Properties constant public
    LEFT   = 0                -- alignment constants
    RIGHT  = 1
    CENTER = 2
    TOP    = 4
    BOTTOM = 5
  Properties inheritable
    im     = Image            -- image
    in     = Insets(0,0,0,0)  -- margins
    scale  = boolean 1        -- when true the image is scaled to component size
    halign = int LEFT         -- horizontal alignment
    valign = int TOP          -- vertical alignment

  -- default constructor no image loaded
  method ImagePanel()
    super()

  -- construct with an image and optional margins
  method ImagePanel(anImage = Image, newInsets = Insets null)
    super()
    setInsets(newInsets)
    setImage(anImage)

  -- construct with an image and optional margins
  method ImagePanel(anImageUrl = Url, newInsets = Insets null)-
         signals LoadImageException
    super()
    setInsets(newInsets)
    setImage(anImageUrl)

  -- construct with an image reading from given file
  method ImagePanel(imageFile = String, newInsets = Insets null)-
         signals LoadImageException
    super()
    setInsets(newInsets)
    setImage(imageFile)

  -- construct with an image and optional margins and optional alignments
  method ImagePanel(anImage = Image, hor_align = int,-
                    ver_align = int, newInsets = Insets null)
    this(anImage,newInsets)
    halign = hor_align
    valign = ver_align
```

Figure 79 (Part 1 of 4). Image Panel Class: ImagePanel.nrx

```
-- construct with an image reading from an URL and optinal alignments
method ImagePanel(anImageUrl = Url, hor_align = int,-
                  ver_align = int, newInsets = Insets null) -
       signals LoadImageException
  this(anImageUrl,newInsets)
  halign = hor_align
  valign = ver_align

-- construct with an image reading from given file and optinal alignments
method ImagePanel(imageFile = String, hor_align = int,-
                  ver_align = int, newInsets = Insets null) -
       signals LoadImageException
  this(imageFile,newInsets)
  halign = hor_align
  valign = ver_align

-- set the scaling of the image (true = scaling, false = noscaling)
method setScaling(b = boolean)
  scale = b

-- load a image from the image file and repaint if component is visible
method setImage(fileName = String) signals LoadImageException
  do
     imgUrl = this.getClass().getResource(fileName)
     im = Toolkit.getDefaultToolkit().getImage(imgUrl)
  catch NullPointerException       -- image file not found
     signal LoadImageException()
  end
  loadTheImage()

-- load a image from a given Url and repaint if component is visible
method setImage(anUrl = Url) signals LoadImageException
  im = Toolkit.getDefaultToolkit().getImage(anUrl)
  loadTheImage()

-- set the image to aImage and repaint if visible
method setImage( anImage = Image)
  im = anImage
  if isVisible() then repaint()

-- load the image and repaint if component is visible
method loadTheImage() signals LoadImageException inheritable
  tracker = MediaTracker(this)
  tracker.addImage(im,0)           -- load the image synchronous
  do
    tracker.waitForID(0)
  catch InterruptedException       -- dont handle the exception
  end
  if tracker.isErrorAny() then do
     im=null
     repaint()                     -- paint an empty panel
     signal LoadImageException()
  end
  if isVisible() then repaint()    -- repaint if visible

-- returns the current image
method getImage() returns Image
  return im
```

Figure 79 (Part 2 of 4). Image Panel Class: ImagePanel.nrx

```
-- sets the horizontal alignment method
method setHorizontalAlignment(align = int)
  halign = align
  repaint()

-- sets the vertical alignment method
method setVerticalAlignment(align = int)
  valign = align
  repaint()

-- returns the horizontal alignment method
method getHorizontalAlignment() returns int
  return halign

-- returns the vertical alignment method
method getVerticalAlignment() returns int
  return valign

-- sets the insets (margins) of the component
method setInsets( newInsets = Insets)
  if newInsets \= null then do
    in = Insets newInsets.clone()
    repaint()
  end

-- sets the insets (margins) of the component
method setInsets(itop=int,ileft=int,ibottom=int,iright=int)
  in.top = itop; in.left = ileft; in.right = iright; in.bottom = ibottom

-- returns the insets (margins) of the component
method getInsets() returns Insets
  return in

-- returns the size of the image with or without scaling
method getImageSize() returns Dimension
  if im  \= null then
    if scale then do
      d = getSize()
       -- preserving the aspect ration of the image
      lx = (d.width - in.left - in.right ) / im.getWidth(this)
      ly = (d.height - in.top - in.bottom) / im.getHeight(this)
      if lx < ly then ratio = lx
                 else ratio = ly
      return Dimension(ratio*im.getWidth(this)%1,ratio*im.getHeight(this)%1)
    end
    else return Dimension(im.getWidth(this),im.getHeight(this))
  else return Dimension(0,0)

-- returns the unscaled image size
method getOriginalImageSize() returns Dimension
 if im \= null then return Dimension(im.getWidth(this),im.getHeight(this))
               else return Dimension(0,0)

-- preferred size is unscaled image size plus insets
method getPreferredSize() returns Dimension
  d = getOriginalImageSize()
  d.width  = d.width + in.left + in.right
```

Figure 79 (Part 3 of 4). Image Panel Class: ImagePanel.nrx

```
    d.height = d.height+ in.top  + in.bottom
    return d

  -- minimum size is unscaled image size
  method getMinimumSize() returns Dimension
    return getOriginalImageSize()

  -- paint the image
  method paint(g=Graphics)
    w = getSize()
    if im  \= null then do
      d = getImageSize()
      select
        when valign = TOP    then y = in.top
        when valign = BOTTOM then y = w.height - d.height - in.bottom
        otherwise y = (w.height - in.top - in.bottom - d.height) % 2 + in.top
      end
      select
        when halign = LEFT  then x = in.left
        when halign = RIGHT then x = w.width - d.width - in.right
        otherwise x = (w.width - in.left - in.right - d.width) % 2 + in.left
      end
      g.drawImage(im,x,y,d.width,d.height,this)
    end
    else g.clearRect(0,0,w.width,w.height)
```

Figure 79 (Part 4 of 4). Image Panel Class: ImagePanel.nrx

A small test program, TestImage.nrx, is provided in the gui\imagepanel subdirectory.

The ImagePanel class is a member of the Redbook package.

Dialogs

All dialog windows have some common behavior. They are subclasses of the Dialog class, they have a set of buttons, and they should become visible at a position relative to the parent position on the screen.

This common behavior can be implemented by an abstract class that will be subclassed by the real implementations of dialog windows.

In this section we introduce the RedbookDialog class that implements the common behavior, a message box that shows a multiline text and an optional image, and a prompt dialog that is useful for asking for input from the user.

RedbookDialog Class

The RedbookDialog class inherits from Dialog.

A Dialog object needs a frame window as the parent in the constructor. If the parent is not a frame window, the class searches in the parent chain for a frame window. To enable message boxes for error messages of owner-written components, it is necessary to allow a component as the parent of the dialog. The RedbookDialog has two constructors; one constructor accepts only frame windows as a parameter, and one constructor accepts any component as a parameter but signals a NoFrameWindow exception if a frame window cannot be found in the parent chain.

Most dialogs have a set of buttons, which should have the same size. RedbookDialog creates an instance of an EqualSizePanel (see “Using Buttons of the Same Size” on page 133) and defines two methods to add buttons to the panel:

addButton(text=String) returns Button
: Creates a button with the specified text and adds an action listener to the button to close the dialog when the button is pressed

addButton(text=String,listener=ActionListener,closeTheDialog=boolean 1) returns Button
: Creates a button with the specified text, adds the given action listener to the button, and adds—if *closeTheDialog* is true—another action listener to the button to close the dialog when the button is pressed

The panel with the buttons is not yet added to the dialog box, because the final layout of the window is unknown.

The RedbookDialog class overrides the setVisible method (defined in Component) to pack the dialog to its preferred size and position it near the parent window.

Figure 80 shows the code for the Redbook Dialog.

```
/* redbook\gui\RedbookDialog.nrx

   The RedbookDialog class is an abstract class which provides basic services */

package Redbook

class RedbookDialog extends Dialog abstract
  Properties inheritable
    buttons = EqualSizePanel()  -- panel for the buttons
    parent  = Component         -- parent of the dialog
    ws      = WindowSupport     -- window support object attached to dialog

  -- constructor which searches for a frame window in the parent window chain
  method RedbookDialog(parentWindow = Component, modal = boolean, -
                       title = Rexx '')  signals NoFrameWindow
    super(RedbookUtil.findParentFrame(parentWindow),title,modal)
    init(parentWindow)

  -- constructor which gets a frame window as parent
  method RedbookDialog(parentFrame = Frame, modal = boolean,title = Rexx '')
    super(parentFrame,title,modal)            -- create the dialog
    init(parentFrame)

  -- initialize the basic dialog window
  method init(theParent = Component) inheritable
    do
      parent = theParent                      -- store the parent
      ws = WindowSupport(RedbookUtil.findParentWindow(parent),this,null)
      buttons.setGaps(10,10)                  -- horizontal and vertical gaps
    catch NoWindow                            -- impossible
    end

  -- add a button to the button panel and attach a window closer to it
  method addButton(text=String) returns Button
    newButton = Button(text)                  -- create new button
    newButton.addActionListener(ws.getCloseWindow()) -- use window support
    buttons.add(newButton)                    -- add the button to the panel
    return newButton

  -- add a button to the panel, attach the given action listener to it
  method addButton(text = String, listener = ActionListener, -
                   closeTheDialog = boolean 1) returns Button
```

Figure 80 (Part 1 of 2). Redbook Dialog Class: RedBookDialog.nrx

```
    newButton = Button(text)                    -- create new button
    newButton.addActionListener(listener)       -- add the action listener
    -- add a CloseWindow object from WindowSupport if closeTheDialog is true
    if closeTheDialog then newButton.addActionListener(ws.getCloseWindow())
    buttons.add(newButton)                      -- add the button to the panel
    return newButton

  -- make the dialog visible  or hide it  calculate the preferred size
  method setVisible(b=boolean)
    if b then do                                -- make dialog visible
      pack()                                    -- use preferred size
      RedbookUtil.positionWindow(parent,this) -- position relativ to parent
      super.setVisible(1)
    end
    else super.setVisible(0)                    -- hide dialog

  -- changes the closeWindow behaviour
  method setCloseBehaviour(newBehaviour = int)
    ws.setCloseBehaviour(newBehaviour)          -- set close behaviour in
                                                -- WindowSupport
```

Figure 80 (Part 2 of 2). Redbook Dialog Class: RedBookDialog.nrx

The RedbookDialog class is a member of the Redbook package.

Message Box

The MessageBox class implements a dialog window that shows text using an ExtendedLabel object, with an optional image and with one or more buttons. Figure 81 shows a message box created with the MessageBox class.

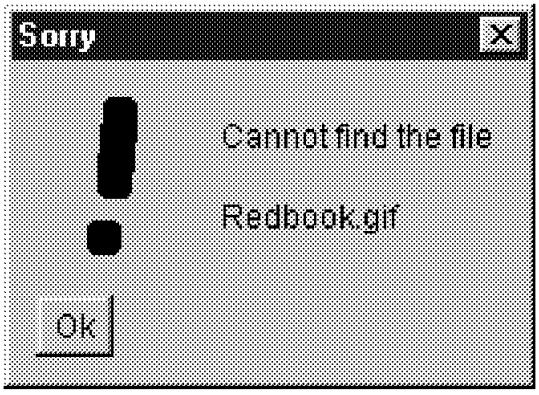


Figure 81. Sample Message Box

The MessageBox class inherits from the RedbookDialog class, and most of the methods defined in the class are constructors. The are two sets of constructors; one set accepts a frame window as a parent, and one set accepts any subclass of Component.

All constructors accept a button as the last parameter. An action listener that closes the window when the button is pressed is added to the button by default. If a button is not specified, a default **OK** button is used.

If the button parameter is null, a button is not added to the message box. Use the addButton methods of the RedbookDialog class if you need more than one button, or if you do not like the default.

The code to display the sample message box shown in Figure 81 is very simple:

```
/* gui\messagebox\TestBox.nrx */

import Redbook.
win = Frame("Test window")
Messagebox(win, "Sorry", "Cannot find the file\n\nRedbook.gif", "ex.gif").setVisible(1)
```

Figure 82 shows the code of the MessageBox class.

```
/* redbook\gui\MessageBox.nrx

   Creates a nonresizable Messagebox.

   If an image or image file is given the image is added on the left of the message.
   The message is a multi-line message.
   If a button text is specified, a button which closes the window is created.
   The messagebox is per default modal. */

package Redbook

class MessageBox extends RedbookDialog uses ExtendedLabel
 Properties inheritable
   img      = ImagePanel
   extlabel = ExtendedLabel

 -- creates a message box with an image loaded from a file
 -- the parent of the box is a frame window
 method MessageBox(theParent = Frame,title = String, message = String,-
                   imageFile = String, aButton = Button Button('Ok'))
   super(theParent,1,title)
   do
     img = ImagePanel add('West',ImagePanel(imageFile,Insets(10,10,0,0)))
   catch LoadImageException
   end
   initBox(message,aButton)

 -- creates a message box with a given image
 -- the parent of the box is a frame window
 method MessageBox(theParent = Frame,title = String, message = String,-
                   anImage = Image, aButton = Button Button('Ok'))
   super(theParent,1,title)
   img = ImagePanel add('West',ImagePanel(anImage,Insets(10,10,0,0)))
   initBox(message,aButton)

 -- creates a message box without any image
 -- the parent of the box is a frame window
 method MessageBox(theParent = Frame,title = String, message = String,-
                   aButton = Button Button('Ok'))
   super(theParent,1,title)
   initBox(message,aButton)

 -- creates a message box with an image loaded from a file
 -- the box looks for a parent frame window by it's own
 method MessageBox(theParent = Component,title = String, message = String,-
                   imageFile = String, aButton = Button Button('Ok')) -
                   signals NoFrameWindow
   super(theParent,1,title)
   do
     img = ImagePanel add('West',ImagePanel(imageFile,Insets(10,10,0,0)))
```

Figure 82 (Part 1 of 2). Message Box Class: MessageBox.nrx

```
    catch LoadImageException
    end
    initBox(message,aButton)

  -- creates a message box with a given image
  -- the box looks for a parent frame window on its own
  method MessageBox(theParent = Component,title = String, message = String,-
                    anImage = Image, aButton = Button Button('Ok')) -
                    signals NoFrameWindow
    super(theParent,1,title)
    img = ImagePanel add('West', ImagePanel(anImage,Insets(10,10,0,0)))
    initBox(message,aButton)

  -- creates a message box without any image
  -- the box looks for a parent frame window on its own
  method MessageBox(theParent = Component,title = String, message = String,-
                    aButton = Button Button('Ok')) signals NoFrameWindow
    super(theParent,1,title)
    initBox(message,aButton)

  -- intitialize the message box
  method initBox(message = Rexx,aButton = Button) inheritable
    if img \= null then in = Insets(10,5,0,10)
                   else in = Insets(10,10,0,10)
    extLabel = ExtendedLabel add('Center', ExtendedLabel(message,in,LEFT,CENTER))

    if aButton \= null then do
      aButton.addActionListener(super.ws.getCloseWindow())
      super.buttons.add(aButton)
    end

    add('South', super.buttons)
    setResizable(0)
```

Figure 82 (Part 2 of 2). Message Box Class: MessageBox.nrx

The MessageBox class is a member of the Redbook package.

Prompt Dialog

The PromptDialog class implements a dialog window that is used to prompt the user for input in a text field (see Figure 83).

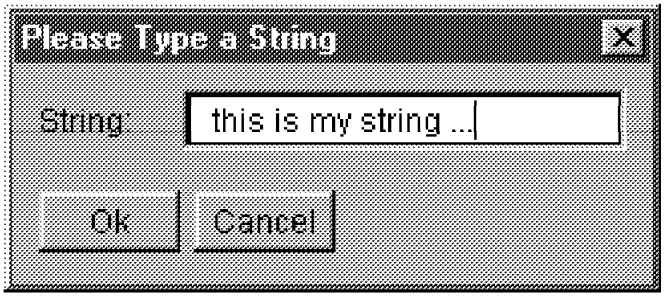


Figure 83. Sample Prompt Dialog

The PromptDialog class inherits from RedbookDialog and only a few methods are defined for the class. There are some methods to control, get, and set the text in the entry field. The init method uses the SimpleGridLayout manager (see Figure 62 on page 106).

Figure 84 shows the code of the PromptDialog class.

```
/* redbook\gui\PromptDialog.nrx

   Creates a prompt dialog box */

package Redbook

class PromptDialog extends RedbookDialog
  Properties inheritable
    entryField = TextField         -- prompt entry field
    keytester  = KeyCheck          -- key tester for the entry field
    keyLimit   = LimitTextField    -- maximum length of the entry field

  -- create the PromptDialog with any component as parent
  method PromptDialog(parentWindow = Component, modal = boolean, -
                      title = Rexx '',  labeltext = Rexx '',-
                      fieldText = Rexx '',  fieldsize = int 20)  signals NoFrameWindow
    super(parentWindow,modal,title)
    init(labeltext,fieldText,fieldsize)

  -- create the PromptDialog with a frame window as parent
  method PromptDialog(parentFrame = Frame, modal = boolean, -
                      title = Rexx '',  labeltext = Rexx '',-
                      fieldText = Rexx '',  fieldsize = int 20)
    super(parentFrame,modal,title)               -- create the dialog
    init(labeltext,fieldText,fieldsize)

  -- init the PromptDialog
  method init(labeltext = Rexx, fieldText = Rexx, fieldsize = int) private
    entryField = TextField(fieldText,fieldsize)
    keytester= KeyCheck(KeyCheck.ALL,0)          -- all keys no translation
    entryField.addKeyListener(keytester)         -- Add key tester to field
    super.ws.setFocusRecipient(entryField)       -- entryField gets the focus

    gridBag = SimpleGridBagLayout(this)          -- now the layout
    this.setLayout(gridBag)
    gridBag.addFixSize(Label(labeltext),0,0,Insets(10,10,5,5))
    gridBag.addVarSize(entryField,1,0,Insets(10,0,5,10),1,0)
    gridBag.addVarSize(super.buttons,0,1,1.0,1.0,GridBagConstraints.REMAINDER)

  -- Returns the value of the entry field
  method getText() returns Rexx
    return entryField.getText()

  -- set the focus to the entry field when visible
  method setVisible(show=boolean)
    super.setVisible(show)
    if show then
      entryField.requestFocus()

  -- set the comparison string for the key tester
  method setKeyMode(newMode = Rexx)
    keytester.setMode(newMode)

  -- if upperCase is true, lowercase letters are translated to uppercase
  method setUpperCase(upperCase = boolean)
    keytester.setMode(upperCase)

  -- set a limit for the length of the test in the entry field
```

Figure 84 (Part 1 of 2). Prompt Dialog Class: PromptDialog.nrx

```
  method setTextLimit(chars = int)
    if keyLimit \= null then entryField.removeTextListener(keyLimit)
    entryField.addTextListener(LimitTextField(entryField,chars))
```

Figure 84 (Part 2 of 2). Prompt Dialog Class: PromptDialog.nrx

The PromptDialog class is a member of the Redbook package.

A sample application that creates the prompt dialog box in Figure 83 on page 147 is shown in Figure 85.

```
/* gui\promptdialog\PromptTest.nrx

   Sample application to illustrate the use of the PromptDialog
   and the PromptDialogActionListener class.

   Shows a window with a push button and an label.
   The push button creates a prompt dialog, the label shows
   the value of the dialog when ready. */

import Redbook.

Gui('Prompt Dialog Test')                  -- start the User Interface

-- The Gui class implements the interface
class Gui implements PromptDialogAction
  Properties inheritable
    main  = Frame                          -- main window
    buttn = Button('Show Dialog')          -- push button
    answer= Label('Press the push button') -- label for the results

  method Gui(title = String)               -- create the interface
    main = Frame(title)

    -- construct the window
    main.add('North',answer)
    main.add('South',buttn)

    -- add ActionListener to the push button
    buttn.addActionListener(PromptDialogActionListener(this))

    -- show the window
    WindowSupport(main,buttn)
    main.setSize(200,150)
    RedbookUtil.positionWindow(main)
    main.setVisible(1)

    -- Invoked for the action listener when Prompt Dialog ready
    method promptReady(text = String,action = ActionListener)
      answer.setText('You entered:' text)

    -- invoked from the Actionlistener, build the Prompt Dialog Box
    method getPromptDialog(action = ActionListener)-
           returns PromptDialog
       dialogw = PromptDialog(main,0,'Please Type a String','String:')
       dialogw.addButton('Ok',action)
       dialogw.addButton('Cancel')
       return dialogw
```

Figure 85. Sample Prompt Dialog Application: PromptTest.nrx

The application defines a GUI class that implements a PromptDialogAction interface. The interface defines two methods:

```
/* gui\promptdialog\PromptDialogAction.nrx */
Package Redbook
class PromptDialogAction interface
  method promptReady(text = String, source = ActionListener)
  method getPromptDialog(source=ActionListener) returns PromptDialog
```

The promptReady method is called when the **OK** button is clicked. The parameters are the text and the action listener itself. The action listener is part of the parameter list to distinguish between two or more action listeners in the same method.

Figure 86 shows the action listener used by the interface.

```
/* redbook\gui\PromptDialogAction.nrx

   Interface to work with the PromptDialogActionListener.
   The method promptReady it invoked when the prompt dialog returns a value.
   The method getPromptDialog is invoked when the PromptDialog is constructed. */

Package Redbook

class PromptDialogAction interface
 method promptReady(text = String, source = ActionListener)
 method getPromptDialog(source=ActionListener) returns PromptDialog
```

Figure 86. Prompt Dialog Action Listener: PromptDialogAction.nrx

The PromptDialogAction class is a member of the Redbook package.

Photograph Album Sample Application

The photograph album application illustrates the use of ImagePanels, EqualSizePanels, and buttons.

The application shows a window with two sets of buttons and an ImagePanel (see Figure 87).

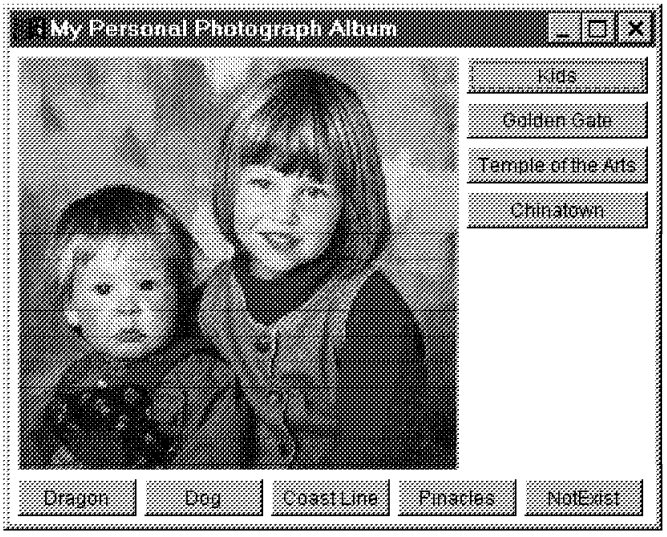


Figure 87. Photograph Album Sample Application

When a button is pressed the corresponding image is loaded. If an image file is not found, a message box is displayed to inform the user.

The application consists of a script part (see Chapter 5, “Using NetRexx As a Scripting Language” on page 49), and an action listener that stores a reference to the image panel and the name of the image.

You can use such a design as long as methods are not called from the user interface. This is true for many small applications. If the application becomes more complex, you should use a design as shown in the PromptDialog sample application (“Prompt Dialog” on page 147).

Figure 88 shows the code for the application.

```
/* gui\photoalbum\PhotoAlbum.nrx

   Sample application to illustrate the use of the EqualSizePanel and ImagePanel class.
   This application is a little photograph album.
   It has a row and a column of buttons to show the pictures. */

 import Redbook.

 win = Frame('My Personal Photograph Album')

 p1 = EqualSizePanel(EqualSizePanel.VERTICAL)  -- two panels for the buttons
 p2 = EqualSizePanel(EqualSizePanel.HORIZONTAL)
 pict = ImagePanel()                           -- ImagePanel for the pictures

 win.add('East',p1)                            -- add the panels to the window
 win.add('South',p2)
 win.add('Center',pict)

 -- add the buttons to the panel and add an ActionListener to every button
 (Button p1.add(Button('Kids'))).addActionListener(ShowPicture(pict,'Redbook/kids.jpg'))
 (Button p1.add(Button('Golden Gate'))).addActionListener(ShowPicture(pict,'Redbook/gate1.jpg'))
 (Button p1.add(Button('Temple of the Arts'))).addActionListener(-
                                                  ShowPicture(pict,'Redbook/temple.jpg'))
 (Button p1.add(Button('Chinatown'))).addActionListener(ShowPicture(pict,'Redbook/chinat.jpg'))
 (Button p2.add(Button('Dragon'))).addActionListener(ShowPicture(pict,'Redbook/dragon.jpg'))
 p1.setGaps(5,5)

 (Button p2.add(Button('Dog'))).addActionListener(ShowPicture(pict,'Redbook/dog.jpg'))
 (Button p2.add(Button('Coast Line'))).addActionListener(ShowPicture(pict,'Redbook/lighthou.jpg'))
 (Button p2.add(Button('Pinacles'))).addActionListener(ShowPicture(pict,'Redbook/pinac.jpg'))
 (Button p2.add(Button('NotExist'))).addActionListener(ShowPicture(pict,'Redbook/NotExist.jpg'))
 p2.setGaps(5,5)

 pict.setInsets(5,5,0,0)                       -- margin for the ImageLabel panel

 WindowSupport(win,null)                       -- CloseWindow Support

 win.setSize(400,250)                          -- set the size and show the window
 RedbookUtil.positionWindow(win)
 win.setVisible(1)


-- ShowPicture stores the name of the image and shows it when an
-- ActionEvent is received
class ShowPicture implements ActionListener
  Properties inheritable
    fname = String                             -- name of the picture file
    pict  = ImagePanel                         -- reference to the ImagePanel

  method ShowPicture(aPict = ImagePanel, aname = String)
    fname = "/"aname
    pict  = aPict

  method actionPerformed(e=ActionEvent)
  do
    pict.setImage(fname)                       -- set the image in the image panel
  catch ex = LoadImageException                -- caught if file not found
    say 'Exception...' ex
    pframe = Frame pict.getParent()            -- get the frame window
    mb=MessageBox(pframe,'Sorry','Cannot find the file\n\n'fname,-
                  '/EX.gif')
    mb.setVisible(1)
  end
```

Figure 88. Photograph Album Sample Application: PhotoAlbum.nrx

Chapter 8. Threads

One of the strongest advantages of Java is its built-in thread support. Most programming languages allow the use of threads, but the threads are part of the operating system and not part of the language.

A thread is a single sequential flow of control within a process. Threads run parallel to each other. A program is a collection of threads. Threads share the same address space of the program.

Only a few basic classes and constructs are specially designed to support threads:

- The *Thread* class, along with some related utility classes, used to initiate and control new threads
- The *wait*, *notify*, and *notifyAll* methods, defined in the Object class
- The *protect* and *volatile* keywords, used to control execution of code in objects that are used by threads

The Thread Class

A new thread is created when we create an instance of the Thread class. We cannot tell a thread which method to run, because threads are not references to methods. Instead we use the *Runnable* interface to create an object that contains the *run* method:

```
class Runnable interface public
  method run()
```

Every thread begins its concurrent life by executing the run method. The run method does not have any parameters, does not return a value, and is not allowed to signal any exceptions.

Any class that implements the Runnable interface can serve as a target of a new thread. An object of a class that implements the Runnable interface is used as a parameter for the thread constructor:

```
class Background implements Runnable
  ...
bkg = Background()
aThread = Thread(bkg)
```

You can give a thread an optional name that is visible when listing the threads in your system. It is good practice to name every thread, because if something goes wrong you can get an idea which threads are still running.

Additionally, threads are grouped by thread groups. If you do not supply a thread group, the new thread is added to the thread group of the currently executing thread. The threads of a group and their subgroups can be destroyed, stopped, resumed, or suspended by using the ThreadGroup object.

The thread name and group are specified at construction time of the thread. The constructors of the Thread class are:

Thread(target=Runnable,threadName=String '')
: Creates a thread with the given name that is a member of the current thread group

Thread(group=ThreadGroup,target=Runnable,threadName=String '')
: Creates a thread with the given name that is a member of the given thread group

More constructors without the Runnable object are available. We do not list them here because we do not subclass the Thread class to create a new thread.

Creating and Starting Threads

A newly created thread remains idle until the start method is invoked. The thread then wakes up and executes the run method of its target object. The start method can be called only once. The thread continues running until the run method completes or the stop method of the thread is called. Figure 89 shows the life cycle of a thread.

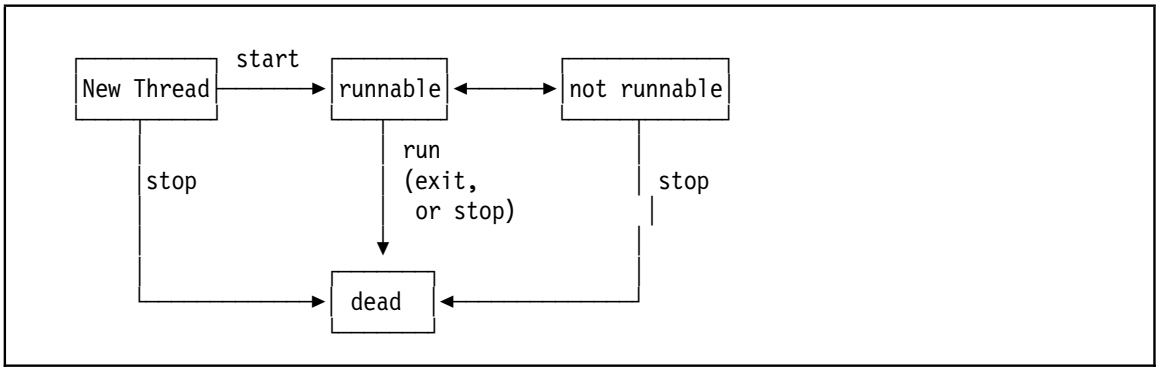


Figure 89. Life Cycle of a Thread

Figure 90 shows a simple example of the use of threads.

```
/* thread\ThrdTst1.nrx */

h1 = Hello1('This is thread 1')
h2 = Hello1('This is thread 2')

Thread(h1,'Thread Test Thread 1').start()
Thread(h2,'Thread Test Thread 2').start()

-------------------------------- Hello1 class implements RUNNABLE
class Hello1 implements Runnable
  Properties inheritable
    message = String

  method Hello1( s = String)
    message = s

  method run()
    loop for 50
```

Figure 90 (Part 1 of 2). Simple Application with Multiple Threads: ThrdTst1.nrx

```
      say message
      Thread.currentThread().yield() -- for OS/2 or Window 95/NT not necessary
    end
```

Figure 90 (Part 2 of 2). Simple Application with Multiple Threads: ThrdTst1.nrx

The Thread class itself implements the Runnable interface. Therefore it is possible to inherit from Thread instead of implementing the Runnable interface (see Figure 91).

```
/* thread\ThrdTst2.nrx */

h1 = Hello2('This is thread 1')
h2 = Hello2('This is thread 2')

h1.start()
h2.start()

-------------------------------- Hello2 class extends THREAD
class Hello2 extends Thread
  Properties inheritable
    message = String

  method Hello2( s = String)
    super('Thread Test - Message' s)
    message = s

  method run()
    loop for 50
      say message
      do
        sleep(10)
        catch InterruptedException
      end
    end
```

Figure 91. Simple Application with Multiple Threads: ThrdTst2.nrx

Subclassing the Thread class may be convenient, but most of the time it is not correct because the Hello class is a class that prints a string and can run as a thread, but it is not a thread (see "Usage or Inheritance" on page 66).

Controlling Threads

The start method is used to start a newly created thread. Three other methods give us control over a thread:

- The stop method destroys the thread. The stop method can be called only once during the life of a thread.
- The suspend method pauses the thread. The thread is suspended until the resume method or the stop method is called.
- The resume method resumes a suspended thread.
- The sleep method suspends the thread for a number of milliseconds.

Each of these methods works on the current thread object. If the current thread object is unknown, it can be received by using a class method of the Thread class:

```
Thread.currentThread()
```

This method returns the current thread object.

Lifetime of a Thread

A thread continues to execute until:

- It returns from the run method.
- The stop method is called.
- An exception occurs and is not caught.

If a thread does not terminate, and the application that started the thread does not call the stop method, the thread lives on, even if the application has finished. The exit statement ends all threads of an application.

Use the setDaemon method to mark a thread as a daemon thread that should be killed and discarded when no other application thread remains.

Scheduling

Java makes some guarantees about how a thread is scheduled. The Java interpreter dispatches threads, using an algorithm that schedules threads on the basis of their priority relative to other runnable threads. The setPriority method is used to change the priority of a thread at run time. The priority must be in the range of Thread.MIN_PRIORITY and Thread.MAX_PRIORITY. When multiple threads are ready to be executed, the run-time system chooses the thread with the highest priority. If two threads have the same priority, the scheduler chooses one of them in a round-robin fashion.

The scheduling is also *preemptive*. If at any time a thread with a higher priority than any other runnable thread becomes runnable, the scheduler chooses that thread for execution.

A thread continues running until it gives up control through one of the following actions:

- Calls the wait or sleep methods
- Calls the suspend method
- Calls the yield method
- Is blocked by I/O
- Terminates

The Java specification for scheduling is not fully defined. Some details can be done differently from implementation to implementation. The main difference is that some Java implementations use time slicing on threads with the same priority. In a time-slicing system, each thread runs for a short period of time before Java switches to the next thread. Higher priority threads still preempt lower priority threads.

Time slicing is implemented in the OS/2, Windows 95, and Window NT Java interpreter. Round-robin scheduling is implemented for Sun′s port of the Solaris Java interpreter.

Because time slicing is not implemented in all interpreters, your code should not rely on such scheduling. For example, if you delete the call to the yield method in the run method in Figure 90 on page 154, the result would show only the “This is thread 1” message with a scheduler using the round-robin scheme.

Synchronization

When two or more threads access the same object, they must be synchronized. Java provides synchronization based on the concept of monitors, a widely used synchronization scheme developed by C.A.R Hoare ("Communicating Sequential Processes," *Communications of the ACM,* Vol. 21, No. 8, August 1978).

Monitors and the Protect Keyword

A monitor is essentially a lock. If the resource is not used, the thread can acquire the lock and access the resource. When the work is done, the thread relinquishes the lock. When threads try to access a resource that is locked, they have to wait until the resource is unlocked.

A lock is set by the *protect* keyword, which can be used to protect a method of an object or access to an object.

A method is protected when the protect keyword in the method declaration is used (see "Method Instruction" on page 28):

```
method lockedMethod() protect
...
```

When the protect keyword is used in the *do*, *select*, or *loop* instruction, an object that is locked is used as a parameter (see "Control Statements" on page 39):

```
-- test is an object of any type
do protect test
   ...
end
```

Any access to the *test* object is locked as long the do block is executed.

The protect keyword in the method statement can be simulated by using a lock for the object:

```
method lockedMethod()
  do protected this   -- same as method lockedMethod() protect
     ...
  end
```

Threads that do not apply a lock have full access to any object, even if another thread locked the object.

Locks are reentrant. If a thread holds a lock on a resource, it can reenter the code section that acquired a lock to the same resource.

Wait and Notify

When the wait method in a protected block is used, a thread goes to sleep. To wake up the thread, use the notify method from a protected block protecting the same object:

```
class NotifyExample
  method myWait() protect
    ...    -- do something
    do
       wait()
     catch InterruptedException
    end
    say 'released'
```

```
method myNotify() protect
   ...    -- do something
   notify()
   ...
```

The myWait method is suspended when the wait method is executed. It is resumed when another thread uses the myNotify method.

When more than one thread is waiting for the same object, notify awakes only one thread. The run-time system chooses this thread in a nonguaranteed fashion.

With the notifyAll method, all waiting threads are notified.

Additionally, there is a wait method with a timeout:

```
wait(ms=int)
```

The wait and notify methods are used by threads that are dependent on each other. They offer a way of signaling that an object for which another thread is waiting has changed.

The example in Figure 92 illustrates the use of the wait and notify methods. The producer thread reads messages from the console input. The messages are stored in an array that is limited to holding four messages.

The consumer thread gets the messages from the array and prints them on the screen. To make the scenario more realistic, the consumer thread waits for a few seconds after each message.

