NetRexx User's Guide

31st August 2000

Mike Cowlishaw

mfc@uk.ibm.com
IBM UK Laboratories

Version 2.00

Table of Contents

Introduction

This document is the User's Guide for the reference implementation of NetRexx. NetRexx is a *human-oriented* programming language which makes writing and using Java[1] classes quicker and easier than writing in Java.

In this Guide, you'll find information on:

- Installing NetRexx
- Using the NetRexx translator as a compiler, interpreter, or syntax checker
- Current restrictions.

The NetRexx documentation and software are distributed free of charge under the conditions of the IBM Employee Written Software program. If you download or use a NetRexx package you agree to the terms in the *IBM License Agreement* included in the package as the file `license.txt`.

For details of the NetRexx language, and the latest news, downloads, *etc.*, please see the NetRexx documentation included with the package or available on the World Wide Web, for example at: `http://www2.hursley.ibm.com/netrexx/`

[1] Java is a trademark of Sun Microsystems Inc.

Installation

This section of the document tells you how to unpack, install, and test the NetRexx translator package. This will install documentation, samples, and executables.

Note that to run any of the samples, or use the NetRexx translator, you must have already installed the Java runtime (and toolkit, if you want to compile NetRexx programs using the default compiler).

The NetRexx samples and translator will run on Java version 1.1.2[2] or later. Java 1.2 or later is recommended, and is required if you want to use the interpreter feature of the translator. You can test whether Java is installed, and its version, by trying the following command at a command prompt:

```
java –version
```

which should display a response similar to this:

```
java version "1.2"
Classic VM (build JDK–1.2–V, native threads)
```

For more information on Java installation:

- For OS/2, AIX, and other IBM operating systems, Java is probably already installed with your operating system. IBM Developer Kits for Java are also available separately for AIX, Linux, AS/400, OS/2, OS/390, VM/ESA, and Windows. See the *IBM Centre for Java Technology* web page at `http://www.ibm.com/java` for details. The full list of downloadable IBM Developer Kits can be found at `http://www.ibm.com/java/jdk/download/index.html`
- For other operating systems, see the *Sun Microsystems Java* web page at `http://www.javasoft.com` – or other suppliers of Java toolkits.

[2] 1.1.2 is required as the classes in the runtime and translator are now compressed, to reduce load time.

Unpacking the NetRexx package

The NetRexx package is shipped as a collection of files compressed into the file `NetRexx.zip`.

You probably know how to handle `.zip` files, but a word of caution: the packages contain directory structures, and files with "long names" (that is, not of 8.3 maximum length names) which are case-sensitive. Many utilities, including some Windows versions of **unzip**, can lose case information, truncate names, or fail to restore directories.

Unpacking the NetRexx.zip file

The most common utilities for "unzipping" are **Info–ZIP**, **WinZip**, and **PKZIP**. An **unzip** command is also included in most Linux distributions. You can also use the **jar** command which comes with all Java development kits.

Choose where you want the NetRexx directory tree to reside, and unpack the zip file in the directory which will be the parent of the NetRexx tree. Here are some tips:

- Ensure that you are unzipping to a disk that supports long file names (for example, an HPFS disk or equivalent on OS/2 or Windows).
- **Info–ZIP:** use version 5.12 (August 1994) or later. The syntax for unzipping NetRexx.zip is simply

  ```
  unzip NetRexx
  ```

  which should create the files and directory structure directly.
- **WinZip:** all versions support long file names.
- **PKZIP:** use a version that supports long file names. The syntax for unzipping NetRexx.zip is

  ```
  pkunzip –d NetRexx
  ```

  which should create the files and directory structure directly. The "`–d`" flag indicates that directory structure should be preserved.
- **Linux unzip:** use the syntax: `unzip –a NetRexx`. The "`–a`" flag will automatically convert text files to Unix format.
- **jar:** The syntax for unzipping NetRexx.zip is

  ```
  jar xf NetRexx.zip
  ```

  which should create the files and directory structure directly. The "`x`" indicates that the contents should be extracted, and the "`f`" indicates that the zip file name is specified. Note that the extension (`.zip`) is required.