```
/* thread\consumer\Consumer.nrx

   Consumer - Producer sample application with wait and notify to synchronize */

------------------------------------- Consumer class

-- name the current thread
Thread.currentThread().setName('Thread Example: Consumer Thread')

p = Producer()                                    -- creates a producer
t = Thread(p,'Thread Example: Producer Thread')   -- creates the thread
t.start()                                         -- runs the producer

s = Rexx ''

loop until s = 'exit'
  Thread.currentThread().sleep(5000) -- wait 5 seconds
  s = p.readMessage()
  say 'Message received' s
end

say 'Program stopped'
exit 0

------------------------------------- Producer class
class Producer implements Runnable
  Properties constant
    MaxEntries = int 4
  Properties inheritable
    queue = Vector()         -- dynamic growing Array
```

Figure 92 (Part 1 of 2). Threads with Wait and Notify: Consumer.nrx

```
method run()
 getMessages()

-- producer method
method getMessages()
 loop forever
   say 'Type new message ('queue.size() 'messages in queue)'
   newMessage = ask
   do protect this      -- protected against readMessages
     -- check if the loop is full
     loop while queue.size() == MaxEntries
        say 'Queue is full'
        wait()
      catch InterruptedException
     end
     queue.addElement(newMessage)  -- add the message
     notify()                      -- notify readMessage: new message
   end
 end

-- called from consumer
method readMessage() protect returns Rexx
 loop while queue.size() == 0
    wait()
  catch InterruptedException
 end
 s = Rexx queue.firstElement()    -- read the first element
 queue.removeElement(s)           -- remove the element from queue
 notify()                         -- notify getMessages: message removed
 return s
```

Figure 92 (Part 2 of 2). Threads with Wait and Notify: Consumer.nrx

Note: Enter `exit` to stop the threads.

Philosophers' Forks

To complete the chapter on threads, we look at an ancient and famous example, the philosophers' forks problem.

Philosophers' forks

- Five philosophers sit around a table. Each one goes through a cycle of sleeping and eating.
- There is a fork between each two philosophers, so there are five forks on the table as well.
- To eat, a philosopher has to grab the forks on both sides. If a fork has already been taken by the philosopher on the other side of the fork, the philosopher must wait until that fork is available.
- The philosophers reach for forks in no particular order, but once they reach out for a fork and have to wait, they do not change their minds, even if the other fork is available.
- When they have finished eating, the philosophers put down both forks and go back to sleep.
- The times that they sleep and eat vary randomly around given values.

Designing the Philosophers' Forks

We approach this problem with two classes, one for the philosophers and one for the forks:

- The philosophers are the independent threads, each running through the cycle of sleeping and eating.
- The forks are the accessories, and access to each fork must be synchronized so that only one philosopher can pick up the fork.
- We use the forks as *monitor* objects and design pickup and laydown methods with the *protect* keyword.
- We invoke *wait* and *notify* to control access to the forks.

Figure 93 shows the implementation of the class for the philosophers.

```
class PhilText extends Thread

 properties private
   num   = int                              -- number
   lfork = ForkText                         -- left fork
   rfork = ForkText                         -- right fork
   out   = Rexx                             -- prefix

 method PhilText(nump=int, lforkp=ForkText, rforkp=ForkText)
   num   = nump
   lfork = lforkp
   rfork = rforkp
   out   = ' '.copies(15*num-14)

 method run() public                        -- run the philosop.
   say out 'Philosopher-'num
   loop for PFtext.cycles                   -- run the loop
      stime = PFtext.sleep % 2 + PFtext.sleep * Math.random() % 1
      say out 'Sleep-'(Rexx stime/1000).format(2,1)
      sleep(stime)                          -- sleep
      say out 'Wait'
      if Math.random() <= PFtext.side then do -- pick up forks
         lfork.pickup()
         rfork.pickup()
         end
      else do                               -- same, right
         rfork.pickup()
         lfork.pickup()
      end
      etime = PFtext.eat % 2 + PFtext.eat * Math.random() % 1
      say out 'Eat-'(Rexx etime/1000).format(2,1)
      sleep(etime)                          -- eat
      lfork.laydown()                       -- lay down forks
      rfork.laydown()
    catch InterruptedException
   end
   say out 'Done'                           -- loop finished
```

Figure 93. Philosophers' Forks: Philosopher Class

The philosophers are subclasses of Thread; therefore they implement the *run* method that is invoked when the thread is started. They go through the cycle of sleeping and eating. To eat they pick up the two assigned forks.

Figure 94 shows the implementation of the class for the forks.

```
class ForkText

 properties private
   used  = boolean 0                          -- flag is fork in use

 method ForkText()                            -- constructor

 method pickup protect                        -- pickup the fork
   if used then wait()                        -- wait for fork
   used = boolean 1                           -- take it

 method laydown protect                       -- laydown the fork
   used = boolean 0                           -- set to free
   notify()                                   -- let others run
```

Figure 94. Philosophers′ Forks: Fork Class

The fork class provides the synchronization using a boolean variable that indicates whether the fork is in use. The *pickup* method waits if the fork is occupied, and the *laydown* method notifies another thread that is waiting.

Both methods use the protect keyword to lock the current object, that is, the fork itself.

To complete the example we need the main program that prepares the parameters, allocates the philosophers and the forks, and starts the five threads (see Figure 95).

```
/* thread\philfork\PFtext.nrx

   Philosophers Forks in a text Window */

class PFtext public

 properties static public
   eat    = int 6000                          -- eat time (ms)
   sleep  = int 8000                          -- sleep time (ms)
   cycles = int 2                             -- number of cycles
   side   = double 0.5                        -- fork pickup (L,R,random)

 method main(args=String[]) static
   nuarg = args.length
   if nuarg > 3 then side   = Rexx args[3]
   if nuarg > 2 then cycles = Rexx args[2]
   if nuarg > 1 then eat    = (Rexx args[1]) * 1000
   if nuarg > 0 then sleep  = (Rexx args[0]) * 1000
   if      side = 'L' then side = 1.0         -- left fork first
   else if side = 'R' then side = 0           -- right
                      else side = 0.5         -- random
   run()                                      -----------> RUN IT

 method run() static
   f1 = ForkText()                            -- create 5 forks
   f2 = ForkText()
   f3 = ForkText()
   f4 = ForkText()
   f5 = ForkText()
   p1 = PhilText(1,f5,f1)                     -- create 5 philos.
   p2 = PhilText(2,f1,f2)
   p3 = PhilText(3,f2,f3)
   p4 = PhilText(4,f3,f4)
   p5 = PhilText(5,f4,f5)
```

Figure 95 (Part 1 of 2). Philosophers′ Forks: Main Program: PFtext.nrx

```
  p1.start()                              -- run 5 philsophers
  p2.start()
  p3.start()
  p4.start()
  p5.start()
```

Figure 95 (Part 2 of 2). Philosophers′ Forks: Main Program: PFtext.nrx

When we run the program, we can see the threads executing in parallel (see Figure 96).

```
d:\NrxRedBk\thread\philfork>jave PFtext

 Philosopher-1
                Philosopher-2
                                Philosopher-3
                                                Philosopher-4
                                                                Philosopher-5
 Sleep- 5.8
                                                                Sleep- 7.5
                Sleep- 7.3
                                                Sleep- 4.7
                                Sleep- 9.8
                                                Wait
                                                Eat- 6.4
 Wait
 Eat- 3.1
                Wait
                                                                Wait
 Sleep- 5.8
                Eat- 7.3
                                Wait
                                                Sleep- 7.7
                                                                Eat- 4.3
 Wait
                                                                Sleep-11.6
                                Eat- 3.3
                Sleep-11.3
 Eat- 9.0
                                                Wait
                                Sleep- 4.8
                                                Eat- 4.6
                                                Done
                                Wait
                                Eat- 5.6
 Done
                                                                Wait
                                                                Eat- 4.5
                Wait
                                Done
                Eat- 4.3
                                                                Done
                Done
```

Figure 96. Philosophers′ Forks: Execution in a Text Window

Enhancing the Philosophers′ Forks with a GUI

To better visualize the parallel threads, we can design a solution using a GUI.

We represent the philosophers and forks as push buttons that can be visible or hidden, and colored according to the activity. We use red for eating, white for waiting, and gray for sleeping. We use little push buttons to represent the hands that hold the forks, and we put pieces of a central cake in front of the philosophers when they eat. We also add push buttons to start and stop the animation.

The basic logic of the philosopher and fork classes does not change. We only introduce a few methods to interact with the GUI object to change the visibility and color of all push buttons that represent our model objects.

Most of the code is needed to define all of the GUI objects with position and size and place them into the applet. The program is called PFgui.nrx, and it runs as an applet or an application. The HTML file for the applet is called PFgui.htm. Both files are in the thread\philfork subdirectory.

Figure 97 shows a snapshot of executing the GUI program.

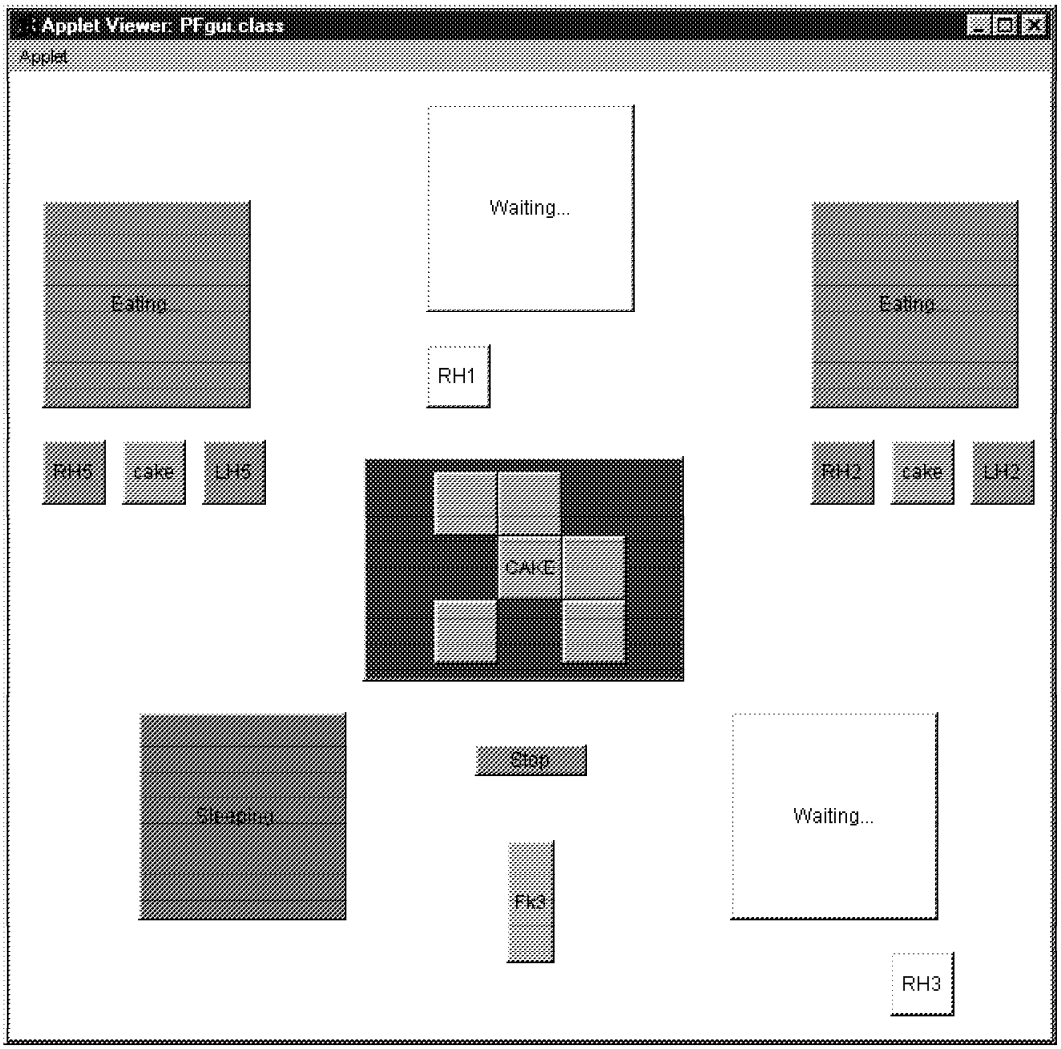


Figure 97. Philosophers′ Forks: Execution in a GUI

Chapter 9. Handling Files

In this chapter we discuss how to read from and write to files using line-mode, byte-oriented, data-oriented, and object-oriented streams.

There are no specifications for I/O in NetRexx Version 1.0. (This is similar to classic Rexx Version 1.0; the I/O specification was added for Version 2.0.) Because NetRexx does not define its own I/O statements, I/O-related functions have to be implemented using the Java class library.

Note: NetRexx does provide the *say* instruction to write to standard output and the *ask* function to get input from the user.

Streams

A stream is a flowing sequence of bytes. In Java, an object from which you can read a sequence of bytes is called an *input stream*. An object to which you can write a sequence of bytes is called an *output stream*. The Java I/O class library provides more than 20 types of streams.

You can connect an input stream to many sources, for example, a file or a TCP/IP socket of a network. You can connect an output stream to many destinations, for example, a file, a printer, or a TCP/IP socket. Streams provide the generalized I/O mechanism that you can use to handle both files and network connections.

JDK 1.1 introduces support for character streams to the java.io package. Character streams are mainly for internationalization support. Before this release, JDK supported byte streams only, through the InputStream and OutputStream classes and their subclasses.

Character streams are like byte streams, but they contain 16-bit Unicode characters rather than 8-bit bytes. They are implemented by the Reader and Writer classes and their subclasses. Reader and Writer classes are enhanced for performance. Our sample programs use Reader and Writer classes where possible.

File Class

To handle a file or a directory, you create a file object. The File class provides many methods for getting the file or directory information and manipulating files and directories. The first sample program, FileInfo, shows how to use these methods (see Figure 98).

```
/* file\FileInfo.nrx

   Display file/directory/path information */

   parse arg fileName .
   if fileName = "" then do
      say "Enter file or directory name to test ?"
      filename = ask
   end
    f1 = File(fileName)                       -- create file object
   if f1.exists() = 0 then do
      say 'File:' filename 'does not exist.'
      exit 8
   end

   say "System related information ----------------------"
   say "  pathSeparator     :" f1.pathSeparator        -- these are not methods
   say "  pathSeparatorChar :" f1.pathSeparatorChar    -- they are public
   say "  separator         :" f1.separator            --          static
   say "  separatorChar     :" f1.SeparatorChar        --          class variables
   say
   say "File/directory related information --------------"
   say "  canRead           :" f1.canRead()
   say "  canWrite          :" f1.canWrite()
   say "  isDirectory       :" f1.isDirectory()
   say "  isFile            :" f1.isFile()
   say "  length            :" f1.length()
   say "  lastModified      :" f1.lastModified() "=" Date(f1.lastModified())
   say "  isAbsolute        :" f1.isAbsolute()
   say "  getAbsolutePath   :" f1.getAbsolutePath()
   say "  getCanonicalPath  :" f1.getCanonicalPath()
   say "  getPath           :" f1.getPath()

   parent1 = f1.getParent()
   if parent1 = null then parent1 = "null returned"
   say "  getParent         :" parent1
   say "  getName           :" f1.getName()
   say "  toString          :" f1.toString()
   say "  hashCode          :" f1.hashCode()

   if f1.isDirectory() then do
     say
     say "List of this directory ----------------------------\n"
     list1 = f1.list()
     if list1.length = 0
        then say "  directory is empty"
        else
          loop i = 0 to list1.length -1
             f2 = File(f1.getAbsolutePath()''f1.separator''list1[i])
             if f2.isDirectory() then say "  Dir :" list1[i]
```

Figure 98 (Part 1 of 2). Display File and Directory Information: FileInfo.nrx

```
                                else say "  File:" list1[i]
          end
     end
     say "\n-------------------------------------------------"
     -- end fileinfo
```

Figure 98 (Part 2 of 2). Display File and Directory Information: FileInfo.nrx

Try the program for various cases and see the results; for example:

```
java FileInfo test.dat                   ==> file
java FileInfo testdir                    ==> directory
java FileInfo d:\dir1\dir2\test.dat       ==> file with absolute path
java FileInfo d:\dir1\dir2\testdir        ==> directory with absolute path
```

This is a sample output listing of the program:

```
d:\NrxRedBk\file>java FileInfo d:\nrxredbk\Thread
System related information ----------------------
  pathSeparator     : ;
  pathSeparatorChar : ;
  separator         : \
  separatorChar     : \

File/directory related information --------------
  canRead           : 1
  canWrite          : 1
  isDirectory       : 1
  isFile            : 0
  length            : 0
  lastModified      : 869011058000 = Tue Jul 15 16:57:38 PDT 1997
  isAbsolute        : 1
  getAbsolutePath   : d:\nrxredbk\Thread
  getCanonicalPath  : d:\NrxRedBk\thread
  getPath           : d:\nrxredbk\Thread
  getParent         : d:\nrxredbk
  getName           : Thread
  toString          : d:\nrxredbk\Thread
  hashCode          : -1850736099

List of this directory ---------------------------

  Dir : consumer
  File: ThrdTst2.nrx
  File: ThrdTst1.nrx
  File: Hello2.class
  File: ThrdTst2.class
  File: Hello1.class
  File: ThrdTst1.class
  Dir : philfork
  Dir : synch

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

There are a few additional methods for manipulating or testing a file or directory:

delete()	Deletes the file or directory specified by this object
equals(Object)	Compares this object against the specified object
list(FilenameFilter)	Returns a list of the files in the directory that satisfy the specified filter
mkdir()	Creates a directory whose path name is the file object

mkdirs()	Creates a directory whose path name is the file object, including any necessary parent directories
renameTo(File)	Renames the file or directory to have the path name given by the file argument

Line Mode I/O

In most cases, we deal with a file line by line. Therefore, we discuss line mode I/O first.

Line I/O Using BufferedReader and PrintWriter

The BufferedReader class facilitates reading files, with good performance, using the *readLine* method.

The PrintWriter class has an improved *println* method that adds the platform-dependent line-end character to the end of the string. When you use this method under OS/2, Windows 95, or Windows NT, new line (\n) and carriage return (\r) characters are added. Under UNIX systems, only a new line (\n) character is added.

Note: In JDK 1.0.2, the println method of the PrintStream class adds the new line (\n) character only. Therefore, you may not get the exact same copy of a file, using readLine and println.

Our first sample program uses a BufferedReader and a PrintWriter class to extract *DEVICE* statements from a file (see Figure 99).

```
/* file\LineIO.nrx

   Line-mode I/O using buffered reader and printer writer.
   Extract 'DEVICE' statements from a file (default CONFIG.SYS).  */

   parse arg filename
   if filename = '' then filename = 'C:/CONFIG.SYS'
   output = 'CONFIG.DEV'
   say 'File:' filename '->' output

   inFile  = FileReader(filename)                  -- input file
   source  = BufferedReader(inFile)                -- buffered

   outFile = FileWriter(output)                    -- output file
   dest    = PrintWriter(outFile)                  -- to printer

   loop forever
      textline = source.readLine()                 -- read the file
      if textline = null then leave                -- end-of-file ?
      parse textline word1 '=' .
      if word1 = "device" then do                  -- DEVICE statement ?
         dest.println(textline)                    -- write output
      end
   end

   source.close()                                  -- close files
```

Figure 99 (Part 1 of 2). Buffered Input and Print Output: LineIO.nrx

```
    dest.close()

    say 'Extracted DEVICE statements:'               -- display results
    source = BufferedReader( FileReader(output) )    -- read output file
    loop until textline = null
        textline = source.readLine()
        if textline \= null then say ' ' textline    -- standard output
    end
-- end LineIO
```

Figure 99 (Part 2 of 2). Buffered Input and Print Output: LineIO.nrx

The string comparison operation with one equal sign is not case sensitive. This feature of NetRexx, combined with the power of the parse instruction, allows easy extraction of DEVICE statements of various coding, for example:

```
DEVICE=C:\driver1.sys
device=C:\driver2.sys
DeViCe = C:\driver3.sys
DEVICE = C:\driver4.sys
```

Line I/O Using BufferedReader and BufferedWriter

When using a BufferedWriter instead of the PrintWriter, you have to add the new line characters, using the *newline* method. On OS/2 and Windows systems, this method adds the (\n)(\r) characters.

Figure 100 shows the modified program.

```
/* file\LineIO2.nrx

   Line-mode I/O using buffered reader and buffered writer.
   Extract 'DEVICE' statements from a file (default CONFIG.SYS).  */

   parse arg filename
   if filename = '' then filename = 'C:/CONFIG.SYS'
   output = 'CONFIG.DEV'
   say 'File:' filename '->' output

   inFile  = FileReader(filename)                  -- input file
   source  = BufferedReader(inFile)                -- buffered

   outFile = FileWriter(output)                    -- output file
   dest    = BufferedWriter(outFile)               -- buffered              <===

   loop forever
      textline = source.readLine()                 -- read the file
      if textline = null then leave                -- end-of-file ?
      parse textline word1 '=' .
      if word1 = "device" then do                  -- DEVICE statement ?
         dest.write(textline,0,textline.length())  -- write output          <===
         dest.newline                              -- add new line char     <===
      end
   end
```

Figure 100 (Part 1 of 2). Buffered Input and Buffered Output: LineIO2.NRX:

```
    source.close()                                  -- close files
    dest.close()
  -- end LineIO2
```

Figure 100 (Part 2 of 2). Buffered Input and Buffered Output: LineIO2.NRX:

Byte-Oriented I/O

Byte-oriented output allows you to read (and write) one byte at a time. Although byte-oriented I/O is used infrequently in real applications, it is important to understand a few basic concepts.

We use the DataInputStream class to read single bytes, using the *readUnsignedByte* method. A DataInputStream object is constructed from a FileInputStream object that in turn is constructed from a file object.

Our sample program (see Figure 101) reads a file and dumps it in the “classic” IBM dump format, 16 characters to a line, with printable characters shown to the right of the hexadecimal dump. We use this program in “Data-Oriented I/O” on page 172 to check the contents of a file with binary data.

```
/*  file\HexPrint.nrx

    Print file content in classic hexadecimal dump format.
    Parameter: inputfile   */

    parse arg inFileName .
    if inFileName = '' then do
      say "Usage: HexPrint fileName"
      exit 8
    end

    infile = File(inFileName)                           -- input object
    source = DataInputStream(FileInputStream(infile))

    say "-------------------------------------------------------------------------"
    say "HexPrint    Version 0.50"
    say "-------------------------------------------------------------------------"
    say "File name:" inFilename
    say "File Date:" Date(inFile.lastModified)          --   yyyy/mm/dd hh:mm:ss
    say "File size:" inFile.length "bytes"
    say "-------------------------------------------------------------------------"
    say "    Offset"
    say "<Hex>    <dec>     +0       +4       +8       +C"
    say "-------------------------------------------------------------------------"

    lineHex   = ''                          -- 16 char line in hexadecimal
    lineChr   = ''                          -- 16 char line as character
    colCount  = 0                           -- count columns to 16
    byteCount = 0                           -- count total bytes

    loop forever
      ch  = Rexx source.readUnsignedByte()      -- read one byte
```

Figure 101 (Part 1 of 2). Byte-Oriented Input/Output: HexPrint.nrx

```
        if colCount = 16 then do
           prtline(lineHex,lineChr,byteCount)     -- print 1 line per 16 characters
           lineHex  = ''
           lineChr  = ''
           colCount = 0
        end

        lineHex   = lineHex || ch.d2x(2)          -- append to hexadecimal format
        lineChr   = lineChr || ch.d2c()           -- append to character format
        colCount  = colCount  + 1                 -- count columns to 16
        byteCount = byteCount + 1                 -- count total bytes

      catch IOException                           -- end of file
        if lineHex <> '' then do                  -- are there unprinted characters
          prtline(lineHex,lineChr,byteCount)
        end
      end

      source.close()                              -- close input
      return

      /* ---------------------------------------- print one line -------------- */
      method prtline(lineHex,lineChr,byteCount) static
        bCount   = (bytecount-1) % 16 * 16        -- calculate offset of 1st byte
        hexCount = bcount.d2x.right(6,'0')        -- hexadecimal offset
        decCount = bcount.right(8,'0')            -- decimal offset
        hexL = lineHex.left(32,' ')
        hStr = hexL.substr(1,8) hexL.substr(9,8)- -- build hex print
               hexL.subStr(17,8) hexL.subStr(25,8)
        cStr = lineChr.translate('', -            -- build character print
                       '\0'.sequence('\x1F'),'.') -- code page dependent ??
        say hexCount '('decCount') ' hStr.left(36) '['cStr']'
        return
    -- end HexPrint
```

Figure 101 (Part 2 of 2). Byte-Oriented Input/Output: HexPrint.nrx

Here is a sample output listing of this program.

```
d:\NrxRedBk\file>java HexPrint test.dat
------------------------------------------------------------------------------
HexPrint    Version 0.50
------------------------------------------------------------------------------
File name: test.dat
File Date: Wed May 21 13:36:48 PDT 1997
File size: 78 bytes
------------------------------------------------------------------------------
    Offset
<Hex>    <dec>     +0       +4       +8       +C
------------------------------------------------------------------------------
000000 (00000000)  54686973 20697320 61206669 72737420  [This is a first ]
000010 (00000016)  6C696E65 206F6620 74657374 2E646174  [line of test.dat]
000020 (00000032)  0D0A4865 72652073 6F6D6520 6865783A  [..Here some hex:]
000030 (00000048)  20010203 0D0A5468 69732069 73206120  [ .....This is a ]
000040 (00000064)  7365636F 6E642046 696E6521 0D0A      [second line!..]
```

Data-Oriented I/O

You often need to read and write variables of the basic types, such as integer and floating point. Java provides data-oriented classes, DataInputStream and DataOutputStream for this purpose. Typical methods for data-oriented input and output are:

writeInt Write an integer value

writeFloat Write a floating point value

writeUTF Write character data in Unicode

readInt Read an integer value

readFloat Read a floating point value

readUTF Read character data in Unicode

The data format is platform independent and compact. The disadvantage is that these files are not readable by the human eye.

Our sample programs show how objects containing integer, floating point, string, and Rexx data can be written and read using data-oriented classes and methods. The first example handles basic types, using a data stream, the second example uses Rexx strings.

Data-Oriented I/O Using Data Streams

The data stream classes support the methods for reading and writing the basic types, for example, a Boolean value, character, numbers of varying types, and string.

The methods for reading and writing the basic types are paired, for example, writeInt and readInt, so that you can easily read back the values written previously.

Strings are usually of varying length, and you have to consider the use of delimiters. The writeUTF method adds the length of the string before the data, and readUTF can read back the string very easily. If you use the writeBytes method, you have to add a delimiter character. One solution is to write a new line (\n) after each string, so you can read back the string, using the readLine method. However, the readLine method of the DataInputStream class is “deprecated” in JDK 1.1, that is, its use is no longer suggested. Therefore, we recommend using the writeUTF/readUTF pair for strings.

One example of the platform-independent design of Java is that writeInt always uses the big-endian format. In contrast, C and C++ use the little-endian format for Pentium machines and the big-endian format for Sun Sparc.

Figure 102 shows the use of data-oriented streams to save and retrieve object attributes.

```
/* file\DataIO.nrx

   Output of a Customer object with binary data using DataOutputStream */

class DataIO
  Properties constant
    yes = boolean 1
    no  = boolean 0
```

Figure 102 (Part 1 of 3). Data-Oriented I/O Using Data Streams: DataIO.nrx

```
  method main(args=String[]) static
    custDB = Customer[4]                          -- allocate 4 customers

    -- instanciate objects
    custDB[0] = Customer(101,"Ueli Wahli"         ,"U.S.A." ,500.5,25,yes)
    custDB[1] = Customer(102,"Peter Heuchert"     ,"Germany",400.4,30,yes)
    custDB[2] = Customer(103,"Frederik Haesbrouck","Belgium",350.9,24,no)
    custDB[3] = Customer(104,"Norio Furukawa"     ,"Japan"  ,250.5,39,no)

    -- writes the object variables to a file
    os = DataOutputStream(FileOutputStream("dataio.dat"))
    os.writeInt(custDB.length)                    -- number of objects
    loop i = 0 to custDB.length-1
       os.writeUTF(custDB[i].getCustNo())         -- write object data
       os.writeUTF(custDB[i].getName())
       os.writeUTF(custDB[i].getAddress())
       os.writeFloat(custDB[i].getHourly())
       os.writeInt(custDB[i].getWork())
       os.writeBoolean(custDB[i].getBool())
    end
    os.close()

    -- reads the object variables from the file
    is = DataInputStream(FileInputStream("dataio.dat"))
    loop i = 1 to is.readInt()                    -- read the objects
      xcustno  = Rexx is.readUTF()
      xname    = Rexx is.readUTF()
      xaddress = Rexx is.readUTF()
      xhourly  = is.readFloat()
      xwork    = is.readInt()
      xbool    = is.readBoolean()
      say xcustno.left(4) xname.left(20) xaddress.left(10) -
          (xhourly*xwork).right(10) xbool
    end
    is.close()

/* ---------------------------------------------------------------------- */
/* Customer class                                                         */
/* ---------------------------------------------------------------------- */
class Customer

  properties private                              -- various data types
    custNo  = String
    name    = String
    address = Rexx
    hourly  = float
    work    = int
    bool    = boolean

  method Customer(aCustNo=String, aName=String, aAddress=rexx, -
                  aHourly=float, aWork=int, aBool=boolean)
    custNo = aCustNo; name = aName; address = aAddress
    hourly = aHourly; work = aWork; bool = aBool

  method getCustNo() returns String
    return custNo
  method getName() returns String
```

Figure 102 (Part 2 of 3). Data-Oriented I/O Using Data Streams: DataIO.nrx

```
    return name
  method getAddress() returns Rexx
    return address
  method getHourly() returns float
    return hourly
  method getWork() returns int
    return work
  method getBool() returns boolean
    return bool
-- end
```

Figure 102 (Part 3 of 3). Data-Oriented I/O Using Data Streams: DataIO.nrx

To verify what was written in the output file, we print its contents with the HexPrint program:

```
d:\NrxRedBk\file>java HexPrint DataIO.dat
------------------------------------------------------------------------------
HexPrint    Version 0.50
------------------------------------------------------------------------------
File name: dataio.dat
File Date: Wed May 21 12:04:38 PDT 1997
File size: 158 bytes
------------------------------------------------------------------------------
    Offset
<Hex>    <dec>     +0       +4       +8       +C
------------------------------------------------------------------------------
000000 (00000000)  00000004 00033130 31000A55 656C6920  [......101..Ueli ]
000010 (00000016)  5761686C 69000655 2E532E41 2E43FA40  [Wahli..U.S.A.Cf@]
000020 (00000032)  00000000 19010003 31303200 0E506574  [........102..Pet]
000030 (00000048)  65722048 65756368 65727400 07476572  [er Heuchert..Ger]
000040 (00000064)  6D616E79 43C83333 0000001E 01000331  [manyC+33.......1]
000050 (00000080)  30330013 46726564 6572696B 20486165  [03..Frederik Hae]
000060 (00000096)  7362726F 75636B00 0742656C 6769756D  [sbrouck..Belgium]
000070 (00000112)  43AF7333 00000018 00000331 3034000E  [C¯s3.......104..]
000080 (00000128)  4E6F7269 6F204675 72756B61 77610005  [Norio Furukawa..]
000090 (00000144)  4A617061 6E437A80 00000000 2700      [JapanCz....'.]
```

You can match the file contents easily with the data-oriented methods:

```
offset length value                                 method
------------------------------------------------------------------------------
000000        00000004                              writeInt(custDB.length)
000004   0003 313031                                writeUTF("101")
00000A   000A 55656C69 20576168 6C69                writeUTF("Ueli Wahli")
000015   0006 552E532E 412E                         writeUTF("U.S.A.)
00001D        43FA4000                              writeFloat(500.5)
000021        00000019                              writeInt(25)
000025        01                                    writeBoolean(1)
------------------------------------------------------------------------------
```

Data-Oriented I/O Using Rexx Strings

The Rexx class provided with NetRexx can handle many data types, including integer, floating point, and strings.

Instead of the methods of the DataOutputStream, you can use the Rexx class to write the object values to a file and retrieve them safely again. The sample program shown in Figure 103 uses Rexx strings to save and retrieve object attributes. Using Rexx strings, you can read and write very easily and do not have handle each data type separately. As a delimiter between the fields, we use the tabulator (\t) character.

```
/* file\DataIO2.nrx

   Output of a Customer object with numeric data using Rexx strings */

import Customer                                  -- from "DataIO.nrx"

class DataIO2
  Properties constant
    yes = boolean 1
    no  = boolean 0

  method main(args=String[]) static
    custDB = Customer[4]                         -- allocate 4 customers


    -- Instanciate objects
    custDB[0] = Customer(101,"Ueli Wahli"         ,"U.S.A." ,500.5,25,yes)
    custDB[1] = Customer(102,"Peter Heuchert"     ,"Germany",400.4,30,yes)
    custDB[2] = Customer(103,"Frederik Haesbrouck","Belgium",350.9,24,no)
    custDB[3] = Customer(104,"Norio Furukawa"     ,"Japan"  ,250.5,39,no)

    -- writes the object variables to a file
    os = PrintWriter(FileWriter("dataio2.dat"))
    os.println(custDB.length)                    -- number of objects
    loop i = 0 to custDB.length-1
       custdata = custDB[i].getCustNo()  || '\t' || custDB[i].getName()   || '\t' -
                  custDB[i].getAddress() || '\t' || custDB[i].getHourly() || '\t' -
                  custDB[i].getWork()    || '\t' || custDB[i].getBool()
       os.println(custdata)
    end
    os.close()

    -- reads the object variables from the file
    is = BufferedReader(FileReader("dataio2.dat"))
    n=is.readLine()                              -- read the objects
    loop i = 1 to n
      parse is.readLine() xcustno '\t' xname '\t' xaddress '\t' xhourly -
                                  '\t' xwork '\t' xbool
      say xcustno.left(4) xname.left(20) xaddress.left(10) -
          (xhourly*xwork).right(10) xbool
    end
    is.close()
-- end
```

Figure 103. Data-Oriented I/O Using Rexx Strings: DataIO2.nrx

Because we used Rexx strings to save the values, we can simply type the contents of the output file:

```
d:\NrxRedBk\file>type dataio2.dat
4
101     Ueli Wahli       U.S.A.  500.5    25     1
102     Peter Heuchert   Germany         400.4     30     1
103     Frederik Haesbrouck       Belgium        350.9     24     0
104     Norio Furukawa   Japan   250.5    39     0
```

Alternatively we can print the contents of the sequential file, using the HexPrint program:

```
------------------------------------------------------------------------------
HexPrint    Version 0.50
------------------------------------------------------------------------------
File name: dataio2.dat
File Date: Wed May 21 12:33:58 PDT 1997
File size: 165 bytes
```

```
-------------------------------------------------------------------------
   Offset
<Hex>    <dec>     +0       +4       +8       +C
-------------------------------------------------------------------------
000000 (00000000)  340D0A31 30310955 656C6920 5761686C  [4..101.Ueli Wahl]
000010 (00000016)  69092055 2E532E41 2E093530 302E3509  [i. U.S.A..500.5.]
000020 (00000032)  20323509 310D0A31 30320950 65746572  [ 25.1..102.Peter]
000030 (00000048)  20486575 63686572 74092047 65726D61  [ Heuchert. Germa]
000040 (00000064)  6E790934 30302E34 09203330 09310D0A  [ny.400.4. 30.1..]
000050 (00000080)  31303309 46726564 6572696B 20486165  [103.Frederik Hae]
000060 (00000096)  7362726F 75636B09 2042656C 6769756D  [sbrouck. Belgium]
000070 (00000112)  09333530 2E390920 32340930 0D0A3130  [.350.9. 24.0..10]
000080 (00000128)  34094E6F 72696F20 46757275 6B617761  [4.Norio Furukawa]
000090 (00000144)  09204A61 70616E09 3235302E 35092033  [. Japan.250.5. 3]
0000A0 (00000160)  3909300D 0A                          [9.0..]
```

Object-Oriented I/O Using Serialization

Serialization, a new feature of JDK 1.1, enables you to write out an object with just one call to the *writeObject* method and read it back in using the *readObject* method.

To serialize objects you write them to an ObjectOutputStream, and to read them back you use an ObjectInputStream.

To add the support for serialization to a class, you have to implement the java.io.Serializable interface. If you do not, the NotSerializableException will be signaled.

```
class Customer2 implements Serializable
```

If you use only the supported basic data types, you do not have to add extra code to the class. For classes with special data types, you must implement the writeObject and readObject methods yourself.

In NetRexx 1.0, the Rexx string class is a nonserializable class. Therefore, you could not use writeObject for the Customer class used in the previous examples. NetRexx 1.1 makes the Rexx class serializable.

For our sample program (see Figure 104) we implemented a Customer2 class that uses only basic data types.

```
/* file\SeriaIO.nrx

   Output of a Customer object with binary data using Serialization */

class SeriaIO
  Properties constant
    yes = boolean 1
    no  = boolean 0

  method main(args=String[]) static
    custDB = Customer2[4]                        -- allocate 4 customers
    custRD = Customer2[]                         -- read back "x" customers

    -- instanciate objects
    custDB[0] = Customer2(101,"Ueli Wahli"         ,"U.S.A." ,500.5,25,yes)
    custDB[1] = Customer2(102,"Peter Heuchert"     ,"Germany",400.4,30,yes)
```

Figure 104 (Part 1 of 3). Object-Oriented I/O Using Serialization: SeriaIO.nrx

```
    custDB[2] = Customer2(103,"Frederik Haesbrouck","Belgium",350.9,24,no)
    custDB[3] = Customer2(104,"Norio Furukawa"     ,"Japan"  ,250.5,39,no)

    -- writes the object variables to a file
    say 'Writing' custDB.length 'customers'
    os = ObjectOutputStream(FileOutputStream("seriaio.dat"))
    os.writeInt(custDB.length)                  -- number of objects

    os.writeObject(custDB)                      -- WRITE OBJECTS WITH ONE CALL

    os.flush()                                  -- force output
    os.close()

    -- reads the object variables from the file
    say 'Reading...'
    is = ObjectInputStream(FileInputStream("seriaio.dat"))
    n  = is.readInt()                           -- number of customers
    say 'Display of' n 'customers:'

    custRD = Customer2[] is.readObject()        -- READ OBJECTS WITH ONE CALL

    loop i = 0 to custRD.length-1
      say custRD[i].getCustNo() (Rexx custRD[i].getName()).left(20) -
          (Rexx custRD[i].getAddress()).left(10) -
          (Rexx custRD[i].getHourly() * custRD[i].getWork()).right(10) -
          custRD[i].getBool()
    end
    is.close()

/* ---------------------------------------------------------------------- */
/* Customer class                                                         */
/* ---------------------------------------------------------------------- */
class Customer2 implements Serializable

  properties private                            -- various data types
    custNo  = String
    name    = String
    address = String                            -- Rexx not allowed
    hourly  = float
    work    = int
    bool    = boolean

  method Customer2(aCustNo=String, aName=String, aAddress=rexx, -
                   aHourly=float, aWork=int, aBool=boolean)
    custNo = aCustNo; name = aName; address = aAddress
    hourly = aHourly; work = aWork; bool = aBool

  method getCustNo() returns String
    return custNo
  method getName() returns String
    return name
  method getAddress() returns Rexx
    return address
  method getHourly() returns float
    return hourly
  method getWork() returns int
    return work
  method getBool() returns boolean
```

Figure 104 (Part 2 of 3). Object-Oriented I/O Using Serialization: SerialO.nrx

```
    return bool
-- end
```

Figure 104 (Part 3 of 3). Object-Oriented I/O Using Serialization: SerialO.nrx

When we run the program, the entire array of four customers is written to the output file using a single call of the writeObject method. Afterward, the array is read back using a single call of the readObject method, and the customers are displayed:

```
d:\NrxRedBk\file>java SeriaI0
Writing 4 customers
Reading...
Display of 4 customers:
101 Ueli Wahli           U.S.A.        12512.5 1
102 Peter Heuchert       Germany       12012.0 1
103 Frederik Haesbrouck  Belgium        8421.6 0
104 Norio Furukawa       Japan          9769.5 0
```

We can print the content of the sequential file, using the HexPrint program. In the output you can see the class-related information before the actual data:

```
-------------------------------------------------------------------------
HexPrint    Version 0.50
-------------------------------------------------------------------------
File name: seriaio.dat
File Date: Wed May 21 13:33:46 PDT 1997
File size: 365 bytes
-------------------------------------------------------------------------
    Offset
<Hex>    <dec>     +0       +4       +8       +C
-------------------------------------------------------------------------
000000 (00000000)  ACED0005 77040000 00047572 000C5B4C  [Ñõ..w.....ur..·L]
000010 (00000016)  43757374 6F6D6572 323BD58F B90559BB  [Customer2;ÿ§.Y¼]
000020 (00000032)  3DF00200 00787000 00000473 72000943  [=Ü...xp....sr..C]
000030 (00000048)  7573746F 6D657232 B73FA0BA CC66DC79  [ustomer2fi?·]fÄy]
000040 (00000064)  0200065A 0004626F 6F6C4600 06686F75  [...Z..boolF..hou]
000050 (00000080)  726C7949 0004776F 726B4C00 07616464  [rlyI..workL..add]
000060 (00000096)  72657373 7400124C 6A617661 2F6C616E  [resst..Ljava/lan]
000070 (00000112)  672F5374 72696E67 3B4C0006 63757374  [g/String;L..cust]
000080 (00000128)  4E6F7400 124C6A61 76612F6C 616E672F  [Not..Ljava/lang/]
000090 (00000144)  53747269 6E673B4C 00046E61 6D657400  [String;L..namet.]
0000A0 (00000160)  124C6A61 76612F6C 616E672F 53747269  [.Ljava/lang/Stri]
0000B0 (00000176)  6E673B78 700143FA 40000000 00197400  [ng;xp.CÙ@.....t.]
0000C0 (00000192)  06552E53 2E412E74 00033130 3174000A  [.U.S.A.t..101t..]
0000D0 (00000208)  55656C69 20576168 6C697371 007E0002  [Ueli Wahlisq...]
0000E0 (00000224)  0143C833 33000000 1E740007 4765726D  [.C·33....t..Germ]
0000F0 (00000240)  616E7974 00033130 3274000E 50657465  [anyt..102t..Pete]
000100 (00000256)  72204865 75636865 72747371 007E0002  [r Heuchertsq...]
000110 (00000272)  0043AF73 33000000 18740007 42656C67  [.C¯s3....t..Belg]
000120 (00000288)  69756D74 00033130 33740013 46726564  [iumt..103t..Fred]
000130 (00000304)  6572696B 20486165 7362726F 75636B73  [erik Haesbroucks]
000040 (00000320)  71007E00 0200437A 80000000 00277400  [q....Czä....'t.]
000150 (00000336)  054A6170 616E7400 03313034 74000E4E  [.Japant..104t..N]
000160 (00000352)  6F72696F 20467572 756B6177 61        [orio Furukawa]
```

Handling an End-of-File Condition

There are two ways of handling the end-of-file (EOF) condition. You can check the return value or catch the exception.

The method you choose for handling the EOF condition in a specific program depends mainly on the stream class and method that is used to read the file.

Check the Return Value

Return value checking can be used in the following methods:

- readLine of the BufferedReader class, null is returned on EOF
- read of the BufferedReader class, -1 is returned on EOF
- readLine of the DataInputStream class, null is returned on EOF

Here is sample code to test the EOF condition:

```
/* file\Eof1.nrx */
source = BufferedReader(FileReader("test.dat"))
loop forever
   textline = source.readLine()
   if textline = null then leave       -- <=== leave loop on EOF
   say textline
end
```

Catch the I/O Exception

You can intercept the EOF condition, using Java's exception handling.

For example, almost all read methods of the DataInputStream class, including readLine, throw an exception if the returning null pointer is assigned to a Rexx string:

```
/* file\Eof2.nrx */
source = DataInputStream(FileInputStream("test.dat"))
loop forever
   textLine = Rexx source.readLine()
   say textLine
 catch NullPointerException            -- <=== leave the loop on EOF
end
```

The sample HexPrint program in Figure 101 on page 170 uses an I/O exception to catch the EOF condition and write out the last line of data:

```
/* file\Eof3.nrx */
source = DataInputStream(...)
loop forever
   ch = Rexx source.readUnsignedByte()
   ...
 catch IOException
   ... handle EOF
end
```

Chapter 10. Database Connectivity with JDBC

In this chapter we discuss the connectivity features of NetRexx with respect to relational databases, using the Java Database Connectivity Application Programming Interface (JDBC API).

We limit ourselves to a discussion focussed on the interaction of Java applications with IBM Database 2 products.

Note: We used DB2 Universal Database (UDB) beta code for the tests. At the time of writing this book, DB2 2.1.2 on OS/2 and Windows did not support JDK 1.1.

JDBC and ODBC

Rexx—the predecessor of NetRexx—is famous for its easy-to-use facilities for accessing DB2 from within scripts using a simple syntax. If we want NetRexx to be used as a decent inheritant of Rexx, we should at least discuss what its possibilities and advantages (over Rexx) are.

The Open DataBase Connectivity (ODBC) API is quite similar to the JDBC API.[2] Users of ODBC will find JDBC very easy to learn, and with the basic information presented here, they can go ahead and write applications. More detailed information is available in the JDK documentation.

Fortunately there is also a generic JDBC implementation that converts JDBC database requests to ODBC—the so-called *JDBC-ODBC Bridge*—so that every database engine that supports ODBC can be accessed.

Thus, instead of seeing this chapter as a complete reference on the JDBC API, look at it as a short introduction for NetRexx and Rexx addicts.

JDBC Concepts

JDBC is an API set that specifies how you should interface with any (relational) database from within your Java—and by consequence also from within your NetRexx—programs. As mentioned before, the purpose is to have one common way of accessing data in different types of database engines.

The various firms producing the database engines are then responsible for providing a way of converting JDBC requests to queries in their own terminology and giving back the result in conformance with the JDBC protocols. This database-specific conversion is accomplished through a *JDBC driver*.

The current versions of JDBC drivers can obtained from the Web site shown in Figure 105.

[2] ODBC is Microsoft's attempt to put an end to the different proprietary APIs of the multiple companies selling database access engines. Today ODBC is widely used on personal computer platforms.

```
http://www.javasoft.com/products/jdbc
```

Figure 105. Source of Latest JDBC Drivers

This Web site should also link you to the Web pages of the various database engine firms to download the latest version of their JDBC drivers; guide you to discussion groups on this subject; and—most important—provide you with the latest information on the rapidly evolving subject of JDBC.

In practice, before you can connect to a relational database, you must:

1. Find the URL of the database
2. Ask the JDBC DriverManager, a Java class, to provide you with the appropriate JDBC driver for the database described in the URL

If you succeed, you have an object that implements the *Connection* interface. You can compare this instance (object) that implements the connection to the notion of a "session" in a classic database context.

To execute a database query, you use an instance of an object that implements the *Statement* interface, and you supply the SQL query as a string.

Executing a statement returns a *ResultSet*, or, more correctly, an object that implements the ResultSet interface. You iterate through such a result set as with a normal database cursor. Other statement objects provide the function of SQL update, delete, and insert.

This is basically the whole idea behind the use of JDBC to access your databases. Every JDBC access matches these general rules; the different database engines can be manipulated by using these simple implementations of Connection, Statement, and ResultSet.

As you probably noticed, these basic concepts are defined as interfaces. In every step we ask a concrete implementation of an interface to get an object that complies to another interface:

- We ask the DriverManager to create a Connection to a certain URL
- We ask a Connection to create a Statement
- We ask a Statement to create a ResultSet (by executing the SQL query)

These interfaces, together with the methods that generate the real objects, make the JDBC concepts applicable to various database implementations. This mode of operation is an example of the *Abstract Factory* design pattern (see *Design Patterns: Elements of Reusable Object-Oriented Software*). This pattern is also used in the *java.net* package and is discussed in Chapter 11, "Network Programming" on page 205.

Now let's go on to some of the details of the JDBC concepts.

Database URLs

URLs are extended—from their most known form to describe where to look for Web pages—to be used for locating database resources on the Internet. The standard syntax for JDBC is:

```
jdbc:<subprotocol>:<subname>
```

Subprotocol is usually the name of the database engine or a network alias, and *subname* locates the data resource in more detail. The subname field can take the form of a normal URL without the protocol part. Here is an example of a JDBC URL:

```
jdbc:db2:/chusa:8888/sample
```

A URL for accessing a database through JDBC can be far more complex but should be recognizable by its protocol part (always *jdbc*) and its colons (:). The makers of the drivers have in fact great flexibility in defining their own URL format to encapsulate some parameters in the URL itself.

The good part of all of this is that database user (the programmers of the Java code that accesses a certain database) only have to copy the URL they get from the database administrator. The URLs offer flexibility by providing some way of indirection through network name services, for example, dynamic name services (DNS).

In the case of DB2 database engines, the URL always starts with the jdbc:db2: sequence. For local databases you append the database name to it:

```
jdbc:db2:sample
```

Remote DB2 databases are located by inserting the host name and port number of a JDBC server daemon before the database name:

```
jdbc:db2://chusa:8888/sample
```

Note: This looks almost like a normal URL to locate a file.

JDBC Drivers

JDBC drivers are Java classes that are specific to a certain database engine. They all enable you—in some way or another—to access your data, locally or remotely, using ODBC, native, or generic implementations.

On the JDBC drivers Web page (Figure 105 on page 182) you will see different categories of JDBC drivers. These categories are defined to distinguish four major concepts for creating a JDBC driver.

In short we suggest that you look for a driver in one of the highest categories, say, category 3 or 4. These categories enable you to support any Java client without having to install some software on the client platform, so JDBC drivers in these categories are closer to the basic Java concept of portability.

Every database engine that supports an ODBC client is by definition JDBC category 1 compliant, because it can be used in combination with the JDBC-ODBC Bridge. Note, however, that this solution is far from ideal with respect to performance and portability because ODBC and client enabler software (platform dependent!) have to be installed on every client.

JDBC-Net is a middleware protocol that is used in category 3 drivers. This database management system (DBMS) independent protocol transfers queries from the client Java application to a server daemon, which in turn translates the queries and executes them on the database. This three-tier model only needs a light, generic driver on the client; all database-dependent drivers are located on the platform running the server deamon.

DB2 products are now delivered with two drivers, category 2 and 3. For category 2 you have to install the DB2 Client Application Enabler (CAE), a native DB2 driver for clients, and for category 3 you run a server daemon that transforms the generic JDBC-Net protocol into queries that are then dispatched to the appropriate DB2 server.

The DB2 JDBC drivers are denoted—in their full classname—by:

```
com.ibm.db2.jdbc.app.DB2Driver
com.ibm.db2.jdbc.net.DB2Driver
```

The second driver handles database URLs that reference a remote host.

JDBC Daemon

The daemon that enables you to connect to a database from a remote client is started by:

```
db2jstrt <portnumber>
```

The server daemon has to run on a machine that either hosts the specific database or has the software to connect to a DB2 server, such as CAE or Distributed Database Connectivity Services (DDCS).

The *portnumber* is the number of the socket to which the server is listening. See Chapter 11, “Network Programming” on page 205 for more information about sockets.

JDBC Driver Installation

DB2 provides a zip file of all DB2 JDBC classes and this zip file must be accessible in the CLASSPATH environment variable:

```
SET CLASSPATH=.....;d:\SQLLIB\JAVA\db2java.zip;.....
```

JDBC Compliance

JDBC-compliant drivers must support the SQL language up to the ANSI SQL-92 Entry Level standard. This guarantees that you can write programs using JDBC that can run on every Java platform, using a DBMS-provided JDBC driver. The fact that a driver is JDBC compliant does not limit you in using database engines that have more capabilities. Because the JDBC client code passes the SQL statement to the driver exactly as you wrote it in your program, you can use every feature of the database server. When using such advanced features, keep the following points in mind:

- Your code will not be as portable
- If the actual database engine does not like your SQL statement, you might have to catch (probably strange) exceptions.

In addition to DBMS-specific features, the JDBC API contains some database interactions that are beyond the ANSI standard. These are mostly accompanied by *getter* methods that enable you to check within your code whether such features are available on the current Connection.

Enough theory now; let’s start with some practical examples.

SQL Select in Practice

We base our example on the Sample database that is supplied with every DB2 product.

DB2 Sample Database

To create the sample database, execute:

```
db2sampl.exe
```

You can find this program in the following directory:

```
\sqllib\bin    :  OS/2, Windows 95/NT
/sqllib/misc/  :  Unix
```

This program creates a database with nine tables on the account of the current user. In the interest of the examples that follow, we suggest that you log on as the default user, USERID. If not, remember the user ID that you use to create the sample database, and supply it in subsequent examples as the *prefix* parameter. (Note that the above procedure can be slightly different on your platform; consult the DB2 documentation if in doubt.)

From the sample database we use the *employee* and *department* tables. Note that we only access a subset of the columns to keep the examples easy to understand.

The employee table has following SQL definition:

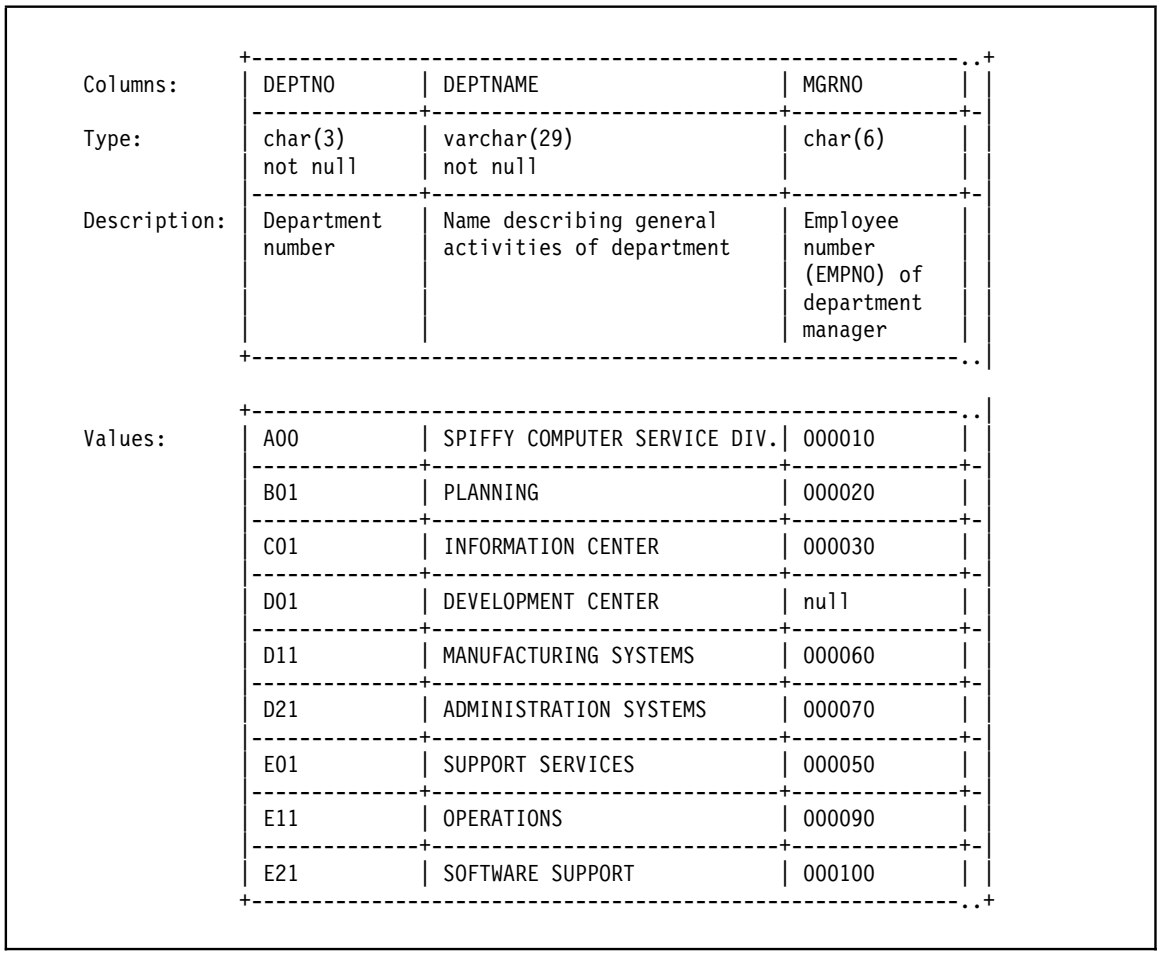
```
CREATE TABLE EMPLOYEE
           (EMPNO      CHAR(6) NOT NULL,
            FIRSTNME   VARCHAR(12),
            LASTNAME   VARCHAR(15),
            WORKDEPT   CHAR(3),
            JOB        CHAR(8),
            ...
            SALARY     DECIMAL(9,2),
            PRIMARY KEY (EMPNO),
           )
```

Figure 106 shows an extract of the data in the employee table.

EMPNO	FIRSTNME	LASTNAME	WORK DEPT	JOB		SALARY	
000010	CHRISTINE	HAAS	A00	PRES		52750	
000020	MICHAEL	THOMPSON	B01	MANAGER		41250	
000030	SALLY	KWAN	C01	MANAGER		38250	
000050	JOHN	GEYER	E01	MANAGER		40175	
000060	IRVING	STERN	D11	MANAGER		32250	
000070	EVA	PULASKI	D21	MANAGER		36170	
000090	EILEEN	HENDERSON	E11	MANAGER		29750	
000100	THEODORE	SPENSER	E21	MANAGER		26150	
000110	VINCENZO	LUCCHESSI	A00	SALESREP		46500	
000120	SEAN	O'CONNELL	A00	CLERK		29250	
000130	DOLORES	QUINTANA	C01	ANALYST		23800	
000140	HEATHER	NICHOLLS	C01	ANALYST		28420	
000340	JASON	GOUNOT	E21	FIELDREP		23840	

Figure 106. Employee Table Sample Data

The second table that we use is the department table. Figure 107 shows the definition of the table and some sample data.



Columns:	DEPTNO	DEPTNAME	MGRNO
Type:	char(3) not null	varchar(29) not null	char(6)
Description:	Department number	Name describing general activities of department	Employee number (EMPNO) of department manager

Values:	A00	SPIFFY COMPUTER SERVICE DIV.	000010
	B01	PLANNING	000020
	C01	INFORMATION CENTER	000030
	D01	DEVELOPMENT CENTER	null
	D11	MANUFACTURING SYSTEMS	000060
	D21	ADMINISTRATION SYSTEMS	000070
	E01	SUPPORT SERVICES	000050
	E11	OPERATIONS	000090
	E21	SOFTWARE SUPPORT	000100

Figure 107. Department Table Layout and Sample Data

Please refer to the reference material supplied with the DB2 products if you have questions regarding the use of SQL, DB2, or the sample database.

Select Query Example

Our first query consists of an inner join of the department and employee tables. It collects all departments that have a manager and prints out the last and first names of the managers, sorted by department number:

```
SELECT deptno, deptname, lastname, firstnme
  FROM userid.department dep, userid.employee emp
 WHERE dep.mgrno = emp.empno ORDER BY dep.deptno
```

Figure 108 shows the NetRexx program that executes this query and prints out the results.

```
/* jdbc\JdbcQry.nrx

   This NetRexx program demonstrate DB2 query using the JDBC API.
   Usage: Java JdbcQry [<DB-URL>] [<userprefix>] */
```

Figure 108 (Part 1 of 3). JDBC NetRexx Query Program: JdbcQry.nrx

```
import java.sql.

parse arg url prefix                    -- process arguments
if url = '' then
   url = 'jdbc:db2:sample'
else do                                 -- check for correct URL
     parse url p1 ':' p2 ':' rest
     if p1 \= 'jdbc' | p2 \= 'db2' | rest = '' then do
        say 'Usage: java JdbcQry [<DB-URL>] [<userprefix>]'
        exit 8
     end
end
if prefix = '' then prefix = 'userid'

do                                      -- loading DB2 support
  say 'Loading DB2 driver classes...'
  Class.forName('COM.ibm.db2.jdbc.app.DB2Driver').newInstance()
  -- Class.forName('COM.ibm.db2.jdbc.net.DB2Driver').newInstance()
catch e1 = Exception
  say 'The DB2 driver classes could not be found and loaded !'
  say 'Exception (' e1 ') caught : \n' e1.getMessage()
  exit 1
end                                     -- end : loading DB2 support

do                                      -- connecting to DB2 host
  say 'Connecting to:' url
  jdbcCon = Connection DriverManager.getConnection(url, 'userid', 'password')
catch e2 = SQLException
  say 'SQLException(s) caught while connecting !'
  loop while (e2 \= null)
     say 'SQLState:' e2.getSQLState()
     say 'Message: ' e2.getMessage()
     say 'Vendor:  ' e2.getErrorCode()
     say
     e2 = e2.getNextException()
   end
   exit 1
end                                     -- end : connecting to DB2 host

do                                      -- get list of departments with the managers
  say 'Creating query...'
  query = 'SELECT deptno, deptname, lastname, firstnme' -
           'FROM' prefix'.DEPARTMENT dep,' prefix'.EMPLOYEE emp'-
           'WHERE dep.mgrno=emp.empno ORDER BY dep.deptno'
  stmt = Statement jdbcCon.createStatement()
  say 'Executing query:'
  loop i=0 to (query.length()-1)%75
     say '   ' query.substr(i*75+1,75)
  end
  rs = ResultSet stmt.executeQuery(query)
  say 'Results:'
  loop row=0 while rs.next()
     say rs.getString('deptno') rs.getString('deptname') -
     'is directed by' rs.getString('lastname') rs.getString('firstnme')
  end
  rs.close()                            -- close the ResultSet
  stmt.close()                          -- close the Statement
```

Figure 108 (Part 2 of 3). JDBC NetRexx Query Program: JdbcQry.nrx

```
    jdbcCon.close()                     -- close the Connection
    say 'Retrieved' row 'departments.'
  catch e3 = SQLException
    say 'SQLException(s) caught !'
    loop while (e3 \= null)
       say 'SQLState:' e3.getSQLState()
       say 'Message: ' e3.getMessage()
       say 'Vendor:  ' e3.getErrorCode()
       say
       e3 = e3.getNextException()
     end
  end                                   -- end: get list of departments
```

Figure 108 (Part 3 of 3). JDBC NetRexx Query Program: JdbcQry.nrx

We compile and run this program; the results are shown in Figure 109.

```
d:\NrxRedBk\jdbc>java JdbcQry jdbc:db2:sample userid
Loading DB2 driver classes...
Connecting to: jdbc:db2:sample
Creating query...
Executing query:
    SELECT deptno, deptname, lastname, firstnme FROM userid.DEPARTMENT dep,
    userid.EMPLOYEE emp WHERE dep.mgrno=emp.empno ORDER BY dep.deptno
Results:
A00 SPIFFY COMPUTER SERVICE DIV. is directed by HAAS       CHRISTINE
B01 PLANNING                     is directed by THOMPSON   MICHAEL
C01 INFORMATION CENTER           is directed by KWAN       SALLY
D11 MANUFACTURING SYSTEMS        is directed by STERN      IRVING
D21 ADMINISTRATION SYSTEMS       is directed by PULASKI    EVA
E01 SUPPORT SERVICES             is directed by GEYER      JOHN
E11 OPERATIONS                   is directed by HENDERSON  EILEEN
E21 SOFTWARE SUPPORT             is directed by SPENSER    THEODORE
Retrieved 8 departments.
```

Figure 109. JDBC NetRexx Query Results

This is—fortunately—exactly the same list that we would have found manually by searching for all managers in the list of departments.

Query Sample Explanation

We begin our program with the import statement. All JDBC classes and interfaces, except for the JDBC drivers, can be found in the java.sql package.

First we check the arguments. If the first argument does not contain two colons, we do not have a valid URL and we print a message giving the proper usage. Note that this simple check allows us to give a sensible response on a *help* request.

The second parameter contains the name of the owner of the sample database. This is the name you logged on with when creating the sample database (using the DB2SAMPL program) and DB2 used as the prefix for the tables. The prefix defaults to *userid*.

Loading the DB2 Support

Now we begin with the JDBC specific parts. We have to load the JDBC drivers of the DB2 product, so that the DriverManager can "choose" the right one. We use the forName method of the Class class to load the driver classes:

```
Class.forName( db2drivername ).newInstance()
```

The forName method throws a *ClassNotFoundException* if it fails to load the class. Loading a driver class in this way automatically calls the registerDriver method:

```
DriverManager.registerDriver()
```

The registerDriver method adds the driver to the list of drivers in the *sql.drivers* system property that the DriverManager uses to accept the appropriate driver class when connecting to a URL.

Connecting to the DB2 Host

Next we connect to the given URL, using the getConnection method:

```
jdbcCon = Connection DriverManager.getConnection(url,userid,password)
```

The DriverManager steps through the list of drivers by calling each driver's acceptsURL method:

```
Driver.acceptsURL(String)
```

Each JDBC driver class replies if it can connect to the data resource described in the given URL. If none of them is successful, an exception is thrown, and we must be prepared to catch these SQLExceptions.

The JDBC drivers have to implement the *Driver* interface. This interface declares—besides the above-mentioned acceptsURL() method—the *connect* method that the DriverManager calls when the specific driver accepts the given URL. It returns an object that implements the *Connection* interface.

The second and third parameter of the DriverManager.getconnection() method are the user ID and password. We use the default *USERID* and *PASSWORD* combination that normally works for local access. On most systems this default user ID is granted access to the sample database no matter who created it. The user ID and the password are stored in the system properties. When you supply a user ID and password combination that is not valid, the DB2 driver returns the message:

```
SQLState: S1501
Message:  [IBM][CLI Driver] SQL1001N  " " is not a valid
database name.  SQLSTATE=2E000
```

Get the List of Departments

Now we are ready to create an SQL statement:

```
-- method createStatement of the Connection class
stmt = Statement jdbcCon.createStatement()
```

We use the Statement object to execute our SQL query, using this code:

```
-- method executeQuery of the Statement class
rs = ResultSet stmt.executeQuery('SELECT .....')
```

Execution of the SQL query returns a ResultSet that consists of multiple rows that qualify the WHERE clause of the select statement. We loop through the result set, using:

```
-- method next of the ResultSet class
rs.next()
```

Under the covers, a database cursor is created and used to browse the ResultSet. More information about the use of cursors can be found in the JDBC API reference documentations.

Fields of the current (selected) row are retrieved, using these methods:

```
-- method getXxxxx of the ResultSet class
rs.getXxxxx(column)
```

where *Xxxxx* is the data type of the column.

JDBC provides many *get* methods, depending on the data type of the respective SQL column (see Table 10).

Table 10. Get Methods for Table Columns by SQL Data Type

	SQL Type	Java Type
getString	CHAR	String
getString	VARCHAR	String
getString	LONGVARCHAR	String
getAsciiStream	LONGVARCHAR	InputStream
getUnicodeStream	LONGVARCHAR	InputStream
getBigDecimal	NUMERIC	java.math.BigDecimal
getBigDecimal	DECIMAL	java.math.BigDecimal
getBoolean	BIT	boolean
getByte	TINYINT	byte
getShort	SMALLINT	short
getInt	INTEGER	int
getLong	BIGINT	long
getFloat	REAL	float
getDouble	FLOAT	double
getDouble	DOUBLE	double
getDate	DATE	java.sql.Date
getTime	TIME	java.sql.Time
getTimestamp	TIMESTAMP	java.sql.Timestamp
getBinaryStream	LONGVARBINARY	InputStream
getBytes	<every SQL type>	byte array
getObject	<every SQL type>	Object

Table 10 also shows the standard mapping between the SQL and Java types. Some of the get methods can be used for a number of SQL types because they are more generic. For example, *ResultSet.getBytes()* can also be used to receive a CHAR, VARCHAR, LONGVARCHAR, or LONGVARBINARY field. The most generic method, *ResultSet.getObject()*, converts the SQL type to the “nearest” Java class.

The parameter to these methods is either the column number or the column name. Therefore, we have two signatures for all of the get methods:

- getXxxxx(int)
- getXxxxx(String)

Column names are case insensitive and are obviously preferred because changes in the select statement could change the column number.

Ending the Program

Our last job is to end the program properly. We call the close method on the ResultSet, Statement, and Connection. Because database resources can consume a lot of memory, you should release the resource as soon as possible.

NULL Values

After they are retrieved by the respective get method, check columns that allow null values, using:

```
ResultSet.wasNull()
```

The get method will have returned the default initializer of its return type.

Note: Be careful when using a Rexx string as the result of the getString method. In NetRexx Version 1.0, an exception is thrown if a null value is returned; this is fixed in NetRexx Version 1.1. We suggest using a Java String for columns with potential null values.

Meta Data

For advanced queries with varying numbers of returned columns, you can examine the result set, using the getMetaData method:

```
-- method getMetaData of ResultSet
md = ResultSetMetaData rs.getMetaData()
```

The object of the ResultSetMetaData interface class provides a number of methods to analyze the number and types of the returned columns, such as:

getColumnCount()	Number of columns
getColumnName(int)	Name of column
getColumnLabel(int)	Suggested title of column for output
getColumnTypeName(int)	Type of column (as String)
isNullable(int)	Is null allowed for column
getColumnDisplaySize(int)	Required width for output

Consult the JDBC API documentation for more details.

SQL Update in Practice

Database access, and by consequence JDBC, is more than making queries. The SQL insert, update, and delete statements change the content of a database table. Because delete is a very simple statement, and insert and update are quite similar, we only provide an example of an update statement.

In this example we pass program variables—also called host variables— through a prepared SQL statement.

Prepared Statements

The JDBC Statement class has two descendants: *CallableStatement* and *PreparedStatement*. Callable statements are used for calling precompiled SQL statements as a stored procedure (see "Stored Procedures" on page 195).