After unpacking, the following directories should have been created:[3]

`NetRexx`

Root of the tree, which should contain the file `read.me.first`, which contains quick installation instructions

`NetRexx\browse`

The directory which contains documentation and sample programs and applets. To view these, point your web browser at `NetRexx\browse\netrexx.html`. You can also go straight to this User's Guide by browsing `NetRexx\browse\nrusers.html`.

`NetRexx\lib`

Contains the NetRexx compiler/interpreter classes (in `NetRexxC.jar`).

`NetRexx\runlib`

Contains the NetRexx runtime classes (in `NetRexxR.jar`). These are included in the NetRexxC.jar, so are not normally needed.

`NetRexx\netrexx\lang`

Contains the NetRexx runtime class files for access by a browser while running the applet samples.

`NetRexx\bin`

Contains sample scripts making it easier to use the compiler/interpreter. The simple test case `hello.nrx` is also included.

[3] On Unix and Linux systems, the directory separator will be "/" instead of "\".

Installing the NetRexx translator

The NetRexx package includes the NetRexx translator – a Java application which can be used for compiling, interpreting, or syntax-checking NetRexx programs. The procedure for installation is briefly as follows (full details are given later):

1. Make the translator visible to the Java Virtual Machine (JVM):

   - If you are running Java **1.2** or later, copy the file `NetRexx\lib\NetRexxC.jar` to the `jre\lib\ext` directory in the Java installation tree. The JVM will automatically find it there and make it available.
   - If you are using an earlier Java version (**1.1.2** through **1.1.8**) instead add the full path and filename of the `NetRexx\lib\NetRexxC.jar` to the CLASSPATH environment variable for your operating system.

   Note: if you have a NetRexxC.zip in your CLASSPATH from an earlier version of Rexx, remove it (NetRexxC.jar replaces NetRexxC.zip).

2. Copy all the files in the `NetRexx\bin` directory to a directory in your PATH (perhaps the `\bin` directory in the Java installation tree). This is not essential, but makes shorthand scripts and a test case available.

3. If you are running Java **1.2** or later, make the file `\lib\tools.jar` (which contains the `javac` compiler) in the Java tree visible to the JVM. You can do this either by adding its path and filename to the CLASSPATH environment variable, or by moving it to the `jre\lib\ext` directory in the Java tree.

4. Test the installation by making the `\bin` directory the current directory and issuing the following two commands exactly as written:

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

   The first of these should translate the test program and then invoke the `javac` compiler to generate the class file (`hello.class`) for the program. The second should run the program and display a simple greeting.

If you have any problems or errors in the above process, please read the detailed instructions and problem-solving tips that follow.

Installing for Java 1.1

This section gives a detailed procedure for installing NetRexx using a Java 1.1 development kit. The steps are as follows:

1. **Locate the Java home directory.** The name of the Java home directory will vary depending on the operating system you are using. Some possibilities are:

   ```
   /java
   \java11
   c:\jdk1.1.6
   ```

   It will contain directories such as "bin" and "lib".

   Note: if your Java home directory is on a CD-ROM, or on an unwritable shared disk, you'll need to leave the NetRexx files elsewhere and set up CLASSPATH and PATH environment settings to refer to them. Consult your Java toolkit and operating system documentation for details on how to do this.

2. **Add the NetRexxC.jar file to the CLASSPATH.** For Java to be able to find the NetRexx classes, you must update the CLASSPATH environment variable by adding the full path and name of the NetRexxC.jar file to the CLASSPATH setting. There will often already be a CLASSPATH variable set, possibly including a path to the standard Java classes.zip file. Specify or add the full path (disk, directories, and file specification) for NetRexxC.jar, making sure that the case of every letter is exactly right (Java is very case-sensitive). The full path might be something like:

   ```
   e:\NetRexx\lib\NetRexxC.jar
   ```

   Note: if you have a NetRexxC.zip in your CLASSPATH from an earlier version of Rexx, remove it (NetRexxC.jar replaces NetRexxC.zip).