A prepared statement object represents a statement that can be parameterized. By inserting question marks (?) instead of exact values in the SQL statement, you change the statement into a parameterized statement. The following SQL statement allows us to change the *firstnme* column value in the row that matches the employee number *empno* given as the second parameter:

```
updateQ = 'UPDATE USERID.EMPLOYEE SET firstnme = ? WHERE empno = ?'
```

We prepare the statement, using the prepareStatement method of the Connection object:

```
-- method prepareStatement of Connection
updateStmt = PreparedStatement jdbcCon.prepareStatement(updateQ)
```

The parameters can be assigned real values before executing the statement, using *set* methods that are counterparts of the get methods (Table 10 on page 190).

```
PreparedStatement.setXXXXXX(index, value)
```

The first argument, *index*, is the sequence number of the question mark in the SQL statement, the second one is the value that replaces the question mark. For example, we change the first name of the employee with number 000010:

```
updateStmt.setString(1, 'Christine')
updateStmt.setString(2, '000010')
```

Instead of using a prepared statement, we could also construct a new SQL statement for every update operation by inserting the real values into the statement:

```
UPDATE USERID.EMPLOYEE SET firstnme = 'Christine' WHERE empno = '000010'
```

In this case we have to create a new statement for every SQL update call, instead of using one prepared statement and setting the parameters before each update call. In most cases the prepared statement executes faster, because the underlying database manager can store it in optimized format. The degree of optimization depends on how the database product implements statement preparation.

Executing a Prepared SQL Statement

There are three of executing prepared statements:

Statement.executeQuery() Execute a prepared select statement and return a result set

Statement.executeUpdate() Execute an update, insert, or delete statement and return the number of rows that were updated. This method is also used to execute data definition language (DDL; see "Data Definition Language" on page 195). Use executeUpdate for SQL statements that do not return a result set.

Statement.execute() Excecute a statement that returns multiple result sets. Consult the JDBC documentation for more information.

SQL Update Example

In this NetRexx program (see Figure 110) we change the first names of all employees to start with an uppercase letter and turn the rest of the name into lowercase.

```
/* jdbc\JdbcUpd.nrx

   This NetRexx program demonstrate DB2 update using the JDBC API.
   Usage: Java JdbcUpd [<DB-URL>] [<userprefix>] [U] */

import java.sql.

parse arg url prefix lowup             -- process arguments
if url = '' then
   url = 'jdbc:db2:sample'
else do                                -- check for correct URL
     parse url p1 ':' p2 ':' rest
     if p1 \= 'jdbc' | p2 \= 'db2' | rest = '' then do
        say 'Usage: java JdbcUpd [<DB-URL>] [<userprefix>] [U]'
        exit 8
     end
end
if prefix = '' then prefix = 'userid'
if lowup \= 'U' then lowup = 'L'

do                                     -- loading DB2 support
  say 'Loading DB2 driver classes...'
  Class.forName('COM.ibm.db2.jdbc.app.DB2Driver').newInstance()
  -- Class.forName('COM.ibm.db2.jdbc.net.DB2Driver').newInstance()
catch e1 = Exception
  say 'The DB2 driver classes could not be found and loaded !'
  say 'Exception (' e1 ') caught : \n' e1.getMessage()
  exit 1
end                                    -- end : loading DB2 support

do                                     -- connecting to DB2 host
  say 'Connecting to:' url
  jdbcCon = Connection DriverManager.getConnection(url, 'userid', 'password')
catch e2 = SQLException
  say 'SQLException(s) caught while connecting !'
  loop while (e2 \= null)
     say 'SQLState:' e2.getSQLState()
     say 'Message: ' e2.getMessage()
     say 'Vendor:  ' e2.getErrorCode()
     say
     e2 = e2.getNextException()
   end
   exit 1
end                                    -- end : connecting to DB2 host

do                                     -- retrieve employee, update firstname

  say 'Preparing update...'                          -- prepare UPDATE
  updateQ = 'UPDATE' prefix'.EMPLOYEE SET firstnme = ? WHERE empno = ?'
  updateStmt = PreparedStatement jdbcCon.prepareStatement(updateQ)
  say 'Creating query...'                            -- create SELECT
  query = 'SELECT firstnme, lastname, empno FROM' prefix'.EMPLOYEE'
```

Figure 110 (Part 1 of 2). JDBC NetRexx Update Program: JdbcUpd.nrx

```
    stmt = Statement jdbcCon.createStatement()
    rs = ResultSet stmt.executeQuery(query)             -- execute select

    loop row=0 while rs.next()                          -- loop employees
      firstname = String rs.getString('firstnme')
      if lowup = 'U' then firstname = firstname.toUpperCase()
      else do
         dChar = firstname.charAt(0)
         firstname = dChar || firstname.substring(1).toLowerCase()
      end
      updateStmt.setString(1, firstname)                -- parms for update
      updateStmt.setString(2, rs.getString('empno'))
      say 'Updating' rs.getString('lastname') firstname ': \0'
      say updateStmt.executeUpdate() 'row(s) updated'   -- execute update
    end

    rs.close()                              -- close the ResultSet
    stmt.close()                            -- close the Statement
    updateStmt.close()                      -- close the PreparedStatement
    jdbcCon.close()                         -- close the Connection
    say 'Updated' row 'employees.'
  catch e3 = SQLException
    say 'SQLException(s) caught !'
    loop while (e3 \= null)
       say 'SQLState:' e3.getSQLState()
       say 'Message: ' e3.getMessage()
       say 'Vendor:  ' e3.getErrorCode()
       say
       e3 = e3.getNextException()
     end
  end                                       -- end: empoyees
```

Figure 110 (Part 2 of 2). JDBC NetRexx Update Program: JdbcUpd.nrx

When you execute this program, you will see one print line per employee:

```
d:\NrxRedBk\jdbc>java JdbcUpd jdbc:db2:sample userid
...
Updating MEHTA      Ramlal    : 1 row(s) updated
Updating LEE        Wing      : 1 row(s) updated
Updating GOUNOT     Jason     : 1 row(s) updated
Updated 32 employees.
```

Note: To reset the sample database to its original form, rerun the program with the third parameter set to U (java JdbcUpd jdbc:db2:sample userid U), or delete the sample database and rebuild it, using the DB2SAMPL program (see “DB2 Sample Database” on page 184):

```
db2 connect reset
db2 drop database sample
db2sampl
```

Update Sample Explanation

The first part of the program is identical to Figure 108 on page 186.

In the second part of the program we construct two statements: one prepared statement that updates the firstnme field of the current employee, and one query statement to loop through the employee table.

For the manipulation of the first name we use the Rexx concatenate operator (||) together with methods of the Java String class:

- String.charAt()
- String.substring()

- String.toLowerCase()

To set the two parameters of the prepared update statement, we use the setString method. The same update statement is executed for each employee, each time with the current value of the employee number and the changed first name. The actual update is invoked with executeUpdate, which returns the number of rows that were affected by the statement. In our case this will always be one row because we use the primary key column of the table to specify the row to be updated.

At the end of the program we close the different JDBC objects to release the resources.

Now you can rerun our sample select program (Figure 108 on page 186) to appreciate the changes the update program made to the employee table (see Figure 111).

```
Loading DB2 driver classes...
Connecting to  jdbc:db2:sample
Creating query...
Executing query : SELECT deptno, deptname, lastname, firstnme
FROM stade2.DEPARTMENT dep, stade2.EMPLOYEE emp WHERE dep.mgrno=emp.empno
ORDER BY dep.deptno
Result :
A00 SPIFFY COMPUTER SERVICE DIV. is directed by HAAS       Christine
B01 PLANNING                     is directed by THOMPSON   Michael
C01 INFORMATION CENTER           is directed by KWAN       Sally
D11 MANUFACTURING SYSTEMS        is directed by STERN      Irving
D21 ADMINISTRATION SYSTEMS       is directed by PULASKI    Eva
E01 SUPPORT SERVICES             is directed by GEYER      John
E11 OPERATIONS                   is directed by HENDERSON  Eileen
E21 SOFTWARE SUPPORT             is directed by SPENSER    Theodore
Retrieved 8 departments.
```

Figure 111. JDBC NetRexx Query Results after Update

Data Definition Language

In addition to the SQL data manipulation language (DML)—select, insert, update, and delete—we also have the SQL DDL. The DDL commands create, delete, or alter the table layout and are usually not part of a (user) program.

Executing DDL consists of basically setting up the correct SQL string and passing it to the database manager by using the executeUpdate method. We do not discuss DDL statements and execution in this redbook.

Stored Procedures

Another part of the JDBC API is dedicated to stored procedures, which are supported by many database engines.

JDBC SQL statements are executed as dynamic SQL. Stored procedures enable you to code a set of SQL statements in a compiled language, using static SQL, and then call such procedures from user programs written in NetRexx (Java) and other languages.

JDBC provides the callable statement to execute a stored procedure. The usage of stored procedures is, however, beyond the scope of this redbook.

Wrapping Up with a Complete JDBC GUI Program

We conclude this chapter with a full-blown database application that lets us list, change, add, and delete departments of our sample database.

The application is a fully functional GUI that uses the techniques described in Chapter 7, “Creating Graphical User Interfaces” on page 75. It could be a starting point for writing your own NetRexx client/server applications that use relational database access.

Figure 112 shows the program in action.

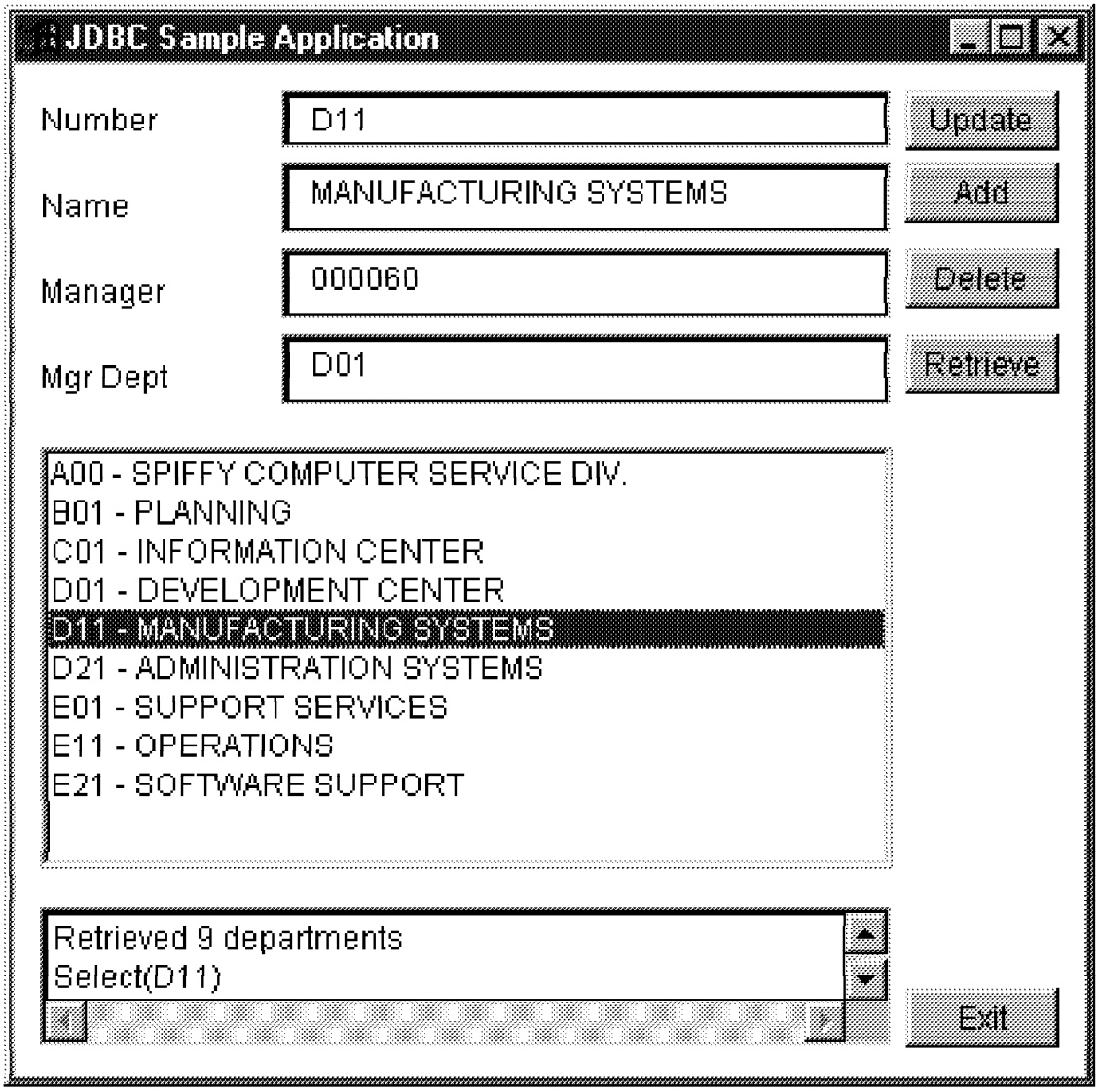


Figure 112. JDBC GUI Application

Figure 113 shows the source of the JDBC GUI application.

```
/* jdbc\JdbcGui.nrx

   JDBC + AWT in one sample application interfacing to the sample DEPARTMENT table
   Usage: Java JdbcGui [<DB-URL>] [<userprefix>] */

import Redbook.
import java.sql.

prefix      = Rexx                    -- table prefix
url         = Rexx                    -- database URL

parse arg url prefix                  -- process arguments
if url = '' then
   url = 'jdbc:db2:sample'
else do                               -- check for correct URL
     parse url p1 ':' p2 ':' rest
     if p1 \= 'jdbc' | p2 \= 'db2' | rest = '' then do
        say 'Usage: java JdbcGui [<DB-URL>] [<userprefix>]'
        exit 8
     end
end
if prefix = '' then prefix = 'userid'

say 'Creating GUI...'
GUI('JDBC Sample Application', url, prefix)


/********************************* class GUI *********************************/

class GUI uses GridBagConstraints

  Properties constant
    UPDATE   = int 0
    ADD      = int 1
    DELETE   = int 2
    RETRIEVE = int 3

  Properties static
    ctr           = Controller           -- controller object

  Properties inheritable
    tf_depno      = TextField(3)
    tf_depname    = TextField(29)
    tf_manager    = TextField(6)
    tf_mgrDept    = TextField(3)
    l_departments = List(10)
    ta_message    = TextArea(2,30)
    b1            = Button
    b2            = Button
    b3            = Button
    b4            = Button
    b5            = Button

  method GUI(title = String, pUrl, pPrefix)
    win = Frame(title)                  -- frame window
    gbl = SimpleGridbagLayout(win)      -- use gridbag layout
    ws = WindowSupport(win)             -- close window support

    buildLayout(gbl,ws)

    win.pack()
    RedbookUtil.positionWindow(win)
    win.setVisible(1)

    ctr = Controller(pUrl, pPrefix)

    b1.addActionListener(DoAction(this,ctr,UPDATE))
    b2.addActionListener(DoAction(this,ctr,ADD))
    b3.addActionListener(DoAction(this,ctr,DELETE))
    b4.addActionListener(DoAction(this,ctr,RETRIEVE))
    l_departments.addItemListener(DoAction(this,ctr))

    setMessage('Loading DB2 driver classes and connecting...')
    msg = ctr.connect()
    setMessage(msg)
    setDepartmentsList(ctr.retrieveListDep())
```

Figure 113 (Part 1 of 6). JDBC GUI Application: JdbcGui.nrx

```
  method buildLayout(gbl = SimpleGridbagLayout, ws=WindowSupport)
    gbl.addFixSize(Label('Number'),  0,0,Insets(10,10,5,30))
    gbl.addFixSize(Label('Name'),    0,1,Insets( 5,10,5,30))
    gbl.addFixSize(Label('Manager'), 0,2,Insets( 5,10,5,30))
    gbl.addFixSize(Label('Mgr Dept'),0,3,Insets( 5,10,5,30))

    gbl.addVarSize(tf_depno  ,1,0,Insets(10,0,5,5),1.0,0.0)
    gbl.addVarSize(tf_depname,1,1,Insets( 0,0,5,5),1.0,0.0)
    gbl.addVarSize(tf_manager,1,2,Insets( 0,0,5,5),1.0,0.0)
    gbl.addVarSize(tf_mgrDept,1,3,Insets( 0,0,5,5),1.0,0.0)
    tf_depno.addTextListener(LimitTextField(tf_depno,3))
    tf_depname.addTextListener(LimitTextField(tf_depname,29))
    tf_manager.addTextListener(LimitTextField(tf_manager,6))
    tf_mgrDept.addTextListener(LimitTextField(tf_mgrDept,3))

    gbl.addVarSize(l_departments,0,4,Insets(10,10,15,5),1.0,2.0,2)
    gbl.addVarSize(ta_message,   0,5,Insets( 0,10,10,5),1.0,1.0,2)

    b1=Button gbl.addFixSize(Button('Update')  ,2,0,Insets(10,0, 5,10),1,1,HORIZONTAL)
    b2=Button gbl.addFixSize(Button('Add')     ,2,1,Insets( 0,0, 5,10),1,1,HORIZONTAL)
    b3=Button gbl.addFixSize(Button('Delete')  ,2,2,Insets( 0,0, 5,10),1,1,HORIZONTAL)
    b4=Button gbl.addFixSize(Button('Retrieve'),2,3,Insets( 0,0, 5,10),1,1,HORIZONTAL)
    b5=Button gbl.addFixSize(Button('Exit')    ,2,5,Insets( 0,0,10,10),1,1,HORIZONTAL,SOUTHWEST)
    b5.addActionListener(ws.getCloseWindow())

    cb = ControlButtons()
    cb.addButton(b1)
    cb.addButton(b2)
    cb.addButton(b3)
    tf_depno.addTextListener(cb)
    cb.disable

  method setDepartmentsList(aList = Rexx)
    l_departments.removeAll()
    loop i = 0 for aList['count']
      l_departments.add(aList[i])
    end
    setMessage(aList['message'])

  method setDepno(s=String)
    tf_depno.setText(s)

  method setName(s=String)
    tf_depname.setText(s)

  method setMgNo(s=String)
    tf_manager.setText(s)

  method setMgDepNo(s=String)
    tf_mgrDept.setText(s)

  method getDepno() returns String
    return tf_depno.getText()

  method getName() returns String
    return tf_depname.getText()

  method getMgNo() returns String
    if tf_manager.getText() = '' then return null
                                 else return tf_manager.getText()

  method getMgDepNo() returns String
    return tf_mgrDept.getText()

  /*---------------------------- interface to display messages -----------------*/
  method setMessage(msg = String)
    say msg
    ta_message.append('\n'msg)

/******************************** class DoAction ***************************/

class DoAction implements ActionListener, ItemListener uses GUI
  Properties inheritable
    ui  = GUI         -- reference to user interface
    ctr = Controller  -- reference to the controller
```

Figure 113 (Part 2 of 6). JDBC GUI Application: JdbcGui.nrx

```
    act = int         -- which action

  method DoAction(aGui=GUI,aControl=Controller,what=int 0)
    ui  = aGui
    ctr = aControl
    act = what

  method actionPerformed(e=ActionEvent)
    select
      when act = UPDATE then do
        ui.setMessage('Updating a row in the DEPARTMENT table...')
        msg = ctr.update(ui.getDepno(),ui.getName(),ui.getMgNo(),ui.getMgDepNo())
        ui.setMessage(msg)
        ui.setDepartmentsList(ctr.retrieveListDep())
      end
      when act = ADD then do
        ui.setMessage('Inserting a row in the DEPARTMENT table...')
        msg = ctr.add(ui.getDepno(),ui.getName(),ui.getMgNo(),ui.getMgDepNo())
        ui.setMessage(msg)
        ui.setDepartmentsList(ctr.retrieveListDep())
      end
      when act = DELETE then do
        ui.setMessage('Deleting a row in the DEPARTMENT table...')
        msg = ctr.delete(ui.getDepno())
        ui.setMessage(msg)
        ui.setDepartmentsList(ctr.retrieveListDep())
      end
      when act = RETRIEVE then do
        ui.setDepartmentsList(ctr.retrieveListDep())
        ui.setDepno('')     -- clear the fields
        ui.setName('')
        ui.setMgNo('')
        ui.setMgDepNo('')
      end
    end

  method itemStateChanged(e=ItemEvent)
    s = (List e.getItemSelectable()).getSelectedItem()
    if s \= null then do
      parse s depno '-' .
      msg = ctr.select(depno.strip())
      ui.setMessage(msg)
      ui.setDepno(depno)     -- set the fields
      ui.setName(ctr.getDepName())
      ui.setMgNo(ctr.getMgNo)
      ui.setMgDepNo(ctr.getMgDepNo())
    end

/********************************* class ControlButtons *********************/

class ControlButtons implements TextListener
 Properties inheritable
   buttons = Vector()
   enabled = boolean 1

 method addButton(aButton = Button)
   buttons.addElement(aButton)

 method textValueChanged(e = TextEvent)
   field = TextField e.getSource()
   text = field.getText()
   if text \= null then
     if text = '' then disable()
                  else enable()
   else disable()

 method enable()
   if enabled then return
   enabled = 1
   change()

 method disable()
   if \enabled then return
   enabled = 0
   change()
```

Figure 113 (Part 3 of 6). JDBC GUI Application: JdbcGui.nrx

```
 method change()
   loop i=0 for buttons.size
     (Button buttons.elementAt(i)).setEnabled(enabled)
   end

/********************************* class Controller *************************/

class Controller

/*-------------------------- variables -----------------------------*/
Properties inheritable
  prefix      = Rexx 'userid'          -- table prefix
  url         = Rexx ''                -- DB URL
  jdbcCon     = Connection             -- DB Connection
  selDepNo    = String ''              -- selected row values
  selName     = String ''
  selMgNo     = String ''
  selMgDepNo  = String ''

/*-------------------------- constructor ---------------------------*/
method Controller(pUrl = Rexx, pPrefix = Rexx)
    url = pUrl
    prefix = pPrefix

/*-------------------------- connect() -----------------------------*/
method connect() returns String
  do
    Class.forName('COM.ibm.db2.jdbc.app.DB2Driver').newInstance()
    -- Class.forName('COM.ibm.db2.jdbc.net.DB2Driver').newInstance()
    jdbcCon = Connection DriverManager.getConnection(url, 'userid', 'password')
    msg = 'Connected to' url
  catch e2 = SQLException
    msg = 'SQLException(s) caught while connecting !'
    loop while (e2 \= null)
       say 'SQLState:' e2.getSQLState()
       say 'Message: ' e2.getMessage()
       say 'Vendor:  ' e2.getErrorCode()
       say
       e2 = e2.getNextException()
    end
  catch e1 = Exception
    msg = 'Exception in DB2 Driver loading !'
    say 'Exception (' e1 ') caught : \n' e1.getMessage()
  end
  return msg

/*-------------------------- retrieveListDep() ----------------------*/
method retrieveListDep() returns Rexx
  deptarr = Rexx ''
  do
    query = 'SELECT deptno, deptname, mgrno, admrdept' -
             'FROM' prefi'.DEPARTMENT ORDER BY deptno'
    stmt = Statement jdbcCon.createStatement()
    rs = ResultSet stmt.executeQuery(query)
    loop row=0 while rs.next()
         deptarr[row] = rs.getString('deptno') '-' rs.getString('deptname')
    end
    deptarr['count'] = row
    rs.close()                         -- close the result set
    stmt.close()                       -- close the statement
    deptarr['message'] = 'Retrieved' row 'departments'
  catch ex=SQLException
    deptarr['message'] = ex.getMessage()
    deptarr['count'] = 0
  end
  return deptarr

/*-------------------------- update() ------------------------------*/
method update(depNo=String, name=String, mgNo=String, mgDepNo=String) -
       returns String
  do
    if depNo = '' then
      return 'Missing department number for update !'
    if depNo <> selDepNo then          -- If depNo already exists you'll
      do                               -- loose the department at selDepNo
         tempDepNo = selDepNo          -- Room for improvement...
```

Figure 113 (Part 4 of 6). JDBC GUI Application: JdbcGui.nrx

```
          add(depNo, name, mgNo, mgDepNo)
          delete(tempDepNo)
          return 'Department number updated (delete old, add new)'
        end
      updQuery = 'UPDATE' prefix'.DEPARTMENT' -
                 "SET deptname = '"name"'," -
                     "mgrno = '"mgNo"'," -
                     "admrdept = '"mgDepNo"'" -
                 "WHERE deptno = '"depNo"'"
      say 'Update SQL query :' updQuery
      stmt = Statement jdbcCon.createStatement()
      stmt.executeUpdate(updQuery)
      msg = 'Replaced a department.'
      stmt.close()
      selDepNo = ''
    catch ex=SQLException
      msg = ex.getMessage()
    end
    return msg

  /*-------------------------- delete() -------------------------------*/
  method delete(depNo=String) returns String
    do
      if depNo = '' then
         return 'Missing department number for delete'
      delQuery = 'DELETE FROM' prefix'.DEPARTMENT' -
                 "WHERE deptno = '"depNo"'"
      say 'Delete SQL query :' delQuery
      stmt = Statement jdbcCon.createStatement()
      stmt.executeUpdate(delQuery)
      msg = 'Deleted a department.'
      stmt.close()
      selDepNo = ''
    catch ex=SQLException
      msg = ex.getMessage()
    end
    return msg

  /*-------------------------- add() ----------------------------------*/
  method add(depNo=String, name=String, mgNo=String, mgDepNo=String) -
         returns String
    do
      query = 'SELECT deptno, deptname, mgrno, admrdept' - --test uniqueness of depNo
              'FROM' prefix'.DEPARTMENT' -
              "WHERE deptno = '"depNo"'"
      stmt = Statement jdbcCon.createStatement()
      rs = ResultSet stmt.executeQuery(query)
      if rs.next() then
        return ''depNo 'is in use, choose another department number.'
      rs.close()                          -- Close the result set
      stmt.close()                        -- Close the statement
      addQuery = 'INSERT INTO' prefix'.DEPARTMENT' -
                 '(deptno, deptname, mgrno, admrdept)' -
                 "VALUES ('"depNo"','"name"','"||-
                 mgNo"','"mgDepNo"')"
      say 'Insert SQL query :' addQuery
      stmt = Statement jdbcCon.createStatement()
      stmt.executeUpdate(addQuery)
      msg = 'Inserted a department.'
      stmt.close()
    catch ex=SQLException
      msg = ex.getMessage()
    end
    return msg

  /*-------------------------- select() -------------------------------*/
  method select(depNo=String) returns String
    if (selDepNo <> depNo) then
      do
        msg = 'Select('depNo')'
        query = 'SELECT deptno, deptname, mgrno, admrdept' -
                'FROM' prefix'.DEPARTMENT' -
                "WHERE deptno = '"depNo"'"
        stmt = Statement jdbcCon.createStatement()
        rs = ResultSet stmt.executeQuery(query)
        rs.next()
```

Figure 113 (Part 5 of 6). JDBC GUI Application: JdbcGui.nrx

```
        selDepNo   = depNo
        selName    = rs.getString('deptname')
        selmgNo    = rs.getString('mgrno')
        selmgDepNo = rs.getString('admrdept')
        rs.close()                          -- Close the result set
        stmt.close()                        -- Close the statement
      catch ex=SQLException
        msg = 'Select() : 'ex.getMessage()
      end
    else msg = 'Selected department number did not change'
    return msg

  /*------------------------- getDepName() -----------------------------*/
  method getDepName() returns String
    return selName

  /*------------------------- getMgNo() --------------------------------*/
  method getMgNo() returns String
    return selMgNo

  /*------------------------- getMgDepNo() -----------------------------*/
  method getMgDepNo() returns String
    return selMgDepNo

  -- end JdbcGui
```

Figure 113 (Part 6 of 6). JDBC GUI Application: JdbcGui.nrx

The JDBC GUI application can be divided into three parts:

- The main part, which parses the arguments and constructs the GUI
- The GUI part, which consists of three classes and uses some classes of the Redbook package
- The Controller class, which encapsulates the JDBC part of the program

The Controller class does not reference the user interface; all communication to the GUI is through the returned data from the controller's methods. Thus, it would be easy to write another user interface (maybe a character-based one) instead of the GUI presented here.

The user interface invokes the controller program logic through the following interface:

Controller(pUrl = Rexx, pPrefix = Rexx)
Provides the user interface with a Controller (the arguments are the URL and table prefix for the database connection)

connect() Connects the Controller object to the database

retrieveListDep() Returns the list of departments as a formatted indexed Rexx string. The *count* index contains the number of departments, and indexes 0 to count contain the rows.

select(depNo=String)
Selects the department with the given department number to be the current department. Select must be used to set the current department before the get methods are used to access the different fields of the department.

delete(depNo=String)
Deletes the department identified by depNo

update(depNo=String, name=String, mgNo=String, mgDepNo=String)
Updates the current department with the given values. If the department number is changed, this request is translated into an add new and delete old department combination.

add(depNo=String, name=String, mgNo=String, mgDepNo=String)
Creates a new department in the table if the given depNo does not already exist

getDepName() Gives the name of the current department as a String object

getMgNo() Gives the employee number of the manager of the current department as a String object

getMgDepNo() Gives the department number of the managing department (field admrdep of the department table) as a String object

Most methods return a Java string with a message that the GUI displays to the user.

This program does contain some flaws; for example, when changing the department number on an update, and the new department already exists (see comment in the source code of the update method). However, considering the length of the code, you have a small program that shows all of the basic database capabilities with a reasonable amount of checking.

Client/Server Program

We use this example again in Chapter 11, “Network Programming” on page 205, where we split the code to run the controller with the database access on a machine different from the client with the GUI. See “Wrapping Up with a Complete RMI Program” on page 227 for further information.

Chapter 11. Network Programming

In this chapter we discuss the (basic) networking features of NetRexx and how they can help you develop client/server applications.

Because NetRexx implies Java, and Java implies the Internet, everything is based on the (good old) TCP/IP protocol. The most frequently used TCP/IP interface is the socket interface that was originally developed for UNIX.

Another interesting feature of Java that we touch on in this chapter is remote method invocation (RMI), that is, remotely invoking methods on Java server objects, and providing you with a simple way to create your own server classes.

We also touch on the built-in World Wide Web related facilities of Java, namely the URL classes.

We do not explain the use of the Datagram classes because that would lead us too far into explaining the Internet protocol (IP) details. Please consult the appropriate books (for example, *UNIX Networking Programming*) in combination with the JDK API reference on such subjects.

In summary, in this chapter we explain how to work with sockets, invoke remote methods from within NetRexx, and interact with the Web.

Socket Interface

The basic socket operations are provided through the *Socket* and *ServerSocket* classes, both part of the java.net package.

Socket

A Socket object is a high-level representation of an IP socket. A socket is uniquely defined by the combination of an IP address and port number, and that is all you need to construct an object of the Socket class:

```
ourFirstSocket = Socket('www.ibm.com',80)
```

For such a socket we have two properties, *inputStream* and *outputStream*, that can be accessed through the respective get methods, *getInputStream* and *getOutputStream*. An HTTP server usually listens to port 80.

Let us extend our example to read the first lines of a home page. HTTP communication uses this protocol to start a conversation:

- The client sends a message asking for a file and optionally supplies additional parameters (that are beyond the scope of this book). The request message has the form:

  ```
  GET /file.html
  ```

- The HTTP server reads the message and sends the page (file) for which the client asked.

Sending a Request to an HTTP Server

Figure 114 shows a simple implementation of the HTTP protocol. The client sends a request to an HTTP server, asking for an HTTP page, and prints out the response of the server as plain text.

```
/* network\net\ClntSock.nrx

   Client HTTP program, sends a request to an HTTP server:
   Usage: Java ClntSock <server> <portnumber> <requeststring> */

parse arg server port str              -- capture + test arguments
if server='' | port='' then
  do
    say 'Usage: java ClntSock <server> <portnumber> <requeststring>'
    exit 1
  end

parse str get rest                     -- check requeststring
if get <> '' then do
   if get <> 'GET' then do
     say 'Request string must be: GET /http-page'
     exit 8
   end
   str = 'GET' rest                    -- make get uppercase
end

do                                     -- ready to process
  say 'Connecting to server:' server '(port:' port')'
  mysocket = Socket(server, port)      -- actual connect
  say 'Requesting:' str                -- what we want
  say
                                       -- output: send
  out = PrintWriter(OutputStreamWriter(mysocket.getOutputStream()))
                                       -- input: receive
  in  = BufferedReader(InputStreamReader(mysocket.getInputStream()))
  out.printLn(str)                     -- send our requeststring
  out.flush()                          -- needed on some platforms !
  say 'Response:'
  line = String(in.readLine())         -- read response
  loop while line \= null
     say ' ' line                      -- print it out
     line = in.readLine()
  end
catch e=IOException
  say 'IOException (' e ') caught:\n' e.getMessage()
end
-- end ClntSock
```

Figure 114. Simple HTTP Client Program: CnltSock.nrx

The first thing this program does is check the user's input supplied on the command line. This is really simple to do through the NetRexx parse facility.

Next, we create our Socket object and initialize local variables to refer to the input and output streams associated with the socket. All communication through a socket is really done through file i/o.

Then the request is sent by using the print output stream object, and the results from the server are read and printed by using the input stream object.

We also check for possibly raised exceptions (see “Exceptions” on page 30). If an exception occurs, we print it out with its short description (implicit invocation of toString on the exception object e) and full message text. This full message text gives a specific description of the current exception, if the specific exception supports this. Otherwise you get the short description as returned by the toString method.

Testing the Simple HTTP Client

The client program is started with:

```
d:\NrxRedBk\network\net>java ClntSock w3.ibm.com 80 get /
                        java ClntSock www.any.business.com 80 get /somepage.html
```

Sending the “GET /” request to a server usually results in the return of the default page.

Adjust the name of the server so that it points to an HTTP server that is directly accessible within your Intranet.

You can also send commands to other ports of a server, for example, to the FTP port (21):

```
d:\NrxRedBk\network\net>java ClntSock w3.almaden.ibm.com 21
Connecting to server: w3.almaden.ibm.com (port: 21)
...
Response:
  220 spider.almaden.ibm.com FTP server (Version 4.4 Tue Aug 8 15:39:30 CDT 1995) ready.
  500 '': command not understood.
```

ServerSocket

Let us now look at the sibling of the Socket class on the server side, the *ServerSocket*. To register a server on a port, for example, HTTP port 80, we construct an object of the ServerSocket class:

```
ourFirstServerSocket = ServerSocket(80)
```

Then we listen to that port and accept a possible client. This is done by invoking the *accept* method on the ServerSocket object. The accept method waits for a client request:

```
clientSocket = ourFirstServerSocket.accept()
```

The accept method blocks the execution of the current thread until a client makes a connection. An object of the Socket class is returned when a client connects, the so-called service or session socket. This socket represents the actual socket through which the client and server communicate.

To support simultaneous connections from multiple clients to the server, we have to run the code that handles the conversation with the client in a separate thread (see “Extended Server with Threads” on page 210).

Accepting an HTTP Client Request

TO illustrate basic server coding, let us now write a simple HTTP server class that does not use threads. This HTTP server runs on your own machine. Therefore, contact your administrator to ask whether you are allowed to register additional socket services on your machine.

Figure 115 shows the simple HTTP server code.

```
/* network\net\SrvSock.nrx                    -- <pre>

   Server HTTP program, accepts a request from an HTTP client:
   Usage: Java SrvSock <portnumber> */

do
  if arg = ''  then arg = 80                  -- default port
  serverS = ServerSocket(arg)                 -- register at port: server socket
  say 'Server:' serverS
  loop forever
    serviceS = serverS.accept()               -- listen/accept client: service socket
    say serviceS '\n  connected at:' Date()
    ptrW = PrintWriter(OutputStreamWriter(serviceS.getOutputStream()))
    sIS  = BufferedReader(InputStreamReader(serviceS.getInputStream()))
    loop while sIS.ready()                    -- consume HTTP request
      line = String(sIS.readLine())
    end
    filename = 'SrvSock.nrx'                  -- always returning the source file
    fileBR = BufferedReader(FileReader(filename))
    line = String(fileBR.readLine())
    loop while(line <> null)                  -- add lines of source file
       ptrW.printLn(line)
       line = fileBR.readLine()
    end                                       -- end loop while(line <> null)
    ptrW.close()                              -- close output and socket
    serviceS.close()
  catch e=IOException
    say 'IOException caught:' e.getMessage() -- error messaging
  end                                         -- end loop forever
end
-- end SrvSock                                -- </pre>
```

Figure 115. Simple HTTP Server Program: SrvSock.nrx

Here we first register our code on the port number that was supplied as an argument. (Notice the bad use of *arg*; the input should be checked!)

The default method, *toString*, is invoked in the *say 'Server:' serverS* instruction. It returns much of the available information of the socket in a formatted string.

After creating the ServerSocket, we start a loop that listens (forever) to incoming clients.

When a client is accepted, we store the references to the input and output streams that are used for communication.

We then run a loop that empties the input stream. It seems that on some platforms additional lines might be present, so, to fulfill the Java promise of real platform independence, we adapt the code to the most demanding platform.

Our simple server always returns the source code of the server program itself. Note the *<pre>* tag in the first line of the server source; it ensures that browsers do not format the source program. To optimize performance, we use a buffer to read the source file.

We end our main loop by closing the output stream and service socket.

Exception handling is used to catch any input/output errors on the port.

Testing the Simple HTTP Server

Start the simple HTTP server:

```
d:\NrxRedBk\network\net>java SrvSock 80
```

Start your favorite browser and point it to the server:

```
http://serverhostname/
```

The result should be the unformatted source code of the server program.

You can also combine the simple client and server. Start the simple client in a separate window, or on a separate machine, and send this request:

```
d:\NrxRedBk\network\net>java ClntSock serverhostname 80
```

To end the simple server, use Ctrl-C in its window.

More on Sockets

Both the Socket and ServerSocket classes are implemented by the *SocketImpl* class. Therefore, you can apply the same methods to both the Socket and the ServerSocket objects to obtain more information from them. This information is stored in the four attributes of their SocketImpl object:

address The IP address of the remote end of this socket

fd The file descriptor object for this socket

localport The serviceport to which this socket is connected

port The port number on the remote machine

Remote in the first and last definition means the server, if you are using the socket on the client side, and the client, if you are invoking the methods on a ServerSocket.

These attributes enable you to use a set of the most commonly used methods:

- getInetAddress()
- getInputStream()
- getOutputStream()
- getLocalPort()
- getPort()
- toString()

To use the input and output streams returned by the getInputStream and getOutputStream methods, you convert them to one of the—more useful—subclasses of InputStream and OutputStream.

For textual (character) communication you should convert these streams to Reader and Writer (sub)classes, using the InputStreamReader and OutputStreamWriter *adapter* classes. The use of Readers and Writers greatly simplifies *internationalization* problems of having to deal with many code pages, keyboard layouts, and so forth. These I/O classes are the same as discussed in Chapter 9, “Handling Files” on page 165.

Extended Server with Threads

To enable the server to *talk* with multiple clients simultaneously, server sockets should hand over the conversation with a client socket to a separate thread. (For more explanation on threads, see Chapter 8, “Threads” on page 153.)

The new thread that gets the client socket should be created just after the acceptance of a client, that is, after calling this method:

```
ServerSocket.accept()
```

We also extend the server program to check the argument and to accept a file name in the request. If the specified file does not exist, the server returns an error message; if a file name is not given, it returns its own source code.

Note: This version of our server provides any *surfing client* within your Intranet with a (simple) viewer to inspect all files accessible through the file system of your computer.

Figure 116 shows the extended server with threads.

```
/* network\net\SrvSockT.nrx

   Server HTTP program, accepts a request from an HTTP client using a Thread:
   Usage: Java SrvSockT <portnumber> */

parse arg port .                              -- capture + test argument
if port = '' then port = 80
if \port.datatype('W') then
  do
    say 'Usage: java SrvSockT <portnumber>'
    exit 8
  end

do                                            -- main program loop
  serverS = ServerSocket(port)                -- register at port: server socket
  say 'Server:' serverS
  loop forever
    serviceS = serverS.accept()               -- listen/accept client: service socket
    st = ServerThread(serviceS)               -- create a Thread (constructor)
    Thread(st).start()                        -- start the Thread
  end                                         -- end loop forever
catch e=IOException
  say 'IOException caught:' e.getMessage()    -- error messaging
end

/*-------------------------- class ServerThread ----------------------*/
class ServerThread implements Runnable
  Properties inheritable
    serviceS = java.net.Socket
    html     = byte 0

  method ServerThread(s = java.net.Socket)    -- constructor
    serviceS = s

  method getRequest() returns Rexx            -- analyze request string
    line = Rexx
    socketInputStream    = serviceS.getInputStream()
    socketBufferedReader = BufferedReader(InputStreamReader(socketInputStream))
```

Figure 116 (Part 1 of 2). HTTP Server Program Using Threads: SrvSockT.nrx

```
                                          -- first line has "get" request
    line = socketBufferedReader.readline()
                                          -- skip the rest
    socketBufferedReader.skip(socketInputStream.available())
    say 'Request received:' line          -- analyze the line
    parse line get '/' file ' ' .
    if file = '' | get \= 'GET' then file = 'SrvSockT.nrx'
    say 'File requested:' file            -- requested file
    return file

  method run()                            -- run the Thread
  do
    say serviceS '\n  connected at:' Date()
    ptrW = PrintWriter(OutputStreamWriter(serviceS.getOutputStream()))
    file = getRequest()                   -- read the request of the client
    if file.right(4) = '.htm' | -
       file.right(5) = '.html' then html = 1
    do                                    -- open requested file
      fileBR = BufferedReader(FileReader(file))
      line = String(fileBR.readLine())
      if html=0 then ptrW.printLn('<' || 'pre>')  -- make page: unformatted file
      loop while(line <> null)            -- add lines of file
         ptrW.printLn(line)
         line = fileBR.readLine()
      end                                 -- end while(line <> null)
      if html=0 then ptrW.printLn('<' || '/pre>')
    catch FileNotFoundException           -- notify when file not found
      ptrW.printLn('Sorry, file "'file'" not found')
    end
    ptrW.close()                          -- close file and socket
    serviceS.close()
  catch e=IOException
    say 'IOException caught in ServerThread.run():' e.getMessage()
  end
-- end SrvSockT
```

Figure 116 (Part 2 of 2). HTTP Server Program Using Threads: SrvSockT.nrx

The extended server defines two classes:

- The SrvSockT class, with the—implicitly defined—static main method
- The ServerThread class, which implements *Runnable*

After checking and initializing the parameter, we register our code and start listening.

When a client is connected, we create an object of the ServerThread class and pass it the session socket; then we start the thread.

We continue the loop and wait for another client.

The run method of the ServerThread starts reading the request sent by the client.

We retrieve the first input line and extract the file name from the request. Note that browsers add additional information after the file name. A call to getRequest filters out the requested file. For invalid or empty requests, we use the source of the program instead.

We return the file in unformatted mode, using the <pre> tag, unless the file is an HTML file (.htm or .html).

If the file cannot be found, we write an error message in return.

Testing the Extended HTTP Server

Start the extended HTTP server:

```
d:\NrxRedBk\network\net>java SrvSockT
```

Start your favorite browser and point it to the server:

```
http://serverhostname/test.html
http://serverhostname/
http://serverhostname/c:\anydirecory\any.file
```

The first example returns a small HTML file that is supplied in the server directory; the second returns the server program source code.

You can also combine the simple client and the extended server. Start the simple client in a separate window, or on a separate machine, and send this request:

```
java ClntSock serverhostname 80 get /test.html
java ClntSock serverhostname 80 get /c:\anydirectory\any.file
```

Using the two programs, you get the same effect as printing files from your server source directory with system commands such as *type* or *cat*.

To end the extended server, use Ctrl-C in its window.

Socket Conclusion

The three examples in this section demonstrate the use of the high-level socket implementation of Java to establish a general IP conversation in NetRexx between any two machines connected to the Internet.

In the next section we discuss an even more specific class for interfacing with the Internet. It fully implements the different higher level protocols, such as FTP and HTTP.

URL Handling

The URL class provides an easy way of manipulating URLs. The URL strings define everything that can be accessed through the Internet, so you can use URL objects in your NetRexx programs to access Internet resources.

A common URL string contains five fields:

```
<protocolname>://<hostname>[:<portnumber>]/<filepath>[#<reference>]
```

See “Database URLs” on page 182 for database specific URL notation.

You can access the URL fields with these methods:

- getProtocol()
- getHost()
- getPort()
- getFile()
- getRef()
- set(String, String, int, String, String)

Other useful methods of the URL class include:

equals(Object) Compares two URLs, including the reference field. This method overrides the Object.equals method and therefore accepts an object as its argument.

sameFile(URL) Compares two URLs, excluding the reference field

getContent() Returns the contents of the URL, that is, the remote object to which the URL refers

openConnection() Returns a URLConnection object that represents a connection to the remote object to which the URL refers

openStream() Opens a connection to the URL, and returns an InputStream for reading from that connection

toExternalForm() and toString()
Constructs a string representation of the URL using the URLStreamHandler.toExternalForm method, which formats the URL by using the conventions of the specified protocol

The most common action we want to apply to URLs is asking the server to return the *object* behind the URL; whatever the *content-type* is. The content-types are also known as *MIME-types*. We do this by using the *getContent* method.

Because the getContent method returns an object of type *Object*, you should cast the result to the right type upon reception. Standard types include AudioClip, ImageProducer, and InputStream. Let us look at how this powerful method manages to get the requested object.

Getting the Content of an URL

The getContent method of our URL object passes this message to the associated *URLConnection* object that it gets using its own method:

```
URLConnection.openConnection()
```

The URLConnection class represents an active connection with the host. The first time the getContent method gets called, the URL object sends a message (*openConnection(URL)*) to the appropriate URLStreamHandler (obtained from the URLStreamHandlerFactory) to get the URLConnection object.

The URLConnection then calls the *ContentHandlerFactory* to get an appropriate *URLContentHandler*. A contenthandler handles a certain content-type.

The URLConnection instance then sends the getContent message to this URLContentHandler object, which then—finally—returns the object behind the URL.

Refer to Figure 117 for an abstract representation of this process.

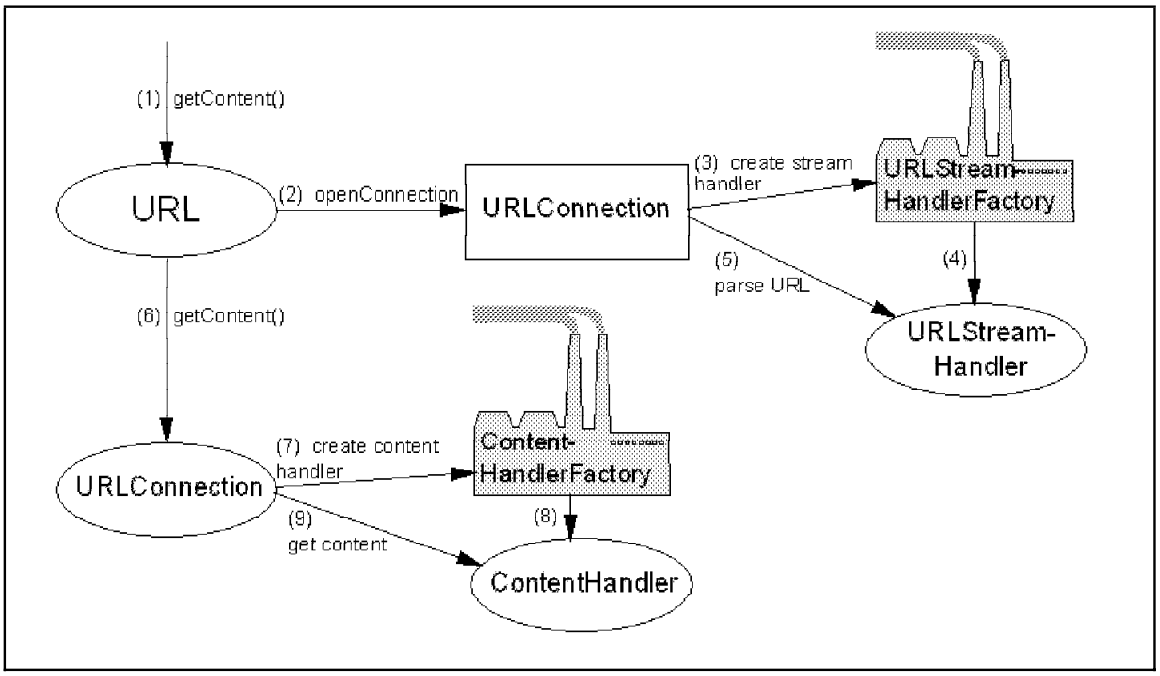


Figure 117. URL Content Handling

This complex construction—which we did not even explain in full detail—ensures a very flexible way of extending the model later on with new protocols and content-types.

Design Pattern Background

This construction is based on the *Abstract Factory* design pattern (see *Design Patterns: Elements of Reusable Object-Oriented Software*). The pattern fully decouples the abstract structure of a design from its implementations. It offers users of abstract classes more flexibility than subclassing the implementations offers. Users do not have to decide or know which implementation of the classes will be used in the end.

The *Factory Methods* (for URLs, the createURLStreamHandler(String) and createContentHandler(String) methods) of the factory provide the client (the code that asks for an URL object) with the appropriate (concrete) version of the object, for which the client only has to know its interface.

The pairs Socket (and ServerSocket) and SocketImpl are also based on a design pattern (described in *Design Patterns: Elements of Reusable Object-Oriented Software*), the Bridge pattern, also known as Handle/Body. This design pattern is based on the *delegation* paradigm—a class encapsulates an object of another class to delegate a part of its responsibilities.

HTTP Client Using URLs

The getContent method makes the writing of an HTTP client very easy. We can use getContent to get the (string) contents of an URL, similar to the Socket example (see “Sending a Request to an HTTP Server” on page 206). Note, however, that this version is more robust; it will handle firewalls (as long as they are configured in your Java Virtual Machine) so that you can reach everything on the Internet like a normal browser. This is the advantage of using high-level classes such as URLs. You do not need to know the process of the protocol on the socket in order to communicate using this protocol. All you need is the name of the protocol, the first part of a URL string.

The sample program in Figure 118 can print out any freely accessible file on the Internet.

```
/* network\url\UrlTest.nrx

   Client HTTP program, sends a URL request to an HTTP server:
   Usage: Java UrlTest <URLstring> */

parse arg prot '://' request         -- protocol & servername are required
if prot = '' | request = '' then
  do
    say 'Usage: java UrlTest <URLString>'
    exit 8
  end

do
  ourURL = URL(arg)                  -- create URL object
  say 'Connect to:' ourURL
  content = ourURL.getContent()      -- get the URL content
  say '\nContent:\n'
  bR = BufferedReader(InputStreamReader(InputStream content))
  line = String(bR.readLine())       -- read and print file
  loop while(line <> null)
     say ' ' line
     line = bR.readLine()
  end
catch e=IOException
  say 'IOException caught:' e '\n  ' e.getMessage()
end
-- end UrlTest
```

Figure 118. HTTP Client Using URLs: UrlTest.nrx

First we perform a quick validation of the supplied argument by checking for the sequence :// and return the proper form of invoking the program if the sequence is missing. This test also covers arguments such as /h, /?, -help, and -?.

Other arguments that contain the :// sequence but are invalid URLs are captured by an IOException. For example, if the protocol part of the URL string is mistyped, we get:

```
d:\NrxRedBk\network\url>java UrlTest httx://host
IOException caught: java.net.MalformedURLException: unknown protocol: httx
   unknown protocol: httx
```

We elaborate on exceptions in “Typical Network Exception Types” on page 218.

Next the program creates a URL object and uses the getContent method to obtain the requested object. The content is converted—using the InputStreamReader class—into a BufferedReader for an easy way of reading the lines in the file and writing them to the console.

Testing the URL Client Program

Examples of running the URL client program are:

```
d:\NrxRedBk\network\url>java UrlTest http://w3.icm.com
                        java UrlTest http://hostname/some.html
                        java Urltest file://hostname/directory/free.file
```

You can also run the program against the server program used in “Extended Server with Threads” on page 210.

Content Handlers

When you experiment with the sample program, you will soon encounter yet another kind of exception, such as:

```
java.lang.ClassCastException:
        at UrlTest.main(UrlTest.nrx:18)
```

This exception indicates that the createContentHandler method derived a content-type (using the *response header* or by guessing the type) and associated it with a specific contentHandler. The corresponding object was created but could not be converted to an object of type InputStream, hence the ClassCastException. For example, you get this exception when you try to access a picture:

```
java UrlTest http://hostname/some.gif
```

Useful information for gaining insight into the supported MIME-types and their associated content-types is available in this file:

```
<Java-home-directory>/lib/content-types.properties
```

You can also make your own content handler class for some unimplemented existing MIME-type, or for a content-type that is the fruit of your own imagination. For writing content handlers we advise you to read the example that lists the files in a UNIX *tar* file in *Exploring Java*.

Images: Images are usually converted to objects of a class that implements the ImageProducer interface. Objects that implement ImageProducer (in the *java.awt.image* package) can be connected to an (object that implements) ImageConsumer, which—in turn—can be part of a GUI. For more details on how to interact with images, consult “Images” on page 118.

Audio: For objects of type AudioClip (in the *java.applet* package), we can call the message:

```
AudioClip.play()
```

which plays the audio file one time, always starting from the beginning.

Extended HTTP Client Using URLs

If we now use the <= operator of NetRexx, which is the equivalent of Java′s instanceOf operator, to determine the object type, we can start to access other URL content types and act on them appropriately.

Figure 119 shows an example of how to handle a few of the standard content handlers. It can form a base implementation for a more general *ULR-test* utility.

```
/* network\url\UrlXTest.nrx

   Client HTTP program, sends a URL request to an HTTP server
   and handles a few content types:
   Usage: Java UrlXTest <URLstring> */

parse arg prot '://' request         -- protocol & servername are required
if prot = '' | request = '' then
```

Figure 119 (Part 1 of 2). Extended HTTP Client Using URLs: UrlXTest.nrx

```
    do
      say 'Usage: java UrlXTest <URLString>'
      exit 8
    end

  do
    ourURL = URL(arg)                   -- create URL object
    say 'Connect to  :' ourURL
    urlConn = ourURL.openConnection()  -- get connection object
    conttype = Rexx urlConn.getContentType()  -- content type
    say 'Content-type:' conttype
    content = urlConn.getContent()     -- get the URL content
    say 'Class       :' content.getClass().getName()
    select
      when content <= ImageProducer then   -- picture ?
           say '==> Content is an ImageProducer'
      when content <= AudioClip then       -- music ?
           do
             say '==> Content is an AudioClip, playing.....'
             (AudioClip content).play()
           end
      when content <= InputStream then     -- InputStream implementers
           do
             say '==> Content is an InputStream'
             say '\nFile content:\n'
             bR = BufferedReader(InputStreamReader(InputStream content))
             line = String(bR.readLine())
             loop while(line <> null)
                  say ' ' line
                  line = bR.readLine()
             end
           end
      otherwise
           say '==> Sorry, content type not handled!'
    end                                 -- end select
  catch e=IOException
    say 'IOException caught:' e '\n  ' e.getMessage()
  end
  -- end UrlXTest
```

Figure 119 (Part 2 of 2). Extended HTTP Client Using URLs: UrlXTest.nrx

The skeleton of this program is much the same as the first URL example (Figure 118 on page 215). Here, using the <= operator, we introduce a select clause that examines the type of the content.

Of course, this example only works with audio clips when your sound support is installed correctly. In addition, depending on your Java run time (the installed ContentHandlerFactory), some types are not recognized. In this case you just receive the document as an InputStream and the program prints the content to the screen. You can install a new StreamHandlerFactory to extend the default one with unrecognized types:

```
URLConnection.setURLStreamHandlerFactory(...)
```

This only works if you are permitted to do so by the security manager. We do not discuss this issue further, as it would lead us to a detailed discussion of the Java run time and its security features, which are not within the scope of this book.

The multiplicity of content types and the fact that everybody can create new protocols and content types do not make the life of a programmer using URLs easy. Java therefore allows a more general solution to this problem. ProtocolHandler and ContentHandler classes can be loaded dynamically into the Java virtual machine (JVM). Such classes can be located on the same server where the pages reside that use irregular protocols or content types. When a client attempts to load a resource of an unknown content type, or a URL with an unknown

protocol, it can ask the server to serialize the class and send it. Using this technique, the client can learn new protocols or content types.

The HotJava browser includes this kind of dynamic loading. Let us hope that the JDK will be enhanced so that the getContent method can search for such ProtocolHandlers and ContentHandlers at a remote host. For a more detailed discussion of downloadable ProtocolHandlers and ContentHandlers, refer to *Exploring Java*.

Testing the Extended URL Client Program

This is the output of the program when given a URL of a .GIF file:

```
d:\NrxRedBk\network\url>java UrlXTest http://hostname/some.gif
Connect to  : http://hostname/some.gif
Content-type: image/gif
Class       : sun.awt.image.URLImageSource
==> Content is an ImageProducer
```

Typical Network Exception Types

The exception mechanism (see “Exceptions” on page 30) enables you to react to the various exceptional situations that can occur when making and using connections to other computers on the Internet. Comparing Internet programs that are written in traditional languages, such as C, with functionally identical programs written in NetRexx (Java) immediately reveals the advantages of using exceptions.

When you are interacting with a network, nearly everything can go wrong, at any time. This implies that—in traditional languages—you have to check for every possible error when using the network. With exceptions, however, you can defer the handling of these situations at an appropriate level. Exceptions also allow a clear separation of the code that handles these network problems and the main logic of your methods.

All of the exceptions that can be thrown out of the java.net package are subclasses of IOException (part of the java.io package).

The exceptions are:

- MalformedURLException
- ProtocolException
- SocketException
- UnknownHostException
- UnknownServiceException

The SocketException itself is refined in these subclasses:

- BindException
- ConnectException
- NoRouteToHostException

In Figure 120 we provide you with sample source code that you can include in your own networking applications, before you refine them to more useful versions.

```
/* network\url\NetExcpt.nrx

   Skeleton code for exception handling in network programs */

catch e1=MalformedURLException
  say 'The port is not available (connection refused):'
  say '- port is not registered on the server'
  say '- the server is down'
  say 'MalformedURLException (' e1 ') caught: \n' e1.getMessage()

catch e2=ProtocolException
  say 'An error occured in the level of the used protocol (FTP, HTTP,...)'
  say 'ex.: Error in the format of the HTTP request string (GET /...).'
  say 'ProtocolException (' e2 ') caught: \n' e2.getMessage()

catch e3=BindException
  say 'The binding of a socket to a local address and port failed.'
  say 'This occurs when you want to register code on'
  say '(one of the IP addresses) on your server:'
  say '- port is not avialable'
  say '- the specified IP address was not found on this server'
  say 'BindException (' e3 ') caught: \n' e3.getMessage()

catch e4=ConnectException
  say 'The port is not available (connection refused):'
  say '- port is not registered on the server'
  say '- the server is down'
  say 'IOException (' e4 ') caught: \n' e4.getMessage()

catch e5=NoRouteToHostException
  say 'The attempt to connect to the host timed out.'
  say '- host is unreachable'
  say '- host is too far'
  say 'NoRouteToHostException (' e5 ') caught: \n' e5.getMessage()

catch e6=SocketException
  say 'A socket creation failed or an attempt was made'
  say 'to register a socketfactory when one has already been set.'
  say 'Older versions of classes can throw this instead a the recent,'
  say 'more specialized versions.'
  say 'SocketException (' e6 ') caught: \n' e6.getMessage()

catch e7=UnknownHostException
  say 'A hostname (' e7.getMessage ')'
  say 'could not be resolved to its IP address.'
  say 'Try to use the IP (numerical) address instead.'
  say 'UnknownHostException (' e7 ') caught: \n' e7.getMessage()

catch e8=UnknownServiceException
  say 'There is something wrong with an URLConnection, either:'
  say '- the MIME type returned by a URL connection does not make sense'
  say '- you tried to write to a read-only connection'
  say '- you tried to read a connection that does not support inputstreams'
  say 'IOException (' e8 ') caught: \n' e8.getMessage()
```

Figure 120. Exception Handling Code for Networking Programs

The exceptions that are caught provide a brief explanation of the possible causes in their respective catch clause.

You should comment out the exceptions that cannot be signaled by your actual code; they are reported when you run the NetRexx compiler. Include the full catch code the first time in your application to ensure that you will not forget implementing the handling of all potential exceptions.

The exceptions that can be thrown are listed in Table 11, together with the methods that explicitly declare the respective exceptions in their throw list. Note, however, that some methods only declare one of the superclasses of the exceptions.

Table 11. Table of Exceptions Thrown in java.net

	Explicitly Thrown by
MalformedURLException	• URL constructors • RMIClassLoader.loadClass() • LoaderHandler.loadClass()
ProtocolException	• HttpURLConnection.setRequestMethod()
SocketException (BindException, ConnectException*, NoRouteToHostException)	• ServerSocket.setSoTimeout(), setSocketFactory() • Socket constructor, setSocketFactory(), setSoTimeout() • SocketOutputStream.write() • DatagramSocket constructors, create(), getSoTimeout(), setSoTimeout() • MulticastSocket.create(), setInterface(), getInterface() • RMISocketFactor.setSocketFactory()
UnknownHostException*	• InetAddress constructor, getByName(), getAllByName(), getLocalHost() • Socket constructor • (URL, implicitly)
UnknownServiceException	• URL constructors, getInputStream(), getOutputStream()
(*) The ConnectException and the UnknownHostException are also defined in the *java.rmi* package.	

The subclasses of SocketException (BindException, ConnectException, and NoRouteToHostException) are not yet explicitly declared in any method of JDK 1.1.1.

Remote Method Invocation

Similar to the Socket classes that encapsulate the traditional socket library, we now have RMI, the object-oriented counterpart of the remote procedure call (RPC) mechanism. The RMI classes are contained in separate packages:

```
java.rmi
java.rmi.server
```

Remote Procedure Call

The RPC mechanism offers a more hassle-free mechanism to support client/server applications than the I/O-streams provided by the sockets. This protocol lets you call remote functions from within your code like normal function calls. These remote function calls are in fact calls to a client stub that runs on the local machine and handles all the communication-related activities transparently. This client stub has its counterpart on the server side (server stub) that handles the socket protocols, such as listen and accept, also transparent to the programmer.

The RPC model is built on top of the socket implementations, thus keeping a clear layered structure.

RMI

The RMI is based on the same layered structure as RPC. Instead of providing a way to remotely call functions, however, we now speak object-oriented; we send messages to remote objects, that is, we *invoke* remote methods. Under the cover we still rely on the UNIX sockets; at least with their object-oriented counterparts, the Socket classes.

To pass arguments and return values, RMI uses *marshalling streams*, input and output streams that are generated by the code that implements the RMI communication. This code is in fact divided into two layers:

- Remote reference layer
- Transport layer

The client stubs and the server skeletons use these layers to pass the message arguments to the remote method and the return values to the caller. They serialize (see "Object-Oriented I/O Using Serialization" on page 176) local objects (by copy) and the references to remote objects (by reference) to the marshaling streams.

More information about the remote reference layer and the transport layer can be found in the documentation that comes with the JDK. We do not go any deeper into this subject here, because you only have to understand the abstract concepts for your first steps in using the RMI package.

Server-locating facilities and error handling and recovery are also strictly defined in the RMI model.

RMI Registry and URLs

To enable RMI, a registration service program called the *registry* must be running on the server:

```
d:\NrxRedBk\network\rmi>start rmiregistry.exe          <=== Windows
                        start rmiregst.exe             <=== OS/2
```

Note: The registry process must have access to the stub classes produced by the RMI compiler (see "RMI Compiler" on page 225).

A client can locate RMI servers by asking the registry to return a reference to the remote object described by a URL that is passed as an argument to the method:

```
Naming.lookup(urlstring)
```

The Naming class is the bootstrap mechanism for obtaining references to remote objects based on the URL syntax. You specify the URL for a remote object, using the usual host, port, and name:

```
rmi://<host>:<port>/<object name>
```

where rmi is the protocol. The other three parameters are:

host Host name of the registry (defaults to current host)

port Port number of the registry (defaults to 1099)

object name Name of the remote object, as registered by the server

The first three fields are optional, so this simple call would be enough to connect to the remote object *MyFirstRMIServer*, which happens to reside on the same machine:

```
rmi:///MyFirstRMIServer
```

You register a server object, using the bind or rebind method of the Naming class:

```
Naming.bind(urlstring,serverobject)
```

Error notification and handling are accomplished through the basic exception throwing and catching mechanisms. All remote methods, and client methods invoking remote methods, can throw *RemoteException* objects (or more specialized subclasses).

RMI Listener Example

When programming with distributed objects, we advise you to first create a working implementation of the application without distribution. Because the RMI model allows you to *split* the object model with only minor code changes, it eliminates much of the confusion in resolving errors.

Our first small example implements an RMI server that listens to clients and displays (using *say*) the text sent.

RMI Client

We begin by writing a very simple RMI construction that passes a string from a local object to a remote object, which we call *Listener*.

Figure 121 shows the client program that sends user-entered text to the server object. We start with the client part to convince you of the simplicity of the changes to your code.

```
/* network\rmi\RmiClnt.nrx

   Client RMI program, sending user text to the server object:
   Usage: Java RmiClnt <serverhostname> */

package network.rmi

import java.rmi.

parse arg serverAdr .                          -- server host name
urlstring = 'rmi://'serverAdr'/Listener'
if serverAdr = '' then serverAdr = '(local)'

do
  say 'Registering the Security Manager ...'
  System.setSecurityManager(RMISecurityManager())
  say 'Looking up the Listener on' serverAdr '... Please Wait !'
  listener = RmiSrvrI Naming.lookup(urlstring)   -- server object
  loop forever
     say 'What should I say to the server ? (or type: end)'
     smalltalk = ask
     listener.listen(smalltalk)                  -- RMI call to server
     say 'Your input has been sent to the server.\n'
     if smalltalk = 'end' then leave
```

Figure 121 (Part 1 of 2). RMI Client Program: RmiClnt.nrx

```
  end
catch e1=RemoteException
  say 'Something is wrong with the RMI connection!'
  say 'RemoteException caught: \n' e1.getMessage()
catch e2=java.net.MalformedURLException
  say 'The URL is not valid:' urlstring
  say 'MalformedURLException caught: \n' e2.getMessage()
catch e3=Exception
  say 'Exception (' e3 ') caught: \n' e3.getMessage()
end
say 'End of RMI Client'
-- end RmiClnt
```

Figure 121 (Part 2 of 2). RMI Client Program: RmiClnt.nrx

The argument you supply when running this sample client program is the host name where the remote object can be found.

There are only two special lines of code in this program:

- Looking up the listener object on the server:

  ```
  listener = RmiSrvrI Naming.lookup('rmi://'serverAdr'/Listener')
  ```

  This call looks up the server object that implements the *Listener*; it is of the RmiSrvI class (see Figure 122 on page 224). As mentioned before, the lookup method returns a reference to the remote object located by the URL. The Naming class is part of the java.rmi package.

- Invoking the *listen* method of the server object, using RMI:

  ```
  listener.listen(userstring)
  ```

The rest of the program is a loop asking the user for input that is passed to the server object. An input of *end* ends the client program.

If you run the program without the registry running on the server you get:

```
d:\NrxRedBk\network\rmi>java RmiClnt
Looking up the Listener on  ... Please Wait !
Something is wrong with the RMI connection!
RemoteException caught:
 Connection refused to host; nested exception is:
        java.net.ConnectException: Connection refused
```

This message is returned because there is no RMI registry registered on the port. We explain how to set up the registry when we discuss the coding on the server side.

If the registry is running, but the server is not running, you get:

```
d:\NrxRedBk\network\rmi>java RmiClnt fundy
Looking up the Listener on fundy ... Please Wait !
Exception ( java.rmi.NotBoundException: Listener )caught:
 Listener
```

RMI Server Interface

The server part of our RMI construction is a bit more complicated. You have to know that for every object you want to *publish* as a remote object, you need an interface describing the services of its class. In this way you (as a user or client of the object) are totally independent of the implementation of the class. Only the methods defined in the interface are available to be called from the outside.

Because we only provide one method, called *listen*, to the clients, the interface definition is very short (see Figure 122).

```
/* network\rmi\RmiSrvrI.nrx

   RMI server interface definition */

package network.rmi

import java.rmi.

class RmiSrvrI interface implements Remote

method listen(str = String) public signals RemoteException
```

Figure 122. RMI Server Interface: RmiSrvrI.nrx

To register a server as a remote object, an RMI interface class must implement (*extend* in Java terms) the interface:

```
java.rmi.Remote
```

The remote methods must be declared as public and signal the RemoteException.[3]

RMI Server Implementation

Figure 123 shows the sample server program that implements the RmiSrvrI interface.

```
/* network\rmi\RmiSrvr.nrx

   Server RMI program, listening to clients sending user text.
   Implements the "listen" method */

package network.rmi

import java.rmi.
import java.rmi.server.

/*-------------------------- class RmiSrvr -----------------------------*/

class RmiSrvr public extends UnicastRemoteObject implements RmiSrvrI

method main(args=String[]) public static
  do
    hostname = ''
    if args.length = 1 then hostname = args[0]
```

Figure 123 (Part 1 of 2). RMI Server Implementation: RmiSrvr.nrx

[3] Currently the NetRexx compiler warns you that this RemoteException is not thrown in the method when compiling the implementation of the server. Ignore this message:

```
Warning: Checked exception 'java.rmi.RemoteException' is in SIGNALS list but is not signalled within the method
```

```
      say 'Registering the Security Manager ...'
      System.setSecurityManager(RMISecurityManager())
      say 'Publishing the "Listener" object: rmi://'hostname'/Listener'
      listener = RmiSrvr()                                  -- server object
      Naming.rebind('rmi://'hostname'/Listener',listener) -- bind Listener
      say 'I am listening ...'
    catch e=Exception
      say 'Exception (' e ') caught: \n' e.getMessage()
    end

  method RmiSrvr() signals RemoteException            -- constructor

  ------------------------------------------------------ public interface
  method listen(str = String) public signals RemoteException
    do
      say 'Client' getClientHost() 'says:\n ' str
    catch ServerNotActiveException
      say 'Only a client can invoke listen'
    end

  -- end RmiSrvr
```

Figure 123 (Part 2 of 2). RMI Server Implementation: RmiSrvr.nrx

The server class must extend a class derived from—the abstract class—*Remote Server*. For our class (*RmiSrvr*) we extend the *UnicastRemoteObject* class. This class implements an RMI server that accepts many clients but does not replicate itself.

The first thing the server program does is registering a security manager, so that the RMI mechanism can load classes. Without a security manager that checks classes for their *good behavior*, you cannot remotely load a new class into the JVM.

After creating an object of our class, we proceed with *publishing* our remote object in the RMI registry. We use one of these methods:

```
Naming.bind(String, Remote)
Naming.rebind(String, Remote)
```

We used rebind to make it possible to register new versions without having the problem of unbinding them first.

From now on your object is prepared to accept client messages that perform its remote listen method.

We explicitly declare the empty constructor to focus on the fact that remote objects can throw a *RemoteException* when they are constructed.

The implementation of listen is rather trivial; feel free to change it to something more original.

RMI Compiler

All remote objects have to be compiled again—after the NetRexx compile—by the RMI compiler (*rmic*) to generate the stub and the skeleton classes for the remote objects. The recompile uses the class file as input and is done with the following call:

```
d:\NrxRedBk\network\rmi>rmic RmiSrvr
```

The RMI compiler creates two classes, *RmiSrvr_Stub* and *RmiSrvr_Skel*; be sure to copy the classes into a directory of the CLASSPATH because the registry process must be able to find them.