   The procedure for setting the CLASSPATH variable depends on your operating system (and there may be more than one way), as described in the *Setting the CLASSPATH* section (see page 16).

3. **Copy the executables.** Copy all the files in the NetRexx\bin directory to a directory which is in your search PATH (perhaps the \bin directory below the Java home directory). This will allow them to be invoked simply by typing their name at a command prompt.

   The files in the bin directory should be:

   ```
   hello.nrx    -- a simple NetRexx program for testing
   NetRexxC.cmd -- the NetRexx compiler command in Rexx
   NetRexxC.bat -- similar NetRexx compiler batch script (Windows .bat)
   NetRexxC.sh  -- similar NetRexx compiler shell script for Linux and Unix
   nrc.cmd      -- shorter name for NetRexxC.cmd
   nrc.bat      -- shorter name for NetRexxC.bat
   nrc          -- shorter name for NetRexxC.sh
   ```

   The .cmd files are simple Rexx scripts for making it easier to use the translator. You don't have to use these, but they save some typing. They should require little

modification to run under the Rexx interpreter for your platform; for details of Rexx interpreters, see: `http://www2.hursley.ibm.com/rexx/`

Under Windows, the `.bat` files should serve the same purpose – they are not as flexible as the Rexx `.cmd` files, but will save typing.

Similarly, under Linux or other Unix systems, the NetRexxC.sh and nrc script simplify use of the translator. You may need to indicate these are executable, using (for example) the commands: `chmod 751 NetRexxC.sh` and `chmod 751 nrc` and (unless you used the `unzip –a` command to unpack them) you may need to run dos2unix on both of them (this converts CRLF to LF).

As an alternative to copying the files, the `NetRexx\bin` directory could be added to the PATH environment setting.

NetRexx installation is now complete. Now would be a good time to check that it works, as described in the *Testing the NetRexx Installation* section (see page 10).

Installing for Java 1.2+

This section gives a detailed procedure for installing NetRexx using a Java 1.2, or later, development kit. The steps are as follows:

1. **Locate the Java home directory.** The name of the Java home directory will vary depending on the operating system you are using. Some possibilities are:

   ```
   \java13\jre
   c:\java1.2\jre
   /opt/IBMJava2-13/jre
   ```

   It will contain directories such as "`bin`" and "`lib`".

   Note: if your Java home directory is on a CD-ROM, or on an unwritable shared disk, you'll need to leave the NetRexx files elsewhere and set up CLASSPATH and PATH environment settings to refer to them. Consult your Java toolkit and operating system documentation for details on how to do this.

2. **Copy or move the NetRexxC.jar file to the Java `lib\ext` directory.** For Java to be able to find the NetRexx classes, it's simplest to just put them (that is, the NetRexxC.jar file which contains them) in the Java extension directory. This is the `\lib\ext` directory below the Java home directory, which should already exist (if it does not, then you probably have the wrong `lib` directory – its parent should be called `jre`). The JVM will then automatically find it there when it is needed.

   Alternatively, you could add the NetRexxC.jar file to the CLASSPATH, as described in the *Installing for Java 1.1* section (see page 6).

   Note: if you have a NetRexxC.zip in your CLASSPATH from an earlier version of Rexx, remove it (NetRexxC.jar replaces NetRexxC.zip).

3. **Copy the executables.** Copy all the files in the `NetRexx\bin` directory to a directory which is in your search PATH (perhaps the `\bin` directory below the Java home directory). This will allow them to be invoked simply by typing their name at a command prompt.