Testing the RMI Listener

To test the server with at least one client, be sure to start the registry, before starting the server:

```
d:\NrxRedBk\network\rmi>start rmiregistry.exe
d:\NrxRedBk\network\rmi>java RmiSrvr
```

Now start one or multiple clients, on the same or different machines, and enter text to be displayed in the server window:

```
d:\NrxRedBk\network\rmi>java RmiClnt serverhostname
Looking up the Listener on serverhostname ... please Wait!
What should I say to the server ?
To Sell: 1 dog, only 5 years old   Tel: 5896   <=== client input
Your input has been sent to the sever          <=== response
What should I say to the server ?
...
```

You can use this construction to have a rudimentary advertising board to which everyone can write. Figure 124 shows the sample server output.

```
d:\NrxRedBk\network\rmi>java RmiSrvr
Registering the Security Manager ...
Publishing the "Listener" object: rmi:///Listener
I am listening. ...
Client SALLY says:
  To Sell: 1 dog, only 5 years old   Tel: 5896
Client HARRY says:
  Seeking an Audi 100.  Tel: 2761
Client DOLE says:
  Lost my wallet, finder will be rewarded with 5% of the cash - Email: me@ibm.com
Client PAT says:
  Selling a mountain bike for $100, in good condition. Juoko, Tel: 3666
Client BIGSAVER says :
  Coffee machine broken, free coffee in E2 ...
```

Figure 124. RMI Listener Sample Output

Close the client by typing *end*, and close the server and the registry with Ctrl-C.

Running RMI on a Single Machine

RMI works well on a single machine that is connected through TCP/IP to a network (LAN or Internet). However, if you are not connected to a network, make sure that:

- TCP/IP is configured
- Loopback is configured (ping localhost gets a reply from address 127.0.0.1)
- The TCP/IP properties (Control Panel, Network) specify a DNS configuration with a host name of localhost and no domain

You should be able to get RMI working with a localhost configuration.

RMI Parameters and Return Values

The parameter objects that are passed to a remote method must be serializable. The same is true for the return value of a method.

The NetRexx Rexx class of NetRexx 1.0 is not serializable; therefore be sure not to use Rexx strings as arguments and return values. The Rexx class of NetRexx 1.1 is serializable.

RMI Chat Application

The listener client/server example is a simple RMI application where the client does not receive any feedback from the server about messages posted by other clients.

We also implemented a more complete chat application where the server sends every message received from a client to all connected clients. In this RMI application every client is also an RMI server, to receive the asynchronous client messages sent from the server.

We do not describe this example in detail. Here are the components of the application:

ChatRMIServerI The public interface of the chat server consists of two methods: newClient to register a new chat client, and sendMessage, to accept a message from a client.

ChatRMIServer Chat server that implements the ChatRMIServerI interface

ChatRMIClientI The public interface of the chat client consists of one method, addMessage, to receive a message from the chat server.

ChatClient The chat client applet contains the ChatRMIClient class that implements the ChatRMIClientI interface.

Chat.htm HTML file to run the chat client applet

The code of the RMI chat application is stored in the network\rmichat directory.

Wrapping Up with a Complete RMI Program

We use RMI to split up the program presented in “Wrapping Up with a Complete JDBC GUI Program” on page 196.

The GUI part runs in the virtual machine of the client. The controller object that handles all database access runs on an RMI server machine, which in turn may connect to a real server where the DB2 database system runs.

We can build a three-tiered model, using RMI, with the advantage that the DB2 native drivers needed for JDBC are only on the middle-tier server.

We take the existing JDBC program (Figure 113 on page 197) and split the code into two programs; one with the Controller class (RmiCont), and one with the GUI classes (RmiGui, DoAction, ControlButtons).

The Controller class is modified to be a remote class, and the GUI class must be able to use the Controller. We name the classes RmiCont and RmiGui, and the interface of the controller, RmiContI.

Controller Interface

The controller class exports the following methods:

```
connect()
retrieveListDep()
select(depNo)
update(depNo, name, mgNo, mgDepNo)
delete(depNo)
add(depNo, name, mgNo, mgDepNo)
getDepName()
getMgNo()
getMgDepNo()
```

Figure 125 shows the controller interface class.

```
/* network\rmijdbc\RmiContI.nrx

   RMI - JDBC Program:  JDBC Controller Interface */

package network.rmijdbc

import java.rmi.

class RmiContI interface implements Remote

method connect()             returns String   public signals RemoteException
method retrieveListDep()     returns String[] public signals RemoteException
method select(depNo=String)  returns String   public signals RemoteException
method delete(depNo=String)  returns String   public signals RemoteException
method add(depNo=String, name=String, mgNo=String, mgDepNo=String) -
                             returns String   public signals RemoteException
method update(depNo=String, name=String, mgNo=String, mgDepNo=String) -
                             returns String   public signals RemoteException
method getDepName()          returns String   public signals RemoteException
method getMgNo()             returns String   public signals RemoteException
method getMgDepNo()          returns String   public signals RemoteException
```

Figure 125. RMI JDBC Application Controller Interface: RmiContI.nrx

RMI JDBC Controller Server

We make the controller class a RemoteServer. Because a point to point implementation of the RemoteServer is not available (yet) in the JDK package, we use the UnicastRemoteObject.

The controller registers itself with the RMI registry. The exported methods are made public, with the proper exception, but otherwise the only change to the code is in the retrieveListDep method that returns a Java string array instead of a Rexx string (see Figure 126).

```
/* network\rmijdbc\RmiCont.nrx

   RMI - JDBC Program:  JDBC Controller Implementation
   Usage: Java RmiCont [<DB-URL>] [<userprefix>] */

package network.rmijdbc

import java.rmi.
import java.rmi.server.

import java.sql.

class RmiCont public extends UniCastRemoteObject implements RmiContI

/*-------------------------- variables -------------------------------*/
Properties static
  prefix      = Rexx 'userid'          -- table prefix
  url         = Rexx 'jdbc:db2:sample' -- database URL
  conthost    = Rexx ''                -- host name of controller

Properties inheritable
  jdbcCon     = Connection             -- database Connection
  selDepNo    = String ''              -- selected row values
  selName     = String ''
  selMgNo     = String ''
  selMgDepNo  = String ''

/*-------------------------- methods ---------------------------------*/
method main(args=String[]) public static
  do
    if args.length > 0 then url      = args[0]
    if args.length > 1 then prefix   = args[1]
    say 'Registering the Security Manager ...'
    System.setSecurityManager(RMISecurityManager())
    say 'Publishing the "Controller" object: rmi://'conthost'/Controller'
    controller = RmiCont()                                  -- controller object
    Naming.rebind('rmi://'conthost'/Controller',controller) -- bind controller
    say 'RMI Controller is ready ...'
  catch e=Exception
    say 'Exception (' e ') caught: \n' e.getMessage()
  end

/*-------------------------- constructor -----------------------------*/
method RmiCont() signals RemoteException

/*-------------------------- connect() -------------------------------*/
method connect() returns String public signals RemoteException
  do
    Class.forName('COM.ibm.db2.jdbc.app.DB2Driver').newInstance()
    -- Class.forName('COM.ibm.db2.jdbc.net.DB2Driver').newInstance()
    say 'Connecting to database:' url
    jdbcCon = Connection DriverManager.getConnection(url, 'userid', 'password')
    msg = 'Connected to' url
    say msg
  catch e2 = SQLException
    msg = 'SQLException(s) caught while connecting !'
    loop while (e2 \= null)
       say 'SQLState:' e2.getSQLState()
       say 'Message: ' e2.getMessage()
       say 'Vendor:  ' e2.getErrorCode()
       say
       e2 = e2.getNextException()
    end
  catch e1 = Exception
    msg = 'The DB2 driver classes could not be found and loaded !'
    say 'Exception (' e1 ') caught : \n' e1.getMessage()
  end
  return msg

/*-------------------------- retrieveListDep() -----------------------*/
method retrieveListDep() returns String[] public signals RemoteException
  deptarr = Rexx ''
  do
    query = 'SELECT deptno, deptname, mgrno, admrdept' -
            'FROM' prefix'.DEPARTMENT ORDER BY deptno'
    stmt = Statement jdbcCon.createStatement()
```

Figure 126 (Part 1 of 3). RMI JDBC Controller Server: RmiCont.nrx

```
      rs = ResultSet stmt.executeQuery(query)
      loop row=1 while rs.next()
           deptarr[row] = rs.getString('deptno') '-' rs.getString('deptname')
      end
      row = row-1
      deptarr['count'] = row
      rs.close()                          -- close the result set
      stmt.close()                        -- close the statement
      deptarr['message'] = 'Retrieved' row 'departments'
    catch ex=SQLException
      deptarr['message'] = ex.getMessage()
      deptarr['count'] = 0
    end
    deptstr = String[row+1]               -- convert Rexx to: string[]
    loop i=1 to row
       deptstr[i] = deptarr[i]
    end
    deptstr[0] = deptarr['message']
    return deptstr                        -- return the string array

  /*-------------------------- update() ---------------------------------*/
  method update(depNo=String, name=String, mgNo=String, mgDepNo=String) -
         returns String public signals RemoteException
    do
      if depNo = '' then
           return 'Missing department number for update !'
      if depNo <> selDepNo then           -- If depNo already exists you'll
        do                                -- loose the department at selDepNo
           tempDepNo = selDepNo           -- Room for improvement...
           add(depNo, name, mgNo, mgDepNo)
           delete(tempDepNo)
           return 'Department number updated (delete old, add new)'
        end
      updQuery = 'UPDATE' prefix'.DEPARTMENT' -
                 "SET deptname = '"name"'," -
                     "mgrno = '"mgNo"'," -
                     "admrdept = '"mgDepNo"'" -
                 "WHERE deptno = '"depNo"'"
      say 'Update SQL query :' updQuery
      stmt = Statement jdbcCon.createStatement()
      stmt.executeUpdate(updQuery)
      msg = 'Replaced a department.'
      stmt.close()
      selDepNo = ''
    catch ex=SQLException
      msg = ex.getMessage()
    end
    return msg

  /*-------------------------- delete() -------------------------------*/
  method delete(depNo=String) returns String public signals RemoteException
    do
      if depNo = '' then
         return 'Missing department number for delete'
      delQuery = 'DELETE FROM' prefix'.DEPARTMENT' -
                 "WHERE deptno = '"depNo"'"
      say 'Delete SQL query :' delQuery
      stmt = Statement jdbcCon.createStatement()
      stmt.executeUpdate(delQuery)
      msg = 'Deleted a department.'
      stmt.close()
      selDepNo = ''
    catch ex=SQLException
      msg = ex.getMessage()
    end
    return msg

  /*-------------------------- add() ---------------------------------*/
  method add(depNo=String, name=String, mgNo=String, mgDepNo=String) -
         returns String public signals RemoteException
    do
      query = 'SELECT deptno, deptname, mgrno, admrdept' - --test uniqueness of depNo
              'FROM' prefix'.DEPARTMENT' -
              "WHERE deptno = '"depNo"'"
      stmt = Statement jdbcCon.createStatement()
      rs = ResultSet stmt.executeQuery(query)
```

Figure 126 (Part 2 of 3). RMI JDBC Controller Server: RmiCont.nrx

```
      if rs.next() then
         return ''depNo ' is in use, choose another department number.'
      rs.close()                          -- Close the result set
      stmt.close()                        -- Close the statement
      addQuery = 'INSERT INTO' prefix'.DEPARTMENT' -
                 '(deptno, deptname, mgrno, admrdept)' -
                 "VALUES ('"depNo"','"name"','"||-
                 mgNo"','"mgDepNo"')"
      say 'Insert SQL query :' addQuery
      stmt = Statement jdbcCon.createStatement()
      stmt.executeUpdate(addQuery)
      msg = 'Inserted a department.'
      stmt.close()
    catch ex=SQLException
      msg = ex.getMessage()
    end
    return msg

  /*-------------------------- select() -------------------------------*/
  method select(depNo=String) returns String public signals RemoteException
    if (selDepNo <> depNo) then
      do
        msg = 'Select('depNo')'
        query = 'SELECT deptno, deptname, mgrno, admrdept' -
                'FROM' prefix'.DEPARTMENT' -
                "WHERE deptno = '"depNo"'"
        stmt = Statement jdbcCon.createStatement()
        rs = ResultSet stmt.executeQuery(query)
        rs.next()
        selDepNo   = depNo
        selName    = rs.getString('deptname')
        selmgNo    = rs.getString('mgrno')
        selmgDepNo = rs.getString('admrdept')
        rs.close()                         -- close the result set
        stmt.close()                       -- close the statement
      catch ex=SQLException
        msg = 'Select():' ex.getMessage()
      end
    else msg = 'Selected department number did not change'
    return msg

  /*-------------------------- getDepName() -------------------------------*/
  method getDepName() returns String public signals RemoteException
    return selName

  /*-------------------------- getMgNo() ----------------------------------*/
  method getMgNo() returns String public signals RemoteException
    return selMgNo

  /*-------------------------- getMgDepNo() -------------------------------*/
  method getMgDepNo() returns String public signals RemoteException
    return selMgDepNo

  -- end RmiCont
```

Figure 126 (Part 3 of 3). RMI JDBC Controller Server: RmiCont.nrx

RMI JDBC GUI Client

We change the GUI client to run as an applet instead of an application. Instead of a main method, we use the init method to initialize the applet. The applet looks up the RMI controller object on the host from where the applet's code came.

The GUI code is basically unchanged, with only a few lines commented out because they do not apply for an applet (see Figure 127).

```
/* network\rmijdbc\RmiGui.nrx

   RMI - JDBC Applet:  GUI Implementation */

package network.rmijdbc

import java.rmi.

import Redbook.

class RmiGui public extends Applet uses GridBagConstraints

  Properties constant
    UPDATE   = int 0
    ADD      = int 1
    DELETE   = int 2
    RETRIEVE = int 3

  Properties static
    ctr           = RmiContI               -- RMI controller
    conthost      = Rexx ''                -- host name of controller

  Properties inheritable
    tf_depno      = TextField(3)
    tf_depname    = TextField(29)
    tf_manager    = TextField(6)
    tf_mgrDept    = TextField(3)
    l_departments = List(10)
    ta_message    = TextArea(2,30)
    b1            = Button
    b2            = Button
    b3            = Button
    b4            = Button
    b5            = Button

/*------------------------- methods --------------------------------*/
  method init()
    do
      win = this                            -- applet frame
      showStatus('RMI JDBC GUI')            -- status message
      gbl = SimpleGridbagLayout(win)        -- use gridbag layout
      -- ws = WindowSupport(win)               -- close window support

      buildLayout(gbl)                      -- ,ws)

      -- win.pack()
      -- RedbookUtil.positionWindow(win)
      -- win.setVisible(1)
      repaint()

      setMessage('Registering the Security Manager ...')
      do
         System.setSecurityManager(RMISecurityManager())
       catch SecurityException
      end

      conthost = getCodeBase().getHost()    -- controller host
      setMessage('Finding the RMI Controller on:' conthost)
      ctr = RmiContI Naming.lookup('rmi://'conthost'/Controller')
      setMessage('RMI is ready...')

      b1.addActionListener(DoAction(this,ctr,UPDATE))
      b2.addActionListener(DoAction(this,ctr,ADD))
      b3.addActionListener(DoAction(this,ctr,DELETE))
      b4.addActionListener(DoAction(this,ctr,RETRIEVE))
      l_departments.addItemListener(DoAction(this,ctr))

      setMessage('Connecting to database...')
      msg = ctr.connect()
      setMessage(msg)

      setMessage('Retrieving departments...')
      setDepartmentsList(ctr.retrieveListDep())
    catch e1=RemoteException
      say 'RMI RemoteException caught in init: \n' e1.getMessage()
    catch e=Exception
```

Figure 127 (Part 1 of 4). RMI JDBC GUI Client: RmiGui.nrx

```
        setMessage('Exception (' e ') caught: \n' e.getMessage())
      end

    method buildLayout(gbl = SimpleGridbagLayout)  -- , ws=WindowSupport)
      gbl.addFixSize(Label('Number'),  0,0,Insets(10,10,5,30))
      gbl.addFixSize(Label('Name'),    0,1,Insets( 5,10,5,30))
      gbl.addFixSize(Label('Manager'), 0,2,Insets( 5,10,5,30))
      gbl.addFixSize(Label('Mgr Dept'),0,3,Insets( 5,10,5,30))

      gbl.addVarSize(tf_depno  ,1,0,Insets(10,0,5,5),1.0,0.0)
      gbl.addVarSize(tf_depname,1,1,Insets( 0,0,5,5),1.0,0.0)
      gbl.addVarSize(tf_manager,1,2,Insets( 0,0,5,5),1.0,0.0)
      gbl.addVarSize(tf_mgrDept,1,3,Insets( 0,0,5,5),1.0,0.0)
      tf_depno.addTextListener(LimitTextField(tf_depno,3))
      tf_depname.addTextListener(LimitTextField(tf_depname,29))
      tf_manager.addTextListener(LimitTextField(tf_manager,6))
      tf_mgrDept.addTextListener(LimitTextField(tf_mgrDept,3))

      gbl.addVarSize(l_departments,0,4,Insets(10,10,15,5),1.0,2.0,2)
      gbl.addVarSize(ta_message,   0,5,Insets( 0,10,10,5),1.0,1.0,2)

      b1=Button gbl.addFixSize(Button('Update')  ,2,0,Insets(10,0, 5,10),1,1,HORIZONTAL)
      b2=Button gbl.addFixSize(Button('Add')     ,2,1,Insets( 0,0, 5,10),1,1,HORIZONTAL)
      b3=Button gbl.addFixSize(Button('Delete')  ,2,2,Insets( 0,0, 5,10),1,1,HORIZONTAL)
      b4=Button gbl.addFixSize(Button('Retrieve'),2,3,Insets( 0,0, 5,10),1,1,HORIZONTAL)
      -- b5=Button gbl.addFixSize(Button('Exit')    ,2,5,Insets( 0,0,10,10),1,1,HORIZONTAL,SOUTHWEST)
      -- b5.addActionListener(ws.getCloseWindow())

      cb = ControlButtons()
      cb.addButton(b1)
      cb.addButton(b2)
      cb.addButton(b3)
      tf_depno.addTextListener(cb)
      cb.disable

    method setDepartmentsList(aList = String[])
      l_departments.removeAll()
      loop i = 1 for aList.length - 1
        l_departments.add(aList[i])
      end
      setMessage(aList[0])

    method setDepno(s=String)
      tf_depno.setText(s)

    method setName(s=String)
      tf_depname.setText(s)

    method setMgNo(s=String)
      tf_manager.setText(s)

    method setMgDepNo(s=String)
      tf_mgrDept.setText(s)

    method getDepno() returns String
      return tf_depno.getText()

    method getName() returns String
      return tf_depname.getText()

    method getMgNo() returns String
      if tf_manager.getText() = '' then return null
                                   else return tf_manager.getText()

    method getMgDepNo() returns String
      return tf_mgrDept.getText()

    /*---------------------------- interface to display messages -----------------*/
    method setMessage(msg = String)
      say msg
      ta_message.append('\n'msg)

  /******************************** class DoAction ***************************/

  class DoAction implements ActionListener, ItemListener uses RmiGui
    Properties inheritable
```

Figure 127 (Part 2 of 4). RMI JDBC GUI Client: RmiGui.nrx

```
    ui  = RmiGui      -- reference to user interface
    ctr = RmiContI    -- reference to the controller
    act = int         -- which action

  method DoAction(aGui=RmiGui,aControl=RmiContI,what=int 0)
    ui  = aGui
    ctr = aControl
    act = what

  method actionPerformed(e=ActionEvent)
    select
      when act = UPDATE then do
        ui.setMessage('Updating a row in the DEPARTMENT table...')
        msg = ctr.update(ui.getDepno(),ui.getName(),ui.getMgNo(),ui.getMgDepNo())
        ui.setMessage(msg)
        ui.setDepartmentsList(ctr.retrieveListDep())
      end
      when act = ADD then do
        ui.setMessage('Inserting a row in the DEPARTMENT table...')
        msg = ctr.add(ui.getDepno(),ui.getName(),ui.getMgNo(),ui.getMgDepNo())
        ui.setMessage(msg)
        ui.setDepartmentsList(ctr.retrieveListDep())
      end
      when act = DELETE then do
        ui.setMessage('Deleting a row in the DEPARTMENT table...')
        msg = ctr.delete(ui.getDepno())
        ui.setMessage(msg)
        ui.setDepartmentsList(ctr.retrieveListDep())
      end
      when act = RETRIEVE then do
        ui.setDepartmentsList(ctr.retrieveListDep())
        ui.setDepno('')     -- clear the fields
        ui.setName('')
        ui.setMgNo('')
        ui.setMgDepNo('')
      end
    catch e1=RemoteException
      say 'RMI RemoteException caught in action' act': \n' e1.getMessage()
    end

  method itemStateChanged(e=ItemEvent)
    s = (List e.getItemSelectable()).getSelectedItem()
    if s \= null then
      do
        parse s depno '-' .
        msg = ctr.select(String depno.strip())

        ui.setMessage(msg)
        ui.setDepno(depno)             -- set the fields
        ui.setName(ctr.getDepName())
        ui.setMgNo(ctr.getMgNo())
        ui.setMgDepNo(ctr.getMgDepNo())
      catch e1=RemoteException
        say 'RMI RemoteException caught in itemChanged: \n' e1.getMessage()
      end

/********************************* class ControlButtons *********************/

class ControlButtons implements TextListener
 Properties inheritable
   buttons = Vector()
   enabled = boolean 1

 method addButton(aButton = Button)
   buttons.addElement(aButton)

 method textValueChanged(e = TextEvent)
   field = TextField e.getSource()
   text = field.getText()
   if text \= null then
     if text = '' then disable()
                  else enable()
   else disable()

 method enable()
   if enabled then return
```

Figure 127 (Part 3 of 4). RMI JDBC GUI Client: RmiGui.nrx

```
    enabled = 1
    change()

  method disable()
    if \enabled then return
    enabled = 0
    change()

  method change()
    loop i=0 for buttons.size
      (Button buttons.elementAt(i)).setEnabled(enabled)
    end

-- end RmiGui
```

Figure 127 (Part 4 of 4). RMI JDBC GUI Client: RmiGui.nrx

Testing the RMI JDBC Applet

To test the applet you have to prepare a server machine with the code of the server and the applet:

- Make sure that the stub classes generated by the RMI compiler are in the CLASSPATH.
- Start the RMI registry program on the server:

  ```
  d:\NrxRedBk\network\rmijdbc>start rmiregistry.exe
  ```

- Start the controller server with defaults or with the user ID that created the sample database:

  ```
  d:\NrxRedBk\network\rmi>java RmiCont
                          java RmiCont jdbc:db2:sample userid
  ```

- Make sure that DB2 is up and running.
- Start the JDBC daemon (db2strt 8888) if JDBC access is through the network.
- Display the applet's HTML page (RmiGui.htm) and run the distributed RMI JDBC applet, using a browser or the appletviewer of the JDK:

  ```
  d:\NrxRedBk\network\rmi>appletviewer RmiGui.htm
  ```

  The applet should look identical to Figure 112 on page 196.

Figure 128 shows the HTML file for the applet.

```
<HTML>
<HEAD>
<TITLE>RMI JDBC Applet</TITLE>
</HEAD>
<BODY>
<H1>RMI JDBC Applet</H1>
<applet code="network.rmijdbc.RmiGui.class" width=500 height=500
 alt='Please enable Java to see the applet'>
Sorry but your browser does not support Java applets.
</applet>
</HTML>
```

Figure 128. RMI JDBC Applet HTML: RmiGui.htm

Enhancements for the RMI Controller

The current implementation assumes that only one GUI connects to the controller object at any given time.

We could modify our code so that the GUI registers itself on a ControllerFactory object, which then makes a new Controller object for this client. This new Controller object can be allocated on a pool of machines. This approach is preferred because it truly makes use of the UnicastRemoteObject.

After adapting the client code to run as an applet on a Web page, we have a highly scalable solution:

- The applet can be accessed from multiple HTTP servers.
- The ControllerFactory can be put on a server. It then can distribute controller objects running on different servers. Once the GUIs have a reference to the controller object, they can *talk* directly to the server running the control object.
- The existing DB2 distribution services can be applied to optimize the access from the controller objects to the database.

Figure 129 summarizes this three-tier construction, which demonstrates the power of distributing objects, using tailorable GUI clients, dedicated code servers (HTTP), distributed database access code on dedicated machines, and optimized DB2 access.

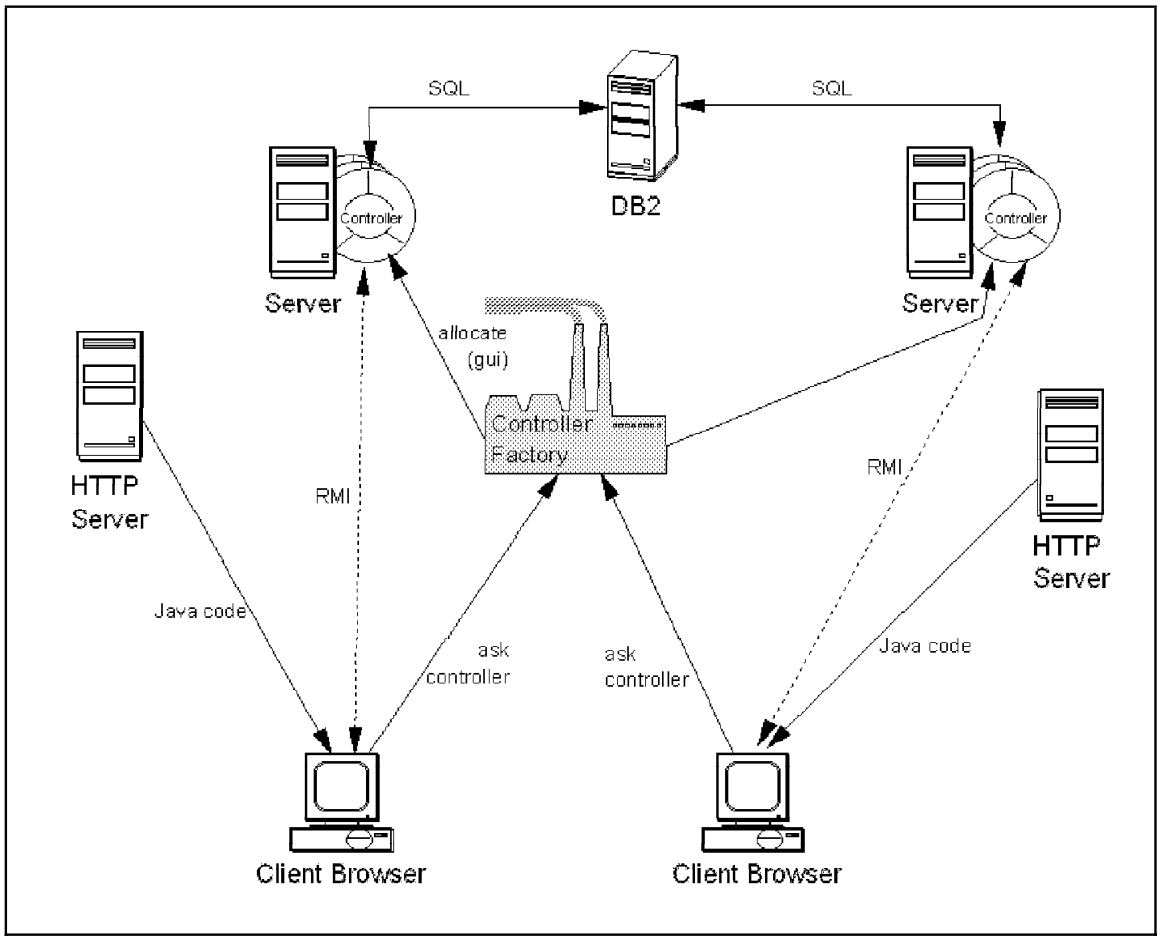


Figure 129. Highly Distributed Client/Server Program Using RMI

Chapter 12. Using NetRexx for CGI Programs

In this chapter we use NetRexx to write Common Gateway Interface (CGI) programs on a Web server. We use the DB2 sample database discussed in “DB2 Sample Database” on page 184 as a base to generate Web pages from real data.

We used the IBM Internet Connection Server (ICS) as a Web server on Windows 95 (or Windows NT). All programs were placed into a JAVA\NRX subdirectory of the main directory of ICS.

Most servers do or will support CGI programs written in Java, also called *server-side Java*. Here is a partial list of servers that support Java:

- IBM Internet Connection Server 4.2
- Microsoft Internet Information Server 3.0
- Netscape Enterprise Server 2.01
- Netscape Fast Track Server 2.01
- Web Site Professional 1.1
- Web Star 2.02

CGI Concepts

Not all Web pages can be predefined on a Web server. In many cases users ask for information that is stored in a database, and a Web page that contains the data retrieved from the database must be dynamically generated.

For this purpose Web servers provide the CGI, a standardized API for invoking programs with parameters supplied from a Web browser and accepting a Web page from such a program to be returned to the Web browser.

CGI programs can be written in many languages, such as C, C++, Rexx, and Java.

Passing Parameters to a CGI Program

Every language has its own way of accepting parameters from a caller. For CGI the designers decided to prepare the parameters in environment variables and let the programs look up the parameters themselves.

Java, and therefore NetRexx, do not allow direct access to environment variables. Web servers were modified to prepare the parameters for Java programs in *system properties* matching the names of the environment variables.

For our sample programs we access the following environment variables:

SCRIPT_NAME Name of invoked CGI program

REMOTE_ADDR TCP/IP address of remote client

QUERY_STRING The parameter string passed from the client's Web browser when the *get* method is used in the HTML form

CONTENT_LENGTH Number of bytes passed through standard input from the client's Web browser when the *post* method is used in the HTML form

These environment variables are available by using the *getProperty* method of the System class:

```
System.getProperty('QUERY_STRING')
```

Get Method

The QUERY_STRING variable is the most important. It contains the data of the Web browser page. In many cases such data is prepared by using an HTML form. The query string has the general format of variable name and value pairs, separated by & signs:

```
varname1=value1&varname2=value2&...
```

The query string format is based on a few special rules:

- Special characters are replaced by a % sign, followed by their ASCII code, for example, %25 for a % sign.
- Blanks are replaced by + signs.

There are more rules and conventions that you need to know for serious Web CGI programming, but for our simple example these two rules are enough.

Post Method

HTML forms can also use the post method to pass data to the CGI program through standard input. Forms with many fields generally specify the post method.

Our example uses the get method, and the CGI program retrieves the client data through the QUERY_STRING environment variable. The following code extract can be used to construct the query string for the post method:

```
query = BufferedReader(InputStreamReader(System.in)).readLine()
```

We also implemented one of the programs by using the post method (see "CGI Program for Employee Details: Post Method" on page 246).

Returning a Web Page from a CGI Program

The Web page that is returned to the Web browser is written to the standard output by the CGI program, where it is picked up by the Web server.

In NetRexx, we can use the say instruction to write the lines of the Web page.

Two special lines must be generated before the Web page:

```
Content-Type: text/html          <=== describes the format
                                 <=== blank line
<html>                           <=== start of HTML page
 ...                             <=== body of page
</html>                          <=== end of page
```

The first line describes the type of file (MIME type) that is generated, and the second line must be an empty line.

The program can also return a file containing a preformatted HTML page instead of generating a new one, for example, to return an error message. The output of the program would consist of only two lines: a line specifying the URL of a file that contains the Web page, and a blank line:

```
Location: /directory/subdir/filename.ext   <=== URL of file
                                           <=== blank line
```

Note: The location tag can specify a complete URL with protocol, server, and document.

Sample CGI Programs with DB2 Access

Our NetRexx CGI sample consists of two programs:

- The first program searches for matching names of employees in the DB2 sample EMPLOYEE table for a given partial name. It then creates a Web page containing a list of matches.
- The second program lists the details of one employee selected in the list produced by the first program.

HTML Form for Employee Search

The first program is started from a Web page displaying a form where the user can enter a partial employee name (see Figure 130).

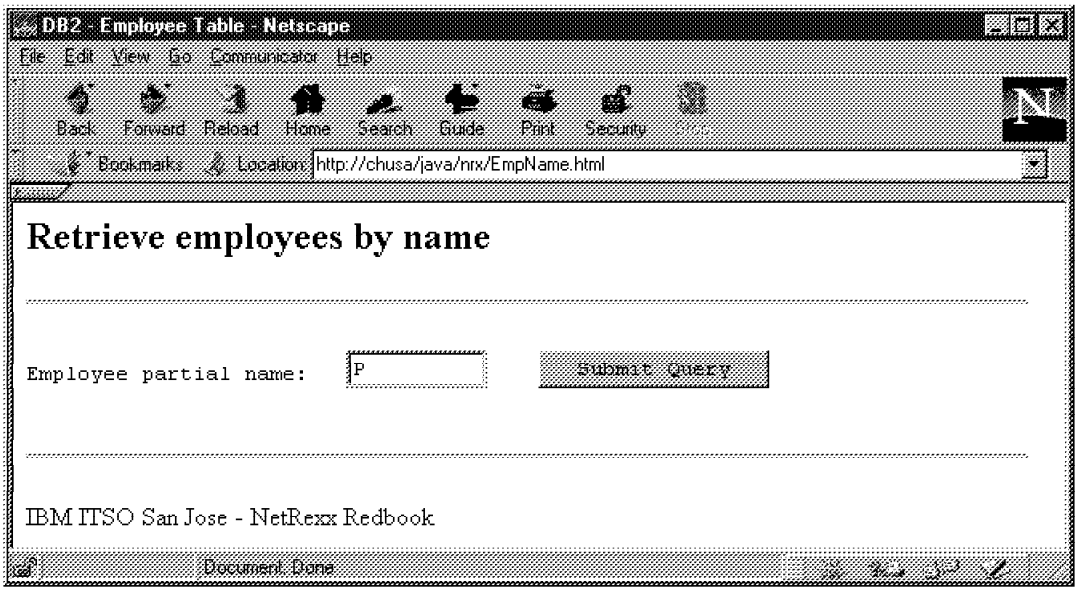


Figure 130. HTML Form for Employee Search

A partial name in the form of a DB2 LIKE clause can be entered, and a percent sign is added at the end. For example, P looks for all employee names starting with P, %N for all names having an N somewhere, and _E looks for names where the second character is an E. The program also allows using an asterisk (*) instead of the percent sign.

Figure 131 shows the HTML file that produces the HTML page.

```
<! cgi\EmpName.html

   NetRexx CGI program for DB2: Employees by name >

<html> <head> <title>DB2 - Employee Table </title> </head>
<body>
<h2> Retrieve employees by name </h2>
<hr>
<form method="GET" action="/cgi-bin/java/nrx/EmpName.class">
     <p> <pre>Employee partial name:   <input name="name" type="text" size="10">    <input type="submit">
        </pre>
</form>
<hr> <br> IBM ITSO San Jose - NetRexx Redbook
</body></html>
```

Figure 131. HTML Code for Employee Search: EmpName.html

The program that is invoked is named in the *action* specification:

```
action="/cgi-bin/java/nrx/EmpName.class"
```

It is the Web server's job to locate the program on the basis of the relative directory information, prepare the query string, and invoke the program. Because of the *class* extension, the Web server knows that the program is a Java program.

The query string is prepared from the form; in this simple case it contains the data of the single input field:

```
name=xxxxxxxxxx
name=P%25                        <=== from P%
```

where *xxxxxxxxxx* is the partial name entered by the user.