   The files in the `bin` directory should be:

   ```
   hello.nrx    -- a simple NetRexx program for testing
   NetRexxC.cmd -- the NetRexx compiler command in Rexx
   NetRexxC.bat -- similar NetRexx compiler batch script (Windows .bat)
   NetRexxC.sh  -- similar NetRexx compiler shell script for Linux and Unix
   nrc.cmd      -- shorter name for NetRexxC.cmd
   nrc.bat      -- shorter name for NetRexxC.bat
   nrc          -- shorter name for NetRexxC.sh
   ```

   The `.cmd` files are simple Rexx scripts for making it easier to use the translator. You don't have to use these, but they save some typing. They should require little modification to run under the Rexx interpreter for your platform; for details of Rexx interpreters, see: `http://www2.hursley.ibm.com/rexx/`

Under Windows, the `.bat` files should serve the same purpose – they are not as flexible as the Rexx `.cmd` files, but will save typing.

Similarly, under Linux or other Unix systems, the NetRexxC.sh and nrc script simplify use of the translator. You may need to indicate these are executable, using (for example) the commands: `chmod 751 NetRexxC.sh` and `chmod 751 nrc` and (unless you used the `unzip -a` command to unpack them) you may need to run dos2unix on both of them (this converts CRLF to LF).

Alternatively, the `NetRexx\bin` directory could be added to the PATH environment setting.

4. **Add the Java tools.jar file to the CLASSPATH.** If you would like NetRexx to use the javac compiler for compilations (which is the default) then you will need to add the Sun tools collection to the classpath. This is usually found in a file called `tools.jar` in the Java `lib` directory. For example:

   ```
   set classpath=%CLASSPATH%;c:\java1.2\lib\tools.jar
   ```

   (with the path changed as appropriate) would add the tools.jar to the existing classpath in an OS/2 or Windows system.

   The procedure for setting the CLASSPATH variable depends on your operating system (and there may be more than one way), as described in the *Setting the CLASSPATH* section (see page 16).

   Alternatively, the tools.jar file could be moved to the Java `lib\ext` directory.

5. **Add the current directory to the CLASSPATH.** If it is not already there, add a reference to the current directory to the CLASSPATH, for example:

   ```
   set classpath=.;%CLASSPATH%
   ```

   This is needed if you cannot compile and run test programs from within a directory other than in the Java tree.

NetRexx installation is now complete. Now would be a good time to check that it works, as described in the *Testing the NetRexx Installation* section (see page 10).

Testing the NetRexx installation

After installing NetRexx, it is recommended that you test that it is working correctly. If there are any problems, check the *Installation Problems* section (see page 18).

To test your installation, make the directory to which you copied the executables the current directory, then (being very careful to get the case of letters correct):

1. Enter the command

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

   This should run the NetRexx compiler, which first translates the NetRexx program `hello.nrx` to the Java program `hello.java`. It then invokes the default Java compiler (`javac`), to compile the file `hello.java` to make the binary class file `hello.class`. The intermediate `.java` file is then deleted, unless an error occurred or you asked for it to be kept.[4]

   If you get errors from Java and you're running Java 1.2 or later, first re-check the final two steps in the *Installing for Java 1.2+* section (see page 8) before trying the *Installation Problems* section (see page 18).

2. Enter the command

   ```
   java hello
   ```

   This runs (interprets the bytecodes in) the `hello.class` file, which should display a simple greeting. On some systems, you may first have to add the directory that contains the `hello.class` file to the CLASSPATH setting so Java can find it.

3. With the sample scripts provided (`NetRexxC.cmd`, `NetRexxC.bat`, or `NetRexxC.sh`,), or the equivalent in the scripting language of your choice, the steps above can be combined into a simple single command such as:

   ```
   NetRexxC.sh –run hello
   ```

   This package also includes a trivial `nrc.cmd`, and matching `nrc.bat` and `nrc` scripts, which simply pass on their arguments to NetRexxC; "`nrc`" is just a shorter name that saves keystrokes, so for the last example you could type:

   ```
   nrc –run hello
   ```

   Note that scripts may be case-sensitive, and unless running the OS/2 Rexx script, you will probably have to spell the name of the program exactly as it appears in the filename. Also, to use `–run`, you may need to omit the `.nrx` extension.