CGI Program for Employee Search

Figure 132 lists the CGI program for the employee search.

```
/* cgi\EmpName.nrx

   NetRexx CGI program for DB2:  Employees by name */

   import java.sql.

Class EmpName

/*------------------------- variables -----------------------------*/
properties static
  prefix    = Rexx 'USERID'              -- table prefix
  con       = Connection                 -- DB2 connection
  driver    = String "COM.ibm.db2.jdbc.net.DB2Driver"
  --driver  = String "COM.ibm.db2.jdbc.app.DB2Driver"
  url       = String "jdbc:db2:/loopback:8888/sample"
  --url     = String "jdbc:db2:sample"
  partialname = String


/*------------------------- main ----------------------------------*/
```

Figure 132 (Part 1 of 3). CGI Program for Employee Search: EmpName.nrx

```
method main(args=String[]) static
  args = args
  say 'Content-Type: text/html'                    -- control lines
  say ''
  say '<html>'                                     -- start HTML
  say '<head><title>Employee Information</title></head>'
  say '<body>'
  say '<H2>Employee List</H2>'
  say '<br> Program:' System.getProperty('SCRIPT_NAME')
  say '<br> Client :' System.getProperty('REMOTE_ADDR')

  list = Rexx System.getProperty('QUERY_STRING')  -- query string
  list = queryTranslate(list)
  list = list.translate('%','*')                   -- DB2 LIKE
  say '<br> Query  :' list
  parse list 'name=' partialnamex ' '              -- get partial name
  partialname = partialnamex.upper"%"

  jdbcConnect()                                    -- JDBC connect to DB
  performRetrieve()                                -- DB2 SQL

  say '</body>'                                    -- end HTML
  say '</html>'
  return

/*-------------------------- Query translate ------------------------*/
method queryTranslate(qry=Rexx) private static returns Rexx
  qryt = qry.translate('  ','+')                   -- + are blanks
  ist = qryt.pos('%')
  loop while ist > 0
     c = qryt.substr(ist+1,2).x2c
     qryt = qryt.substr(1,ist-1)''c''qryt.substr(ist+3)
     ist = qryt.pos('%',ist+1)
  end
  return qryt

/*-------------------------- JDBC connect --------------------------*/
method jdbcConnect() private static
  do
      -- say '<p> JDBC driver:' driver
      say '<br>Connection :' url
      Class.forName(driver)
      con = Connection DriverManager.getConnection(url,"userid","password")
      if con.getWarnings() \= null then do
         say '<p> Error' con.getWarnings().getMessage()
         return
      end
      dma = DatabaseMetaData con.getMetaData()
      say "<br>Driver     :" dma.getDriverName() dma.getDriverVersion()

    catch ex=SQLException
      say "<p> *** SQLException caught ***"
      say '<br>' ex.getMessage()
      loop while (ex \= null)
         say "<br>SQLState:" ex.getSQLState()
         say "<br>Message: " ex.getMessage()
         say "<br>Vendor:  " ex.getErrorCode()
         ex = ex.getNextException()
```

Figure 132 (Part 2 of 3). CGI Program for Employee Search: EmpName.nrx

```
          say "<br>"
        end
      catch ex2=java.lang.Exception
                      -- Got some other type of exception.  Dump it.
        say '<p> Error:' ex2.getMessage()
        ex2.printStackTrace()
    end
    return

  /*------------------------- retrieve employee ----------------------*/
  method performRetrieve() private static

    say '<p> Retrieving employees:' partialname
    do                                                    -- DB2
        query = "SELECT empno, lastname, firstnme" -
                 "FROM" prefix".employee" -
                 "WHERE lastname LIKE '"partialname"'"
        /* say '<br>' query */
        say '<p>'
        say '<table border=2 cellpadding=0>'
        say '<tr>'
        say '<th>Number</th> <th>Lastname</th> <th>Firstname</th> '
        say '<tr>'
        stmt = Statement con.createStatement()
        rs = ResultSet stmt.executeQuery(query)
        more = boolean rs.next()
        loop row=0 by 1 while (more)
           num = Rexx rs.getString('empno')
           say '<td>' '<a href="EmpNum.class?number='num'"> <b>' num '</b></a></td>'
           say '<td>' rs.getString('lastname') '</td>'
           say '<td>' rs.getString('firstnme') '</td>'
           say '<tr>'
           more = rs.next()
        end
        say '</table>'
        rs.close()                              -- Close the result set
        stmt.close()                            -- Close the statement
        say '<p> Retrieved' row 'employees'

      catch ex=SQLException
        say '<p> Error:' ex.getMessage()
    end
  -- end EmpName
```

Figure 132 (Part 3 of 3). CGI Program for Employee Search: EmpName.nrx

The program starts by writing the two control lines and the start of the Web page. It then accesses the environment variables through the system properties.

The query string is analyzed by the queryTranslate method that reformats special characters and blanks.

The jdbcConnect method is invoked next; it uses JDBC to connect to DB2 (see "JDBC Concepts" on page 181 for details). The performRetrieve method runs the SQL statement to find matching employees and formats the list of matches as an HTML table.

The employee number is formatted in the first column of the table as a hot reference:

```
<a href="EmpNum.class?number='num'"> <b>' num '</b></a>
```

Clicking on the employee number in the browser invokes the second CGI program, EmpNum.class, with the number as the single parameter:

```
number=xxxxxx
```

HTML Table of Employees

The first program displays the matching employees in an HTML table (see Figure 133).

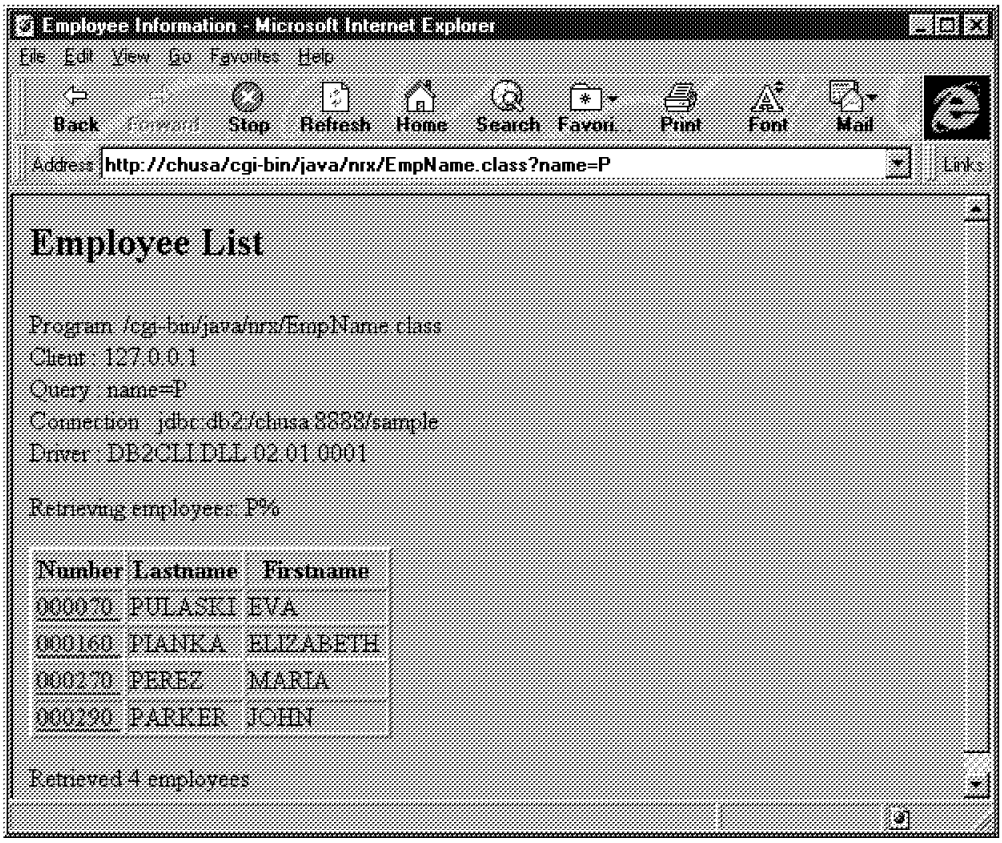


Figure 133. HTML Table of Matching Employees

Clicking on one of the employee numbers invokes the second program to display the employee's details.

CGI Program for Employee Details

Figure 134 shows the CGI program for employee details.

```
/* cgi\EmpNum.nrx

   NetRexx CGI program for DB2:  Employee by number (using GET) */

   import java.sql.

Class EmpNum

/*------------------------- variables ------------------------------*/
properties static
```

Figure 134 (Part 1 of 3). CGI Program for Employee Details: EmpNum.nrx

```
  prefix    = Rexx 'USERID'              -- table prefix
  con       = Connection                 -- DB2 connection
  driver    = String "COM.ibm.db2.jdbc.net.DB2Driver"
  --driver  = String "COM.ibm.db2.jdbc.app.DB2Driver"
  url       = String "jdbc:db2:/loopback:8888/sample"
  --url     = String "jdbc:db2:sample"
  empno     = String

/*------------------------- main ----------------------------------*/
method main(args=String[]) static
  args = args
  say 'Content-Type: text/html'                    -- special lines
  say ''
  say '<html>'                                     -- start HTML
  say '<head><title>Employee Information</title></head>'
  say '<body>'
  say '<H2>Employee Data</H2>'
  say '<br> Program:' System.getProperty('SCRIPT_NAME')
  list = System.getProperty('QUERY_STRING')        -- query string
  -- list = queryTranslate(list)                   -- not necessary here
  parse list 'number=' empno ' '

  jdbcConnect()                                    -- JDBC connect to DB2
  performRetrieve()                                -- DB2 SQL

  say '</body>'                                    -- end HTML
  say '</html>'
  return

/*------------------------- JDBC Connect --------------------------*/
method jdbcConnect() private static
  do
      Class.forName(driver)
      con = Connection DriverManager.getConnection(url,"userid","password")
      if con.getWarnings() \= null then do
         say '<p> JDBC driver:' driver
         say '<br>Connection :' url
         say '<p> Error' con.getWarnings().getMessage()
         return
      end

    catch ex=SQLException
      say "<p> *** SQLException caught ***"
      say '<br>' ex.getMessage()
      loop while (ex \= null)
         say "<br>SQLState:" ex.getSQLState()
         say "<br>Message: " ex.getMessage()
         say "<br>Vendor:  " ex.getErrorCode()
         ex = ex.getNextException()
         say "<br>"
      end
    catch ex2=java.lang.Exception
                    -- Got some other type of exception.  Dump it.
      say '<p> Error:' ex2.getMessage()
      ex2.printStackTrace()
  end
  return
```

Figure 134 (Part 2 of 3). CGI Program for Employee Details: EmpNum.nrx

```
/*-------------------------- retrieve employee ----------------------*/
method performRetrieve() private static

  say '<br> Retrieving employee:' empno
  do                                                        -- DB2
      query = "SELECT empno, firstnme, midinit, lastname," -
                     "phoneno, sex, birthdate, hiredate, job, edlevel," -
                     "salary, bonus, comm, workdept" -
                "FROM" prefix".employee" -
                "WHERE empno = '"empno"'"
      stmt = Statement con.createStatement()
      rs = ResultSet stmt.executeQuery(query)
      more = boolean rs.next()
      say '<pre>'
      loop row=0 by 1 while (more)
         say '<p>'
         say ' Emp-num  :' rs.getString('empno')
         say ' Name     :' rs.getString('firstnme') rs.getString('midinit') -
                              rs.getString('lastname')
         say ' Phone    :' rs.getString('phoneno')
         say ' Sex      :' rs.getString('sex')
         say ' Birthdate:' rs.getString('birthdate')
         say ' Hiredate :' rs.getString('hiredate')
         say ' Job      :' rs.getString('job')
         say ' Educ-lev :' rs.getString('edlevel')
         say ' Salary   :' rs.getString('salary')
         say ' Bonus    :' rs.getString('bonus')
         say ' Commiss  :' rs.getString('comm')
         if rs.getString('workdept') \= null then
         say ' Dept     :' rs.getString('workdept')
         more = rs.next()
      end
      say '</pre>'
      rs.close()                                -- close the result set
      stmt.close()                              -- close the statement
      if row=0 then say '<p> Employee' empno 'not found'

    catch ex=SQLException
      say '<p> Error:' ex.getMessage()
  end
-- end EmpNum
```

Figure 134 (Part 3 of 3). CGI Program for Employee Details: EmpNum.nrx

The second program is very similar to the first one. It writes the two control lines and the start of the Web page, and accesses the environment variables through the system properties.

The query string is not translated because no special characters are passed. The jdbcConnect method is invoked to connect to DB2, and the performRetrieve method runs the SQL statement to retrieve the details of one employee.

The employee details are written as preformatted HTML lines (see Figure 135).

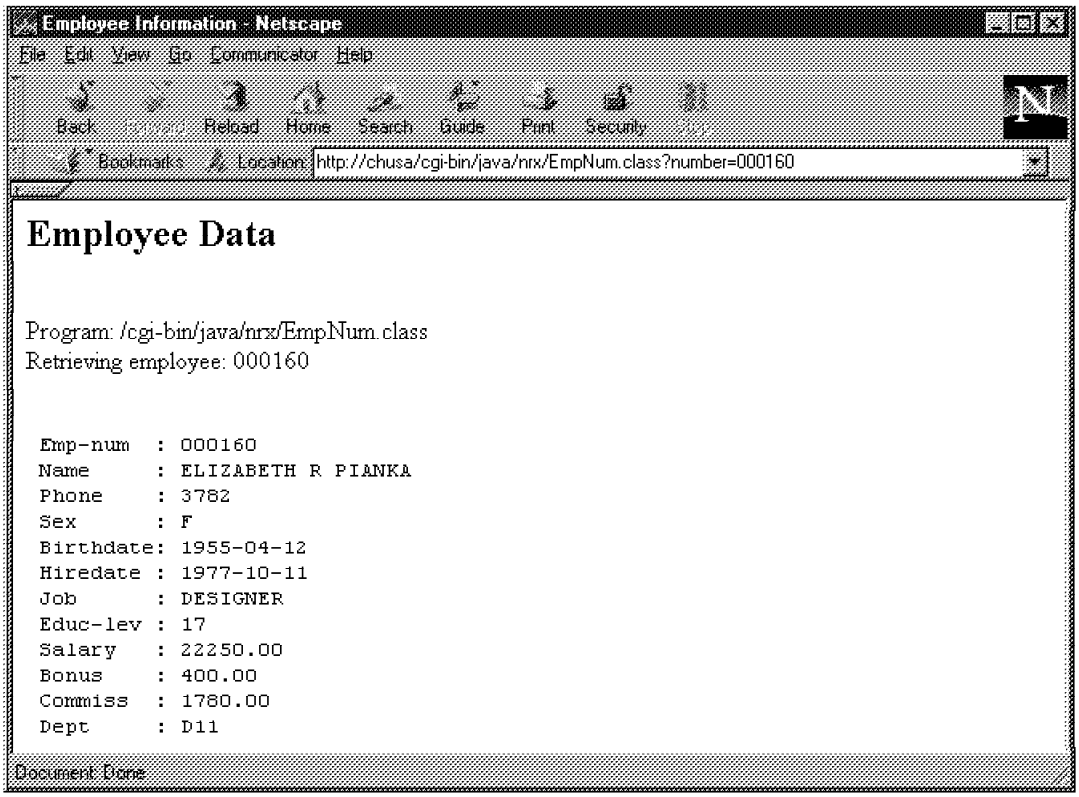


Figure 135. HTML Page with Employee Details

CGI Program for Employee Details: Post Method

We also implemented the employee detail program, using the post method.

With the post method, there is no QUERY_STRING; the data from the form is passed through standard output and must be read by the CGI program. The post method is convenient for large forms with many data items.

Figure 136 shows an extract of the CGI program for employee details using the post method. Note that, in contrast to the CGI program in Figure 134 on page 243, in the main method the query string is constructed by reading the form′s data from standard input.

```
/* cgi\EmpNum2.nrx

   NetRexx CGI program for DB2:  Employee by number (using POST) */

   ......

/*-------------------------- main ----------------------------------*/
method main(args=String[]) static
  args = args
  say 'Content-Type: text/html'                    -- special lines
  say ''
  say '<html>'                                     -- start HTML
  say '<head><title>Employee Information</title></head>'
  say '<body>'
  say '<H2>Employee Data</H2>'
  say '<br> Program:' System.getProperty('SCRIPT_NAME') -
       'query length' System.getProperty('CONTENT_LENGTH')

  -- read the posted data (it is in one line)
  list = BufferedReader(InputStreamReader(System.in)).readLine()
  -- list = queryTranslate(list)                   -- not necessary here
  parse list 'number=' empno ' '

  jdbcConnect()                                    -- JDBC connect to DB2
  performRetrieve()                                -- DB2 SQL

  say '</body>'                                    -- end HTML
  say '</html>'
  return
  ......
```

Figure 136. CGI Program for Employee Details Using Post: EmpNum2.nrx

These simple programs show that NetRexx can be used to quickly write CGI programs.

Chapter 13. Creating JavaBeans With NetRexx

In this chapter we use NetRexx to create JavaBeans that can be used by other products, such as VisualAge for Java. We create simple beans only and do not elaborate on all of the techniques and concepts associated with JavaBeans.

JavaBeans Concepts

A JavaBean has:

Properties Attributes of the class, usually with both a get and a set method, to be retrieved and changed from outside. Properties can be simple, bound, or constrained. Simple properties do not fire an event when they are changed. Bound properties fire a *PropertyChange* event when they are changed, and other beans (classes) can add themselves to the list of PropertyChangeListeners. Constrained properties allow outside classes to verify and potentially veto the change.

Methods Public methods that can be invoked from outside

Events Events that happen inside the bean and that trigger actions in outside classes that are registered on the event. The PropertyChange event is an event associated with bound or constrained properties. Additional user defined events can be added.

A JavaBean class is usually accompanied by a bean information class that defines the names of the properties, methods, and events.

The bean information is optional for properties and methods, but it is required for events. The properties and methods information can be extracted from the bean itself by using Java's introspection and reflection support.

Writing a Bean in NetRexx

Let us define a simple nonvisual bean for an employee. The bean has three properties, two methods, and one event.

Properties The properties are the employee number, employee name, and the salary. The salary only has a public get method; the set method is private. Changing the salary is possible only through special public methods.

Methods Two methods, increaseSalary and decreaseSalary, are provided to change the employee's salary.

Events An action event is triggered when the salary exceeds a certain limit.

Bean Class

A bean definition starts with an import of the java beans package and the class instruction:

```
import java.beans.
package beanlab
class EmpBean public binary
```

We use a binary class to omit references to the Rexx class.

Properties

The properties are defined as private, because accessor methods for get and set will be defined:

```
properties private
  fieldNumber = int 0
  fieldName   = String null
  fieldSalary = float 0
```

Note: NetRexx provides experimental support for *indirect* properties that automatically generates get and set methods:

```
properties indirect
  fieldNumber = int 0
```

The generated variable is a private instance variable with a get and set method. For more information see *nrbean.htm* in the NetRexx directory.

Property Get Methods

For each property there is a public get method:

```
method getNumber() public returns int
  return fieldNumber

method getName() public returns String
  if fieldName == null then fieldName = String ''
  return fieldName

method getSalary() public returns float
  return fieldSalary
```

The set methods are somewhat more complex because we have to trigger the PropertyChange event for bound properties.

PropertyChange Event

Whenever a bound property changes, we have to invoke the PropertyChange event, using an attribute and three methods:

```
properties inheritable
  propertyChange = PropertyChangeSupport(this)

method addPropertyChangeListener(listener=PropertyChangeListener) public protect
  propertyChange.addPropertyChangeListener(listener)

method removePropertyChangeListener(listener=PropertyChangeListener) public protect
  propertyChange.removePropertyChangeListener(listener)

method firePropertyChange(propertyName=String, oldValue=Object, newValue=Object) public
```

```
  propertyChange.firePropertyChange(propertyName, oldValue, newValue)
```

Other classes that want to be notified when a property changes have to:

- Invoke the addPropertyChangeListener method to be added to the list of classes that are notified
- Implement the PropertyChangeListener interface, that is, provide a propertyChange method.

The firePropertyChange method of the employee bean invokes the propertyChange method of all classes that are registered for the ProperyChange event.

Property Set Methods

The set methods of bound properties invoke the firePropertyChange method. In our case we define the set method for salary private, because the salary can change only through increase and decrease methods:

```
method setNumber(number=int) public
  oldValue = int fieldNumber
  fieldNumber = number
  firePropertyChange("number", Integer(oldValue), Integer(number))

method setName(name=String) public
  oldValue = String fieldName
  fieldName = name
  firePropertyChange("name", oldValue, name)

method setSalary(salary=float) private
  oldValue = float fieldSalary
  fieldSalary = salary
  firePropertyChange("salary", Float(oldValue), Float(salary))
```

Note that the old and new values of the property must be passed as subclasses of Object (Integer or Float), and not using the basic int and float types.

Public Methods

Changes to the salary are made by using two public methods:

```
method increaseSalary(amount=float) public
  setSalary( getSalary() + amount )

method decreaseSalary(amount=float) public
  setSalary( getSalary() - amount )
```

These methods use the private setSalary method and therefore trigger the PropertyChange event.

Action Event

An action event is implemented through a *fire* method and two other methods to maintain a Vector of listeners. Classes that want to be notified add themselves to the Vector, using the addActionListener method:

```
properties inheritable
  aActionListener = Vector null

method addActionListener(newListener=ActionListener) public
  if aActionListener == null then aActionListener = Vector()
  aActionListener.addElement(newListener)
```

```
method removeActionListener(newListener=ActionListener) public
  if aActionListener \= null then
     aActionListener.removeElement(newListener)

method fireActionPerformed(e=ActionEvent)
  if aActionListener == null then return
  currentListeners = aActionListener.clone().elements()
  loop while currentListeners.hasMoreElements()
     (ActionListener currentListeners.nextElement()).actionPerformed(e)
  end
```

The fireActionPerformed method must be invoked within the class when the event occurs. It sends the event to every registered listener.

Triggering the Action Event

In our sample we want to trigger an action when the salary exceeds a certain limit:

```
properties constant
  salaryLimit = float 100

method setSalary(salary=float) private
  oldValue = float fieldSalary
  fieldSalary = salary
  if salary > salaryLimit then do       -- over the limit
     salary = salaryLimit
     fieldSalary = salaryLimit
     fireActionPerformed( ActionEvent( this, -
                                       ActionEvent.ACTION_FIRST, -
                                       "SALARY-LIMIT" )            )
  end
  firePropertyChange("salary", Float(oldValue), Float(salary))
```

To fire the action event, we must create an Action Event object containing the source of the action (the employee), an action code, and a string.

Bean Information Class

It is enough to write the class of the bean (in our example, EmpBean) if the bean consists only of properties and methods. Java's introspection support allows other programs to analyze the class and find its public interface.

However, if a bean has an action event, we have to create a *bean information class* with additional information:

```
import java.beans.
package beanlab
class EmpBeanBeanInfo extends SimpleBeanInfo public binary
```

The constructor records the name of the class:

```
properties private
   beanClass = Class

method EmpBeanBeanInfo() public
   beanClass = Class.forName("beanlab.EmpBean")
```

The action event is defined by using a descriptor method:

```
method getEventSetDescriptors() returns EventSetDescriptor[] public
  eventDesc = EventSetDescriptor[1]
  do
     eventDesc[0] = EventSetDescriptor( beanClass, "actionPerformed", -
                                        ActionListener.class, "actionPerformed" )
     eventDesc[0].setDisplayName("empEvent")
     eventDesc[0].setShortDescription("employee event")
     return eventDesc
   catch Throwable
  end
  return null
```

Optionally you can describe properties and methods in the bean information class, using *xxxxxPropertyDescriptor* and *xxxxxMethodDescriptor* methods, where *xxxxx* is the name of the property or method.

The source code for the nonvisual employee bean (EmpBean.nrx and EmpBeanBeanInfo.nrx) is in the nrxbeans\sample subdirectory.

Using the NetRexx Bean in VisualAge for Java

First compile the NetRexx source into Java source code:

```
nrc EmpBean -keep
nrc EmpBeanBeanInfo -keep
```

Rename the source code from EmpBean.java.keep to EmpBean.java, and from EmpBeanBeanInfo.java.keep to EmpBeanBeanInfo.java.

Import the Java source code into a VisualAge for Java project. A package named *nrxbeans.sample* is created.

Open the EmpBean class, and go to the BeanInfo page. Select each property and change the value of the bound attribute to true. This process regenerates the full EmpBeanBeanInfo class.

Using the Bean in an Applet

Open the Visual Composition Editor of a new applet. Place the employee bean on the free form surface. Connect the properties to entry fields in the applet frame, and connect push buttons to the public methods of the bean, increaseSalary and decreaseSalary. Use the bean action event to trigger methods in the applet frame, for example, to change the text in entry fields, or to disable push buttons.

Figure 137 shows the Visual Composition Editor with an applet and the sample employee bean.

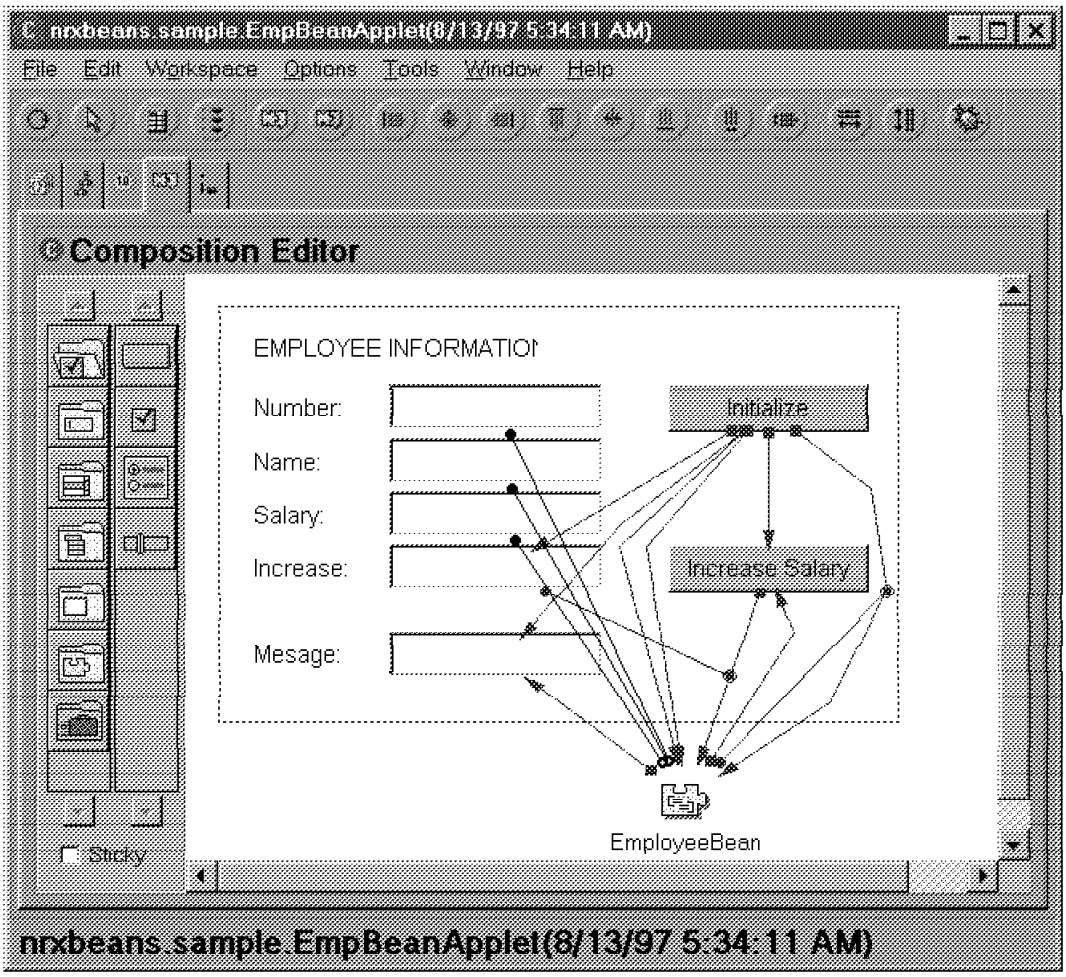


Figure 137. Visual Composition Editor with NetRexx Bean

To initialize the applet use the **Initialize** button. The employee number and name are set to a string, the increase field to 30, and the message to display *Initialized*. The **Increase Salary** push button is enabled.

The employee bean is connected in many ways:

- Properties are connected to entry fields in the applet frame.
- Methods are connected to push buttons; the **Initialize** button is connected to the decreaseSalary method to set the salary to 0, and the **Increase Salary** button is connected to the increaseSalary method with the increase entry field value as a parameter.
- The action event, which is triggered when the salary limit is reached, is connected to the message to display *Salary limit reached* and to the **Increase Salary** push button to disable it.

Figure 138 shows the applet in action when the salary has exceeded 100 after four increases. The message field has been set, and the push button is disabled.

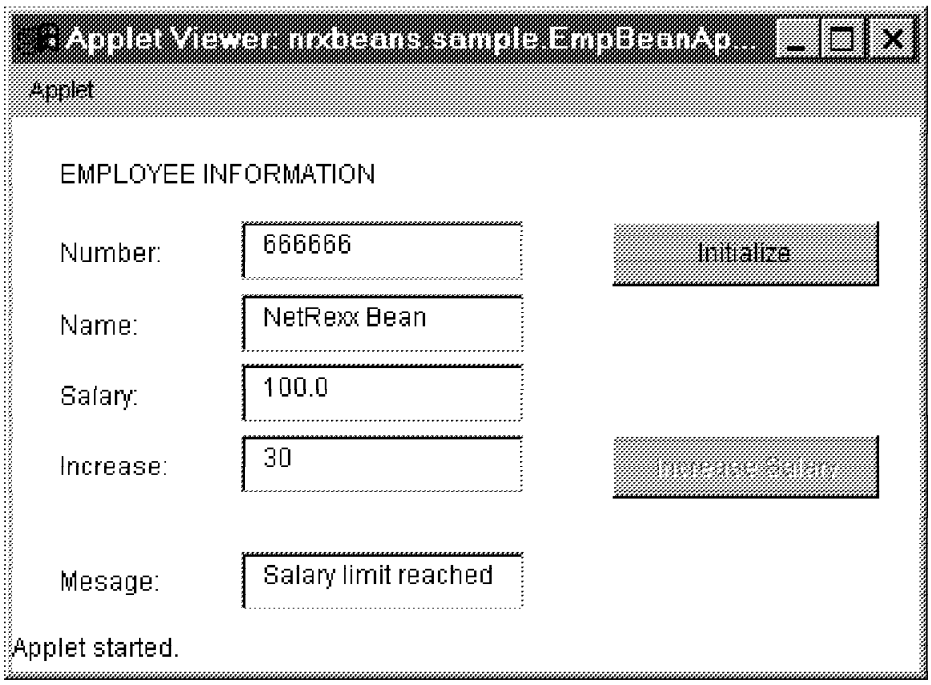


Figure 138. VisualAge for Java Applet with NetRexx Bean in Action

The executable applet (EmpBeanApplet.class) was exported from VisualAge for Java and stored in the nrxbeans\sample subdirectory, together with an HTML file to run the applet (EmpBean.htm)

Creating an Animated JavaBean

This example takes the animated applet (see Figure 67 on page 122) and turns it into a bean.

The code changes are minimal:

- We name the class AnimBean.
- Instead of an applet, we use a subclass of Frame.
- We add a default constructor.
- We rename the init method to *animate* and accept a text string as a parameter; this is the external interface.

We compile the program into Java code and import the source code into VisualAge for Java.

We open the AnimBean class and generate the bean information, that is, the AnimBeanBeanInfo class.

Now we create an applet, AnimTest, and add the bean to the free form surface. We define an **Animate** push button and connect it to the *animate* method of the bean. We use an entry field for the text and connect it as a parameter to the previous connection. We use the same technique to set the text string as the title of the frame window.

Figure 139 shows the Visual Composition Editor with the applet in the dashed rectangle and the bean (frame) outside.

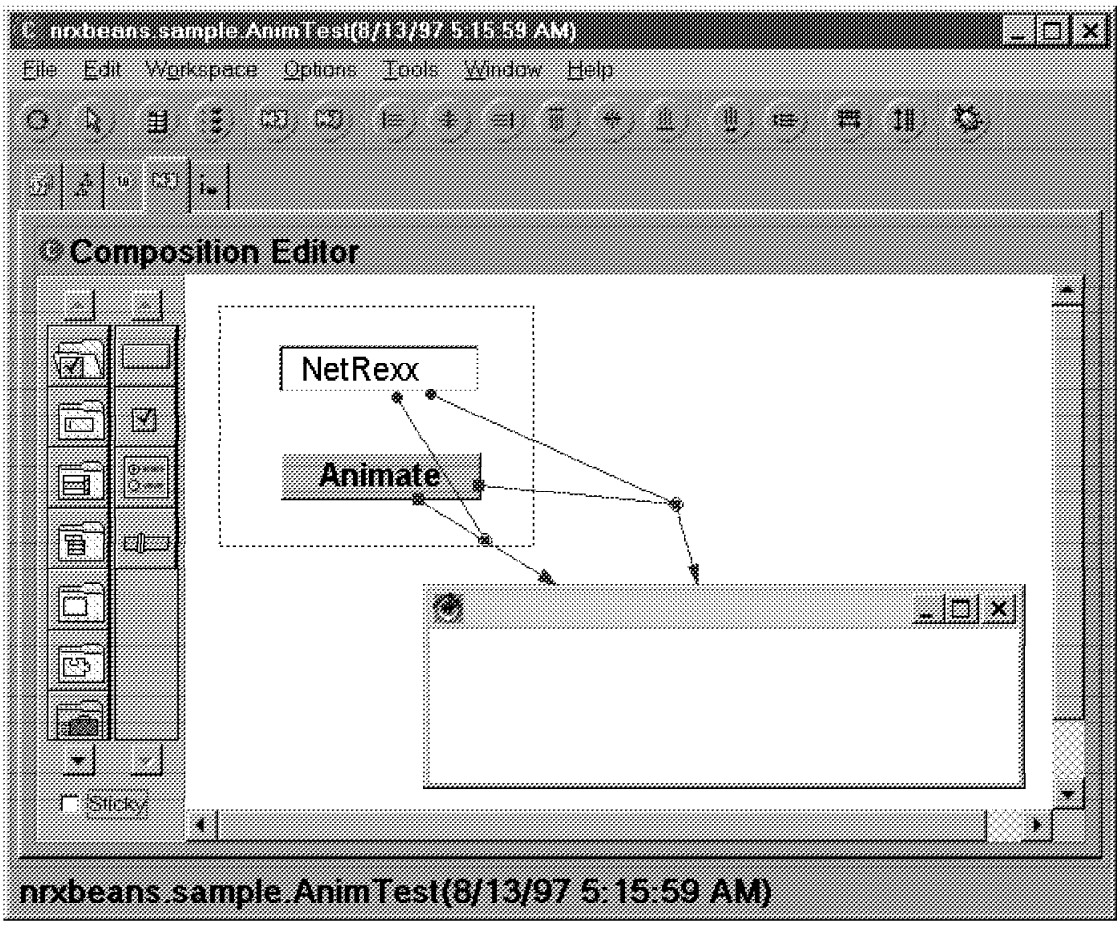


Figure 139. VisualAge for Java Applet with Animated Bean

When we run the applet as an application (a feature of VisualAge for Java), we can see the text running from right to left and being squeezed to the border. A snapshot is shown in Figure 140.