   You could also edit the appropriate `nrc.cmd`, `nrc.bat`, or `nrc` script and add your favourite "default" NetRexxC options there. For example, you might want to add the `–prompt` flag (described later) to save reloading the translator before every compilation. If you do change a script, keep a backup copy so that if you install a new version of the NetRexx package you won't overwrite your changes.

---

[4] For example, by specifying the `–keep` or `–nocompile` flags.

Installing on an EBCDIC system

(Many thanks to Mark Cathcart and John Kearney for the details in this section.)

The NetRexx binaries are identical for all operating systems; the same NetRexxC.jar runs everywhere. However, during installation it is important to ensure that binary files are treated as binary files, whereas text files (such as the accompanying HTML and sample files) are translated to the local code page as required.

The simplest way to do this is to first install the package on a workstation, following the instructions above, then copy or FTP the files you need to the EBCDIC machine. Specifically:

- The NetRexxC.jar file should be copied "as-is", that is, use FTP or other file transfer with the BINARY option. The CLASSPATH should be set to include this NetRexxC.jar file.
- Other files (documentation, etc.) should be copied as Text (that is, they will be translated from ASCII to EBCDIC).

In general, files with extension `.au`, `.class`, `.gif`, `.jar`, or `.zip` are binary files; all others are text files.

For specific hints and tips for installing on OS/390, see Mark Cathcart's web site at `http://www.s390.ibm.com/corner` which includes a presentation that describes OS/390 Java and NetRexx installation. Setting the classpath might look like this:

```
export CLASSPATH=$CLASSPATH:/u/j390/j1.1.8/lib/NetRexxC.jar
```


Installation on a Linux or Unix system

The NetRexx binaries are identical for all operating systems; the same NetRexxC.jar runs everywhere, and the same installation process is used as on other systems. Some changes may be needed to text files, however (especially to the shell scripts), and there are alternatives to the "standard" installation process. Here are some tips:

- It is strongly recommended that you use the `unzip` command with the `–a` flag, if available. This will automatically convert text files to Unix text file format, while leaving binaries (such as `.jar`, `.class`, or `.gif` files) unchanged.
- If you cannot use the `unzip –a` command, you may need to take special action to use text files, such as the documentation or shell scripts. In the NetRexx package text files use a two-byte line end sequence (CRLF) whereas some Unix programs (including bash, the shell interpreter) only accept the one-byte (LF) line end sequence. Some Unix file systems convert the files automatically, but if you are getting a **No such file** or **syntax error** message from bash you probably need to use the `dos2unix` command, to convert CRLF to LF. For example: `dos2unix NetRexxC.sh`.
- File access control information is not preserved in the package. You may therefore get a a **Permission denied** message when you try and run the scripts, indicating that the files are not marked as executable. To mark them as executable, use the `chmod` command, for example: `chmod 751 NetRexxC.sh`.
- Instead of moving the files to specific locations, as suggested in the general installation instructions, you can instead link them symbolically. For example, something like:

  ```
  ln –s /usr/local/NetRexx/bin/* /usr/local/bin/.
  ```

  would link the shell scripts directory into a different `bin` directory.

Installing for Visual J++

(Many thanks to Bill Potvin and Bernhard Hurzeler for the details in this section.)

Here's how to install NetRexx for use with Visual J++:

1. Copy the following file into the [java_root]\Lib directory:

   ```
   NetRexxC.jar
   ```

   For example, if [java_root] is at E:\Java:

   ```
   C:\>copy NetRexxC.jar E:\Java\Lib
   ```

2. Similarly, copy the nrc.bat and NetRexxC.bat files to the [java_root]\Bin directory.

3. Add the jar file explicitly to the CLASSPATH:

   ```
   C:\>set CLASSPATH=%CLASSPATH%;[java_root]\Lib\NetRexxC.jar;
   ```

   For example, if [java_root] is E:\Java, your CLASSPATH might then look like this:

   ```
   CLASSPATH=E:\Java\Lib;E:\Java\TrustLib;E:\Java\Lib\NetRexxC.jar;
   ```

   Under Windows NT 4.0 and Windows 2000 this can be done using Start, Settings, Control Panel, System, Environment tab, System Variables, and clicking on CLASSPATH.

Using NetRexx with Visual J++

Using NetRexx with J++ is very similar to using it with other Java development kits; the main difference is in the command names:

1. Use the J++ jview command to invoke the NetRexx translator to convert a NetRexx program (*e.g.*, hello.nrx) into a Java program (hello.java).

   Note that some versions of the jview package do not provide a classes.zip file by default, but the NetRexx compiler needs this to determine information about classes that you use. If this is the case, you will get a *class not found* error for java.lang.Object. In this case, run the command

   ```
   clspack –auto
   ```

   from an MS-DOS prompt to create the classes.zip file.

   When running the NetRexx compiler, you must specify the –nocompile option to NetRexx in order to prevent it from trying to invoke the Java toolkit compiler javac (which isn't in the MicroSoft J++ classes).

   For example, if hello.nrx is in the current directory:

   ```
   jview COM.ibm.netrexx.process.NetRexxC hello –nocompile
   ```

   The result of this step should be a Java source file called hello.java.

2. Use the J++ "jvc" command to compile the Java source file:


```
jvc hello.java
```

The result of this step should be a Java class file called `hello.class`.

3. Execute the class file with the J++ command `jview`:

```
jview hello
```

4. Note that some earlier versions of `jview` fail with an exception (an `ArrayIndexOutOfBoundsException` in RexxUtil.translate) when compiling `hello.nrx`. This is due to a bug in the jview JIT;[5] the workaround is to turn the JIT off.

Note that all the commands above probably have to be typed exactly as shown (Java is very case-sensitive). The supplied `NetRexxC.bat` can be modified to work with the above commands by using `jview` instead of the `java` command and adding the `jvc` step.

[5] Just In Time compiler.

Installing just the NetRexx runtime

If you only want to run NetRexx programs and do not wish to compile or interpret them, or if you would like to use the NetRexx string (Rexx) classes from other languages, you can install just the NetRexx runtime classes.

To do this, follow the appropriate instructions for installing the compiler, but use the NetRexxR.jar instead of NetRexxC.jar. The NetRexxR.jar file can be found in the `NetRexx\runlib` directory.

You do not need to use or copy the executables in the `NetRexx\bin` directory.

The NetRexx class files can then be referred to from Java or NetRexx programs by importing the package "`netrexx.lang`". For example, a string might be of class "`netrexx.lang.Rexx`".

For information on the `netrexx.lang.Rexx` class and other classes in the runtime, see the *NetRexx Language* and *NetRexx Supplement* documents.

Note: If you have already installed the NetRexx translator (NetRexxC.jar) then you do not need to install NetRexxR.jar; the latter contains only the NetRexx runtime classes, and these are already included in NetRexxC.jar.

Setting the CLASSPATH

Most implementations of Java use an *environment variable* called CLASSPATH to indicate a search path for Java classes. The Java Virtual Machine and the NetRexx translator rely on the CLASSPATH value to find directories, zip files, and jar files which may contain Java classes.

The procedure for setting the CLASSPATH environment variable depends on your operating system (and there may be more than one way). Here are some examples:

- For most **Windows** installations, or for **OS/2**, use a `SET CLASSPATH=` command in AUTOEXEC.BAT (for Windows) or in CONFIG.SYS (for OS/2), and then re-boot after changing. In both cases the command syntax is the same, and might look like this:

  ```
  set classpath=.;c:\java1.2\lib\NetRexxC.jar
  ```

  In this example, the first segment of the value (before the semicolon) lets classes in the current directory be found, and the second segment includes the classes needed by the NetRexx translator. Both environments normally include the standard Java classes automatically. Under Java 1.2, you may need to add the Sun tools classes explicitly (in tools.jar, see above).