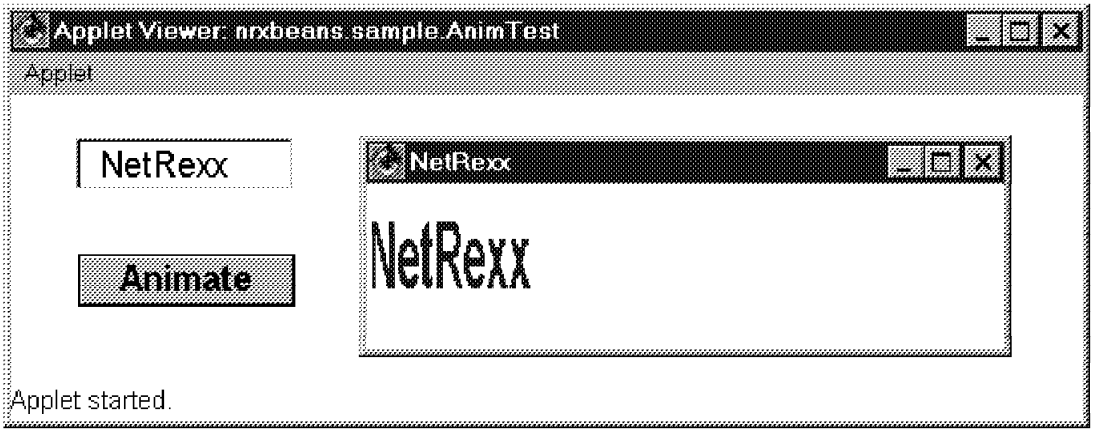


Figure 140. Animated Bean in Action

The source code of the animated bean (AnimBean.nrx) and the HTML file to run the applet (AnimTest.htm) are in the nrxbeans\sample subdirectory. The executable applet (AnimTest.class) was exported from VisualAge for Java into the same directory.

Sample NetRexx Beans

We created a number of additional beans with NetRexx:

NrxCounter A simple counter with two bound properties: count and step. The count is initialized with 0 and the step with 1. The changeCount method changes the counter by adding the step value.

NrxLED An LED display that displays a string, using bitmaps that represent the numbers (0 to 9) and the characters blank, colon, period, and minus. The numberOfDigits property defines the size of the LED, and the text property, the string that is displayed; both properties are not bound. The setText method changes the text and displays it, using the paint method. The paint method uses an array of preloaded GIF files to display the text.

NrxTimer A timer bean that signals an event in a specified interval, using a thread. Three bound properties are provided: interval (default 1000 milliseconds), enabled (0), and tickAtBeginning (0). The setEnabled method starts and stops the timer. The setTickAtBeginning method can be used to send an event when the timer is started. Other beans that want to be notified at each time interval register with the timer by using the addActionListener method.

NrxLight A light bulb with three bound properties: lightOn (default 0), onColor (green), and offColor (red). The switchLight method changes the state of the light bulb from off to on and vice versa. The paint method displays the light bulb in the on or off color. The bean also implements the ActionListener interface and switches the light when the actionperformed method is invoked. Thus you can add a light as an ActionListener to the timer, and the light switches at each timer interval.

We created a number of applications and applets, using the sample NetRexx beans:

ViewTime An applet that displays the time it runs in seconds. It uses the timer to signal the seconds, the counter to count, and the LED to display the counter.

Timer An applet that displays the time it runs in hours, minutes, and seconds. It uses the timer to signal the seconds, the counter to count, and the LED to display the time in the format h:mm:ss.

CountDown An application that simulates a countdown timer. You enter the time in seconds to count down, and click on **Start**. It uses the timer to signal the seconds, the LED to display the remaining time, and two light bulbs. One light bulb is on when the timer is running, and the other switches on and off at each tick of the timer (see Figure 141).

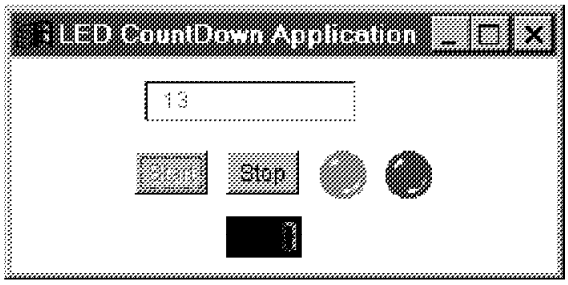


Figure 141. CountDown Applet with NetRexx Beans

StopWatch An applet that simulates a stop watch. It uses the timer to signal every 1/10th of a second, the LED to display the elapsed time, and a light that is on when the stop watch is running. Three push buttons enable you to start, stop, and reset the stop watch (see Figure 142).

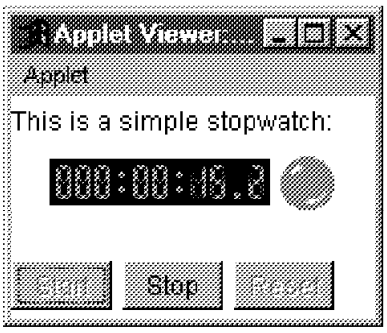


Figure 142. StopWatch Applet with NetRexx Beans

We also imported the NetRexx beans into VisualAge for Java and implemented the stopwatch using the imported beans. Figure 143 shows how we created the applet with the Composition Editor.

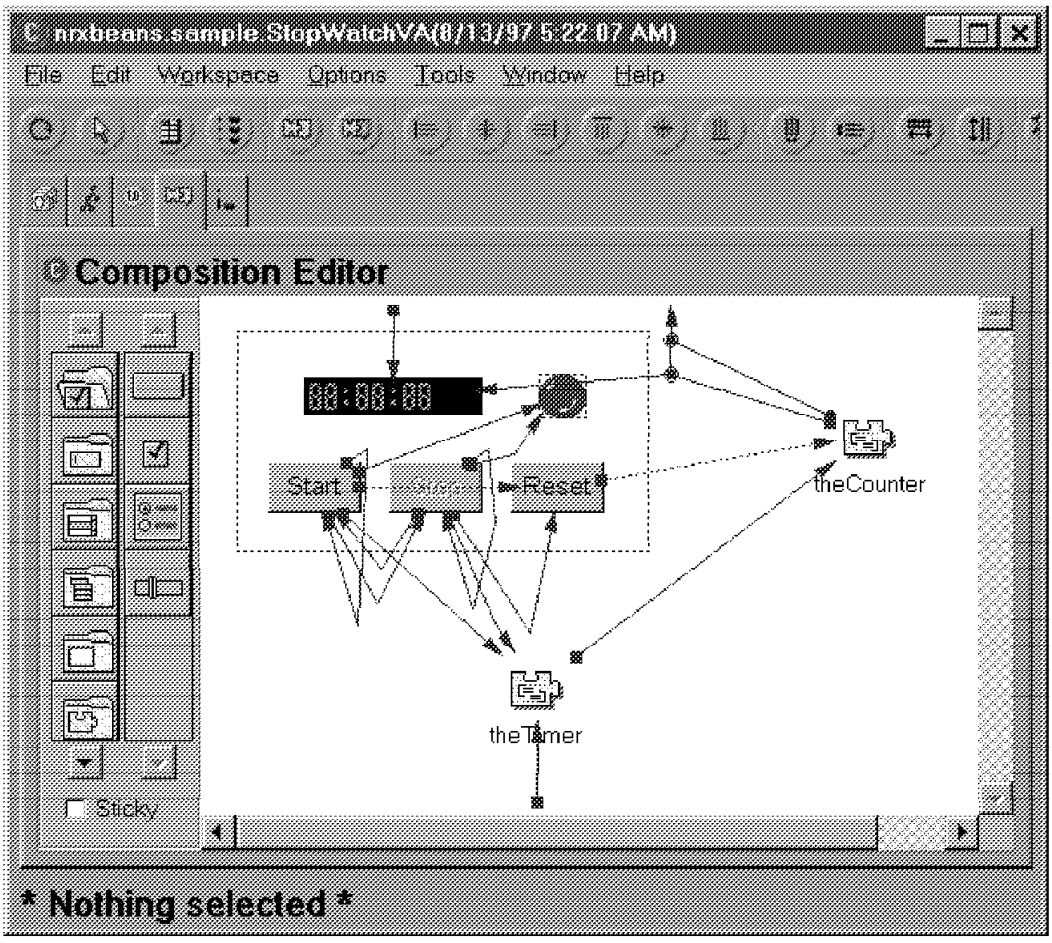


Figure 143. StopWatch Applet in VisualAge for Java

The sample NetRexx beans are in the nrxbeans\lab directory. The sample applets, applications, and HTML files to run the applets are in the nrxbeans\sample directory.

Chapter 14. Why NetRexx?

By now you are familiar with many of the features of NetRexx and Java. So why should you use NetRexx and not Java directly? Let's look at some advantages of using NetRexx.

Simplicity of Coding

- NetRexx programs are very easy to read.
- You have a choice of nested comments and line comments:

```
/* this is a comment
   with /* a nested */ comment */
a = b * b      -- calculating the square (line comment)
```

- Variable declarations with type, and optionally an initial value or constructor, are easy to understand:

```
var1 = int
var2 = float 6.6
var3 = Frame()
```

- Say and ask provide simple console input and output:

```
say 'Enter your age:\-'     -- no newline
age = ask                   -- ask keyword for input
```

- Automatic generation of class and main method allows for simple scripting programs:

```
/* my program */
say 'There is no need for a class or main method'
```

See "Our First NetRexx Program" on page 7 for more information.

- Scripting programs can have subroutines and functions coded as static methods:

```
/* my program */
say 'result=' factorial(13)

method factorial(num=int) static returns Rexx
  res = 1
  loop i=2 to num
    res = res * i
  end
  return res
```

See "Subroutines and Functions" on page 50 for more information.

- Methods with optional parameters generate multiple Java methods, one for each signature. This example generates three methods in Java:

```
method myroutine(p1=int, p2=float 6.0, p3=String "A")
```

- The syntax for exception handling using do blocks, loops, and select groups is simple and easy to read:

```
loop i=1 to 10 while ...
   ....
   ....
 catch Exception
   ....
 finally
   ....
end
```

  See “Control Statements” on page 39 for more information.

- The compiler automatically adds exceptions to the signal list of methods if they are signaled in a method and not caught by a catch clause.

Rexx Class

- Provides arithmetic to any precision
- Combines numeric and string processing into one class
- Provides a multitude of methods for string handling (see “Built-In Methods” on page 31)
- Provides a simple parsing mechanism to analyze string data (see “Parsing a String” on page 30)
- Indexed strings provide great flexibility for handling collections of strings with any kind of indexes (see “Indexed Strings” on page 38).

See “The Rexx Class for Strings” on page 30 for more information.

Compiler Functions

- The compiler reports variables that are declared but never used.
- The compiler checks each variable against the Java class library; this includes very strict checking of assignments and method parameters.
- Case insensitivity facilitates finding Java classes (even if they are not spelled with the exact case).
- The binary option provides Java-equivalent performance.
- Each statement can be traced at execution time (see “Trace Instruction” on page 47).

Conclusions

- **NetRexx makes Java programming easy.**
- **NetRexx is a great scripting language.**
- **Everything you can do in Java you can do in NetRexx, much more easily.**
- **The whole Java class library is at your fingertips.**
- **NetRexx is the language for server programming** *(GUI programming might be easier with a visual programming tool)***.**
- **The move from classic Rexx to NetRexx is simple; don′t wait, step up now!**

Appendix A. Redbook Package Reference

The redbook package is a set of utility classes that help you create programs fast. Most parts of the package are discussed in Chapter 7, “Creating Graphical User Interfaces” on page 75.

Note: To use the redbook package you must add the NRXREDBK directory to the CLASSPATH:

```
SET CLASSPATH=.;.........;d:\NRXREDBK
```

CloseWindow Class

The CloseWindow class implements an ActionListener and a WindowListener that closes your window. The objects of the class can be configured to shut down the application and destroy or hide the window.

The window listener part of the class implements the windowClosing method, which is called every time the window is closed from the system menu.

Constants

DESTROY Destroys the window when activated

HIDE Sets the visibility to false when activated

SHUTDOWN
Exits the application with *exit 0* when activated

Constructors

CloseWindow(cWindow=Frame)
Creates a CloseWindow object for the given frame. The behavior is set to SHUTDOWN.

CloseWindow(cWindow=Dialog, theBehaviour=int DESTROY)
Creates a CloseWindow object for the given dialog window. When the dialog window is closed, the parent of the dialog is brought to the front.

CloseWindow(parent=Window, cWindow=Window, theBehaviour=int DESTROY)
Creates the CloseWindow object for *cWindow*. If the window is a modal dialog only DESTROY and SHUTDOWN are excepted as behavior. If HIDE is defined, DESTROY is used. When the window is closed, the *parent* window is brought to the front.

EqualSizePanel Class

The EqualSizePanel class implements a panel that makes all its component the same size. The size is determined by the component with the largest preferred size.

Constants

HORIZONTAL
: The components are added to one row.

VERTICAL
: The components are added to one column.

Constructor

EqualSizePanel(alignment=int HORIZONTAL, gap=int 5)
: Creates a panel with the given alignment and the given gap between the components

Methods

add(comp=Component) returns Component
: Adds the component to the panel and returns the reference of the component

getFlowLayout()
: Returns a reference of the FlowLayout manager that is used

setGaps(hgap=int, vgap=int)
: Set the gaps around the components

ExtendedLabel

The ExtentedLabel class is a lightweight component that supports multiple line labels. The horizontal and vertical alignment can be specified.

The lines in the text are separated by \n if no other separator is specified.

Constants

BOTTOM Vertical alignment, at the bottom of the component

CENTER Vertical and horizontal alignment, in the center of the component

LEFT Horizontal alignment, to the left of the component

RIGHT Horizontal alignment, to the right of the component

TOP Vertical alignment, on top of the component

Constructors

ExtendedLabel(ltext=Rexx, hAlign=int LEFT, vAlign=int TOP)
: Creates an ExtendedLabel with the given text and alignment

ExtendedLabel(ltext=Rexx, aInsets=Insets, hAlign=int LEFT, vAlign=int TOP)
: Creates an ExtendedLabel with the given text and alignment. The Insets parameter defines the margins around the text.

Methods

getHorizontalAlignment() returns int
: Returns the current horizontal alignment

getInsets() returns Insets
: Returns the current margins of the component

getMinimumSize() returns int
: Returns the minimum size (the size of the text without any margins)

getPreferredSize() returns int
: Returns the preferred size (the size of the text with the margins)

getVerticalAlignment() returns int
: Returns the current vertical alignment

setFont(f=Font)
: Sets the font and calculates the new size

setForeground(c=Color)
: Sets the color of the text

setHorizontalAlignment(align=int)
: Sets the horizontal alignment

setInsets(newInsets=Insets)
: Sets new margins for the component

setSeparator(sep=char)
: Sets a new separator and reparses the label text with the new separator

setText(lText=String)
: Changes the text of the label

setUpdate(update=boolean)
: If *update* is false, the component suppresses any painting of itself. If *update* is true, the component restarts the painting.

setVerticalAlignment(align=int)
: Sets the vertical alignment

FieldSelect Class

The FieldSelect class is a FocusListener that can be used for TextField objects. It selects the contents of a text field when the focus is set to the field and removes any selection when the focus is lost.

The FieldSelect class is automatically used by the WindowSupport class (see “WindowSupport Class” on page 270).

Constructor

FieldSelect()
: Default constructor

ImagePanel Class

The ImagePanel class is a lightweight component that supports the use of images. The source for an image can be a file or a URL. The image is loaded with a MediaTracker and shown only when loaded completely.

Margins (Insets) can be defined for the component.

The image is scaled with the component, when the component is scaled. The aspect ratio of the image is kept when scaling.

Constants

BOTTOM Vertical alignment, at the bottom of the component

CENTER Vertical and horizontal alignment, in the center of the component

LEFT Horizontal alignment, to the left of the component

RIGHT Horizontal alignment, to the right of the component

TOP Vertical alignment, at the top of the component

Constructors

ImagePanel()
Default constructor

ImagePanel(anImage=Image, newInsets=Insets null)
Creates the component using the given image and insets (margins)

ImagePanel(anImage=Image, hAlign=int, vAlign=int, newInsets=Insets null)
Creates the component using the given image, insets (margins), and alignment

ImagePanel(anImageURL=URL, newInsets=Insets null)
Creates the component and loads the image from a URL. Signals a LoadImageException if loading fails.

ImagePanel(anImageURL=URL, hAlign=int, vAlign=int, newInsets=Insets null)
Creates the component and loads the image from a URL, using the given alignment and insets. Signals a LoadImageException if loading fails.

ImagePanel(imageFile=String, newInsets=Insets null)
Creates the component and loads the image from a file. Signals a LoadImageException if loading fails.

ImagePanel(imageFile=String, hAlign=int, vAlign=int, newInsets=Insets null)
Creates the component and loads the image from a file, using the given alignment and insets. Signals a LoadImageException if loading fails.

Methods

getHorizontalAlignment() returns int
Returns the horizontal alignment

getMinimumSize() returns Dimension
Returns the original size of the image

getImage() returns Image
Returns the current image

getImageSize() returns Dimension
Returns the current scaled size of the image

getInsets() returns Insets
Returns the current insets (margins)

getOriginalImageSize() returns Dimension
Returns the original size of the image

getPreferredSize() returns Dimension
Returns the original size plus the insets (margins)

getVerticalAlignment() returns int
Returns the vertical alignment

setHorizontalAlignment(hAlign=int)
Sets the horizontal alignment

setImage(newImage=Image)
: Sets the current image to *newImage* and repaints the component if visible

setImage(anUrl=URL)
: Loads an image from the given URL and repaints the component if visible. Signals a LoadImageException if loading fails.

setImage(fileName=String)
: Loads an image from the given file and repaints the component if visible. Signals a LoadImageException if loading fails.

setInsets(newInsets=Insets)
: Sets the insets (margins) of the component by cloning the given Insets object

setInsets(itop=int, ileft=int, ibottom=int, iright=int)
: Sets the insets (margins) of the component

setScaling(on=boolean)
: If *on* is true, the image is scaled to the size of the component; if *on* is false, the image is always shown in its original size

setVerticalAlignment(vAlign=int)
: Sets the vertical alignment

KeyCheck Class

The KeyCheck class is a key listener that can be used to control the input of text fields.

A key checker can be configured to translate lowercase characters to uppercase and allow only characters of a given set.

A string of specified characters is used to define the allowed set of characters.

Constants

ALL
: Set that includes all available characters

ALPHA
: Set that includes alphabetic characters and the blank character only

ALPHANUM
: Set that includes numeric and alphabetic characters only

HEXADECIMAL
: Set that includes hexadecimal characters only

NUMERIC
: Set that includes only the numbers from 0 to 9

Constructor

KeyCheck(set=Rexx ALL, toUpperCase=boolean 1)
: Creates a key checker object with the given set. If *toUpperCase* is true, all keys are translated to uppercase.

Methods

setMode(set=Rexx)
: Sets the set of allowed characters

setMode(toUpperCase=boolean)
: Sets the translation mode of the key checker object. If *toUpperCase* is true, all keys are translated to uppercase.

setMode(set=Rexx, toUpperCase=boolean)
: Sets the set of allowed characters and the translation mode of the key checker object

LimitTextField Class

The LimitTextField class is a text listener that can be used to limit the number of characters in a text field.

If more characters are typed in the field, the right-most characters are deleted. If characters are deleted, the field warns with a beep.

Constructor

LimitTextField(theField=TextField, limit=int)
: Creates a LimitTextField object for the given field to the given number of characters

MessageBox Class

The MessageBox class implements a dialog window that shows text in an ExtendedLabel object, with an optional image to the left of the text.

One or more buttons can be added to the message box.

A message box can be created with a default button (Ok) that closes the box.

A newly created message box is invisible. To make it visible, the *setVisible(1)* method must be used.

The parent of a message box must be a frame window. If a component is specified as the parent of the message box, the parent chain of the given component is searched for a frame window. If no frame window is found, a NoFrameWindow exception is signaled.

Constructors

MessageBox(theParent=Frame, title=String, message=String, aButton=Button Button('Ok'))
: Creates a message box with the given title and the given text. If the text contains \n characters, it is split into multiple lines. If null if used for *aButton*, a button is not added to the box.

MessageBox(theParent=Frame, title=String, message=String, imageFile=String, aButton=Button Button('Ok'))
: Creates a message box as above with the given image to the left of the text. No exception is signaled if the image cannot be loaded.

MessageBox(theParent=Frame, title=String, message=String, anImage=Image, aButton=Button Button('Ok'))
: Creates a message box as above with the given image to the left of the text.

MessageBox(theParent=Component, title=String, message=String, aButton=Button Button('Ok'))
: Creates a message box with the given title and the given text. If the text contains \n characters, it is split into multiple lines. If null is used for *aButton*, a button is not added to the box. The parent of the message box is a component. The box itself searches for a frame window in the parent chain. If a frame window is not found, a NoFrameWindow exception is signaled.

MessageBox(theParent=Component, title=String, message=String, imageFile=String, aButton=Button Button('Ok'))
: Creates a message box as above with the given image to the left of the text. No exception is signaled if the image cannot be loaded.

MessageBox(theParent=Component, title=String, message=String, anImage=Image, aButton=Button Button('Ok'))
Creates a message box as above with the given image to the left of the text

Methods

addButton(text=String) returns Button
Creates a button with the given text, adds it to the message box, and attaches a CloseWindow listener to it

addButton(text=String, listener=ActionListener, closeDialog=boolean 1) returns Button
Creates a button with the given text and adds it to the message box. The given action listener is attached to the button, and a CloseWindow listener is attached to the button if *closeWindow* is true.

setCloseBehaviour(newBehaviour=int)
Sets the close behavior of the CloseWindow object (see "CloseWindow Class" on page 261). Possible values are CloseWindow.DESTROY, CloseWindow.HIDE, and CloseWindow.SHUTDOWN.

setVisible(visible=boolean)
If *visible* is true, the message box is made visible. The position of the message box is dependent on the parent window.

PromptDialog

The PromptDialog class creates a dialog window that prompts the user to enter a value.

A newly created prompt dialog box is invisible. To make it visible the *setVisible(1)* method must be used.

The parent of a prompt dialog must be a frame window. If a component is specified as the parent of the prompt dialog, the parent chain of the given component is searched for a frame window. If a frame window is not found, a NoFrameWindow exception is signaled.

Constructors

PromptDialog(parent=Frame, modal=boolean, title=Rexx '', labeltext=Rexx '', fieldText=Rexx '', fieldsize=int 20)
Creates a prompt dialog. If *modal* is true, the dialog is created as a modal dialog; otherwise, as a modeless dialog. The *fieldSize* parameter defines the visible size of the text field.

PromptDialog(parent=Component, modal=boolean, title=Rexx '', labeltext=Rexx '', fieldText=Rexx '', fieldsize=int 20)
Same as the constructor above, except that the parent is a component. If a frame window is not found in the parent chain of the component, a NoFrameWindow exception is signaled.

Methods

addButton(text=String) returns Button
Creates a button with the given text, adds it to the prompt dialog, and attaches a CloseWindow listener to it.

addButton(text=String, listener=ActionListener, closeDialog=boolean 1) returns Button
Creates a button with the given text and adds it to the prompt dialog. The given action listener is attached to the button, and a CloseWindow listener is attached to the button if *closeWindow* is true.

getText() returns Rexx
Returns the text in the entry field

setCloseBehaviour(newBehaviour=int)
: Sets the close behavior of the CloseWindow object (see “CloseWindow Class” on page 261). Possible values are CloseWindow.DESTROY, CloseWindow.HIDE, and CloseWindow.SHUTDOWN.

setKeyMode(set=Rexx)
: Sets the set of allowed characters for the entry field (see “KeyCheck Class” on page 265)

setTextLimit(limit=int)
: Limits the text field to the given number of characters (see “LimitTextField Class” on page 266)

setUpperCase(toUpperCase=boolean)
: Sets the translation mode for the entry field. If *toUpperCase* is true, all characters are translated to uppercase.

setVisible(visible=boolean)
: If *visible* is true, the prompt dialog window is made visible. The position of the window is dependent on the parent window.

PromptDialogActionListener Class

The PromptDialogActionListener class can be used to automate working with prompt dialogs. It implements an action listener that can be attached to any component in your application that fires action events.

When an action event is received, the getPromptDialog method in your application is called. This method must create and return a prompt dialog. The action listener makes the prompt dialog visible and waits for its results.

When the prompt dialog is closed successfully, the action listener calls the promptReady method in your application with the value of the text field as a parameter.

Your application has to implement the PromptDialogAction interface to make use of the PromptDialogActionListener class (see “PromptDialogAction Interface”).

Constructor

PromptDialogActionListener(application=PromptDialogAction)
: Creates an action listener. The application specified as a parameter is called back when an action event occurs.

PromptDialogAction Interface

The PromptDialogAction interface is used by the PromptDialogActionListener. Two methods are defined by the interface; one to create a prompt dialog, and one to return the value of the prompt dialog. See Figure 85 on page 149 for an example.

Methods

getPromptDialog(source=ActionListener) returns PromptDialog
: This method is invoked when a PromptDialogActionListener receives an action event from a source that is not part of its prompt dialog.

  The *source* parameter is a reference to the PromptDialogActionListener. The reference enables the application to use multiple PromptDialogActionListeners.

  The application must build a prompt dialog, attach the given action listener to the *Ok* button, and return the dialog.

promptReady(text=String, source=ActionListener)

This method is invoked when a PromptDialogActionListener reveives an action event from the prompt dialog. The *text* parameter is the value of the text field of the prompt dialog.

The *source* parameter is a reference to the PromptDialogActionListener. The reference enables the application to use multiple PromptDialogActionListeners.

RedbookUtil Class

The RedbookUtil class contains class methods that implement some shortcuts used by many Redbook package classes.

Methods

findParentFrame(aComponent=Component) returns Frame

Searches in the parent chain of the given component for a frame window. If a frame window is not found in the chain, a NoFrameWindow exception is signaled.

findParentWindow(aComponent=Component) returns Window

Searches in the parent chain of the given component for a window. If a window is not found in the chain, a NoWindow exception is signaled.

positionWindow(currentWindow=Window, xoffset=int 0, yoffset=int 0)

Sets the position of the given window to the middle of the screen plus the given offsets

positionWindow(parent=Component,currentWindow=Window,xoffset=int 20,yoffset=int 20)

Sets the position of the given window relative to the position of the window of the *parent* component plus the given offset. If the window would fall outside the screen, the position is corrected so that the window is fully visible.

sleep(ms=long)

Suspends the execution for the given number of milliseconds

SimpleGridbagLayout Class

The SimpleGridbagLayout class is a subclass of the GridbagLayout class. It is designed to make the usage of a gridbag layout manager more efficient.

The main advantage of the class is that it automatically creates the GridbagConstraints objects that are used to define the position and size of a component in a gridbag layout.

The class uses the constants defined by the GridbagConstraints class.

Constructor

SimpleGridbagLayout(aContainer=Container)

Creates an instance of the SimpleGridbagLayout class and sets the layout manager of the given container to the newly created instance

Methods

addFixSize(comp=Component, x=int, y=int) returns Component

Adds a component to the layout manager at the given position (x,y). The component does not resize. The default values for anchor and insets are used. The component is returned for further use.

addFixSize(comp=Component, x=int, y=int, newInsets=Insets, sizex=int 1, sizey=int 1, fill=int NONE, anchor=int NORTHWEST) returns Component
Adds a component to the layout manager at the given position (x,y). The component does not resize and is returned for further use.

addVarSize(comp=Component, x=int, y=int, weightx=double, weighty=double, sizex=int 1, sizey=int 1, fill=int BOTH, anchor=int NORHTWEST) returns Component
Adds a component to the layout manager at the given position (x,y). The component can be resized according to the *weightx* and *weighty* parameters. The default insets object is used for the insets. The component is returned for further use.

addVarSize(comp=Component, x=int, y=int, newInsets=Insets, weightx=double, weighty=double, sizex=int 1, sizey=int 1, fill=int BOTH, anchor=int NORHTWEST) returns Component
Adds a component to the layout manager at the given position (x,y). The component can be resized according to the *weightx* and *weighty* parameters. The component is returned for further use.

newConstraints(x=int, y=int, sizex=int, sizey=int, fill=int, anchor=int, weightx=double, weighty=double) returns GridbagConstraints
Creates and returns a GridbagConstraints object with the given parameters

setAnchor(newAnchor=int)
Sets the default anchor value to the *newAnchor* parameter

setInsets(top=int, left=int, bottom=int, right=int)
Sets the default insets object to the new values

WindowFocus Class

The WindowFocus class implements a FocusListener that sets the focus to a component of the window, whenever the window itself receives the focus.

When the window is deactivated, the component that owns the focus is stored. When the window is activated again, the focus is set to the stored component.

When the window is destroyed, the specified component is stored to receive the focus at the next activation.

Constructor

WindowFocus(aWindow=Window, focusRecipient=Component)
Creates the WindowFocus object and attaches it to the window. The focus is set to the focusRecipient component every time the window becomes visible.

WindowSupport Class

The WindowSupport class concentrates the usage of the CloseWindow class, the WindowFocus class, and the FieldSelect class in a single class.

The WindowSupport class creates and adds a CloseWindow object to the given window. The WindowSupport class uses the constants defined by the CloseWindow class.

Constructors

WindowSupport(aFrame=Frame, getFocus=Component null)
Creates the WindowSupport object for the given frame window. Adds a CloseWindow object for the frame (with SHUTDOWN as behavior) and a WindowFocus object for the focus recipient *getFocus*.

WindowSupport(aDialog=Dialog, getFocus=Component null, closeBehaviour=int DESTROY)
: Creates the WindowSupport object for the given dialog window. Adds a CloseWindow object for the dialog with the given behavior and a WindowFocus object for the focus recipient *getFocus*. When the dialog is closed, the parent of the dialog is brought to the front.

WindowSupport(parent=Window, cWindow=Window, getFocus=Component null, closeBehaviour=int DESTROY)
: Creates the WindowSupport object for *cWindow*. Adds a CloseWindow object for the window with the given behavior and a WindowFocus object for the focus recipient *getFocus*. When the window is closed, the parent window is brought to the front.

Methods

getCloseWindow() returns CloseWindow
: Returns the CloseWindow object that is used by WindowSupport. The CloseWindow object can be used as an action listener for menu items or push buttons.

setCloseBehaviour(newBehavior=int)
: Sets the close behavior of the internal CloseWindow object (see "CloseWindow Class" on page 261)

setFocusRecipient(aComponent=Component)
: Sets the focus recipient used by WindowFocus

Exceptions

The redbook package defines a basic exception class, *RedbookException*, and some subclasses of it. You can catch every exception thrown by the redbook package classes by catching the single RedbookException class.

Table 12 shows the exception classes of the redbook package.

Table 12. Exception Classes of the Redbook Package

Class	Description
RedbookException	Base class for all exceptions signaled by the redbook package
LoadImageException	The load of an image using an ImagePanel component failed
NoFrameWindow	The findFrameWindow method of the RedbookUtil class could not find any frame window in the parent chain of the given window or component.
NoWindow	The findFrameWindow method of the RedbookUtil class could not find any window in the parent chain of the given component.

Appendix B. Special Notices

This publication is intended to help programmers write Java applications by using an easy-to-use language called NetRexx. The information in this publication is not intended as the specification of any programming interfaces that are provided by NetRexx. See the PUBLICATIONS section of the IBM Programming Announcement for NetRexx for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM′s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM′s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (″vendor″) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer′s ability to evaluate and integrate them into the customer′s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

DATABASE 2
DB2
IBM
ThinkPad
VisualAge

The following terms are trademarks of other companies:

- C-bus is a trademark of Corollary, Inc.
- Java and HotJava are trademarks of Sun Microsystems, Incorporated.
- Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.
- PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.
- Pentium, MMX, ProShare, LANDesk, and ActionMedia are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.
- UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.
- Other company, product, and service names may be trademarks or service marks of others.

Appendix C. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

NetRexx and Java Documentation

Refer to “NetRexx Documentation” on page 5 for documentation about NetRexx, and to “Java Toolkit Documentation” on page 6 for documentation about the JDK.

International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 277.

- *Getting Started with VisualAge for Java*, Prentice Hall, IBM number SG24-2232 (in press)
- *Object Rexx for OS/2 Warp*, by Trevor Turton and Ueli Wahli, published by Prentice Hall, 1996, ISBN 0-13-273467-2, IBM number SG24-4586-00
- *Object Rexx for Windows 95/NT with OODialog*, by Ueli Wahli, Ingo Holder, and Trevor Turton, published by Prentice Hall, 1997, ISBN 0-13-858028-6, IBM number SG24-4825-00

Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
Application Development Redbooks Collection	SBOF-7290	SK2T-8037
Personal Systems Redbooks Collection	SBOF-7250	SK2T-8042

Other Publications

These publications are also relevant as further information sources:

- *UNIX Network Programming*, by W. Richard Stevens, published by Prentice Hall, ISBN 0-13-949876
- *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, published by Addison-Wesley Professional Computing Series, ISBN 0-201-63361
- *Exploring Java*, by Patrick Niemeyer and Joshua Peck, published by O'Reilly, ISBN 1-56592-184
- *The Java Class Libraries: An Annotated Reference*, by Patrick Chan and Rosanna Lee, published by Addison-Wesley, ISBN 0-201-63458-9
- "Communicating Sequential Processes," by C.A.R Hoare, *Communications of the ACM*, Vol. 21, No. 8, August 1978

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at http://www.redbooks.ibm.com.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **PUBORDER** — to order hardcopies in United States
- **GOPHER link to the Internet** - type GOPHER.WTSCPOK.ITSO.IBM.COM
- **Tools disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get BookManager BOOKs of redbooks, type the following command:

  ```
  TOOLCAT REDBOOKS
  ```

  To get lists of redbooks, type one of the following commands:

  ```
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET LISTSERV PACKAGE
  ```

  To register for information on workshops, residencies, and redbooks, type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1996
  ```

  For a list of product area specialists in the ITSO: type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE
  ```

- **Redbooks Web Site on the World Wide Web**

  ```
  http://w3.itso.ibm.com/redbooks
  ```

- **IBM Direct Publications Catalog on the World Wide Web**

  ```
  http://www.elink.ibmlink.ibm.com/pbl/pbl
  ```

  IBM employees may obtain LIST3820s of redbooks from this page.

- **REDBOOKS category on INEWS**
- **Online** — send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL
- **Internet Listserver**

  With an Internet e-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an e-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank). A category form and detailed instructions will be sent to you.

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (http://www.redbooks.ibm.com/redpieces.htm). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** — send orders to:

	IBMMAIL	**Internet**
In United States:	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada:	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America:	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- **Telephone orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	(long distance charges apply)
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish

- **Mail Orders** — send orders to:

IBM Publications	IBM Publications	IBM Direct Services
Publications Customer Support	144-4th Avenue, S.W.	Sortemosevej 21
P.O. Box 29570	Calgary, Alberta T2P 3N5	DK-3450 Allerød
Raleigh, NC 27626-0570	Canada	Denmark
USA		

- **Fax** — send orders to:

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States)** or **(+1)001-408-256-5422 (Outside USA)** — ask for:

  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **Direct Services** - send note to softwareshop@vnet.ibm.com

- **On the World Wide Web**

Redbooks Web Site	http://www.redbooks.ibm.com
IBM Direct Publications Catalog	http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Internet Listserver**

  With an Internet e-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an e-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank).

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (http://www.redbooks.ibm.com/redpieces.htm). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbook Order Form

Please send me the following:

Title	Order Number	Quantity

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

- Invoice to customer number
- Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Index

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

ITSO Redbook Evaluation

Creating Java Applications Using NetRexx
SG24-2216-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.com
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@vnet.ibm.com

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction _____

Please answer the following questions:

Was this redbook published in time for your needs? Yes____ No____

If no, please explain:

_____

_____

_____

_____

What other redbooks would you like to see published?

_____

_____

_____

Comments/Suggestions: **(THANK YOU FOR YOUR FEEDBACK!)**

_____

_____

_____

_____

_____

Printed in U.S.A.