- Under **Windows NT 4.0** and **Windows 2000** the CLASSPATH should be set using Start, Settings, Control Panel, System, Environment tab, System Variables, and clicking on CLASSPATH; new command windows will then inherit the new setting immediately.

- For **Linux** and **Unix** (**BASH**, **Korn**, or **Bourne** shell), use:

  ```
  CLASSPATH=<newdir>:$CLASSPATH
  export CLASSPATH
  ```

  Changes for re-boot or opening of a new window should be placed in your `/etc/profile`, `.login`, or `.profile` file, as appropriate.

- For **Linux** and **Unix** (**C** shell), use:

  ```
  setenv CLASSPATH <newdir>:$CLASSPATH
  ```

  Changes for re-boot or opening of a new window should be placed in your `.cshrc` file.

If you are unsure of how to do this, check the documentation you have for installing the Java toolkit.

Documentation packages

The NetRexx package (NetRexx.zip) contains links to the primary NetRexx documentation in HTML format, for browsing. These documents are:

- The NetRexx Language Overview (Part 2 of *The NetRexx Language* book, updated)
- The NetRexx Language Definition 1.00 (Part 3 of *The NetRexx Language* book)
- The NetRexx Supplement (enhancements to the language since the book was published)
- The NetRexx User's Guide (this document – installation and use of the NetRexx translator).

The Overview and User's Guide are included in full in the package, so they will always be accessible locally. The other two documents are held on the NetRexx web site, `http://www2.hursley.ibm.com/netrexx`

In addition, all four documents are also available from the web site in Adobe Acrobat (PDF) format, which are linked from the package HTML files. You can also download all four PDF files at once in the documentation package (NetRexxD.zip). To access the documents:

1. Download the package from the NetRexx web site, at:
   `http://www2.hursley.ibm.com/netrexx/NetRexxD.zip`
2. Copy or move the zip file to the root directory of your choice for documentation.
3. With your chosen directory as your current directory, unzip the package, following the instructions in the *Unpacking the NetRexx package* section (see page 3).

   This should add the directory `NetRexxD` to your chosen directory, containing the PDF documentation files.
4. You can now view and print any of the files from the new directory that have the extension "`.pdf`" using Adobe's Acrobat Reader program, available from Adobe's web site at `http://www.adobe.com`

Installation Problems?

If the “hello” example described in the *Testing the NetRexx Installation* section (see page 10) doesn’t work, one of the following problems may be the cause:

- A **Can't find class COM.ibm.netrexx.process.NetRexxC...** message probably means that the NetRexxC.jar file has not been specified in your CLASSPATH setting, or is misspelled, or is in the wrong case, or (for Java 1.2 or later) is not in the Java `\lib\ext` directory. Note that in the latter case there are two `lib` directories in the Java tree; the correct one is in the Java Runtime Environment directory (`jre`).

  The *Setting the CLASSPATH* section (see page 16) contains information on setting the CLASSPATH.

- A **Can't find class hello...** message may mean that the directory with the hello.class file is not in your CLASSPATH (you may need to add a “`.;`” to the CLASSPATH, signifying the current directory), or either the filename or name of the class (in the source) is spelled wrong (the java command is [very] case-sensitive). Note that the name of the class must *not* include the `.class` extension.

- The compiler appears to work, but towards the end fails with **Exception ... NoClassDefFoundError: sun/tools/javac/Main**. This indicates that you are running Java 1.2 or later but did not add the Java tools to your CLASSPATH (hence Java could not find the javac compiler). See the *Installing for Java 1.2+* section (see page 8) for more details, and an alternative action.

  Alternatively, you may be trying to use NetRexx under Visual J++, which needs a different procedure (see page 13). You can check whether `javac` is available and working by issuing the `javac` command at a command prompt; it should respond with usage instructions.

- You have an extra blank or two in the CLASSPATH. Blanks should only occur in the middle of directory names (and even then, you probably need some double quotes around the SET command or the CLASSPATH segment with the blank). The JVM is sensitive about this.

- You are trying the `NetRexxC.sh` or `nrc` scripts under Linux or other Unix system, and are getting a **Permission denied** message. This probably means that you have not marked the scripts as being executable. To do this, use the `chmod` command, for example: `chmod 751 NetRexxC.sh`.

- You are trying the `NetRexxC.sh` or `nrc` scripts under Linux or other Unix system, and are getting a **No such file** or **syntax error** message from bash. This probably means that you did not use the `unzip -a` command to unpack the NetRexx package, so CRLF sequences in the scripts were not converted to LF.

- You didn’t install on a file system that supports long file names (for example, on OS/2 or Windows you should use an HPFS or FAT32 disk or equivalent). Like most Java applications, NetRexx uses long file names.

- You have a down-level **unzip** utility, or changed the name of the `NetRexxC.jar` file so that it does not match the spelling in the classpath. For example, check that the name of the file "`NetRexxC.jar`" is exactly that, with just three capital letters.
- You have only the Java runtime installed, and not the toolkit. If the toolkit is installed, you should have a program called `javac` on your computer. You can check whether `javac` is available and working by issuing the `javac` command at a command prompt; it should respond with usage information.
- An **Out of environment space** message when trying to set CLASSPATH under Win9x-DOS can be remedied by adding `/e:4000` to the "Cmd line" entry for the MS-DOS prompt properties (try `command /?` for more information).
- An exception, apparently in the `RexxUtil.translate` method, when compiling with Microsoft Java SDK 3.1 (and possibly later SDKs) is caused by a bug in the Just In Time compiler (JIT) in that SDK. Turn off the JIT using Start -> Settings -> Control Panel -> Internet to get to the Internet Properties dialog, then select Advanced, scroll to the Java VM section, and uncheck "Java JIT compiler enabled". Alternatively, turn of the JIT by setting the environment variable:

  ```
  SET MSJAVA_ENABLE_JIT=0
  ```

  (this can be placed in a batch file which invokes NetRexxC, if desired).
- A **java.lang.OutOfMemoryError** when running the compiler probably means that the maximum heap size is not sufficient. The initial size depends on your Java virtual machine; you can change it to (say) 24 MegaBytes by setting the environment variable:

  ```
  SET NETREXX_JAVA=-mx24M
  ```

  In Java 1.2.2 or later, use:

  ```
  SET NETREXX_JAVA=-Xmx24M
  ```

  The `NetRexxC.cmd` and `.bat` files add the value of this environment variable to the options passed to `java.exe`. If you're not using these, modify your `java` command or script appropriately.
- You have a down-level version of Java installed. NetRexxC will run only on Java version 1.1.2 (and later versions). You can check the version of Java you have installed using the command "`java -version`".
- Included in the documentation collection are a number of examples and samples (Hello, HelloApplet, etc.). To run any of these, you must have Java installed.

  Further, some of the samples must be viewed using the Java toolkit applet-viewer or a Java-enabled browser. Please see the hypertext pages describing these for detailed instructions. In general, if you see a message from Java saying:

  ```
  void main(String argv[]) is not defined
  ```

  this means that the class cannot be run using just the "java" command; it must be run from another Java program, probably as an applet.

Do you have any NetRexx problem-solving tips not covered above? If so, please let me know, at `mailto:mfc@uk.ibm.com`

Using the translator

This section of the document tells you how to use the translator package. It assumes you have successfully installed Java and NetRexx, and have tested that the `hello.nrx` testcase can be compiled and run, as described in the *Testing the NetRexx Installation* section (see page 10).

The NetRexx translator may be used as a compiler or as an interpreter (or it can do both in a single run, so parsing and syntax checking are only carried out once). It can also be used as simply a syntax checker.

When used as a compiler, the intermediate Java source code may be retained, if desired. Automatic formatting, and the inclusion of comments from the NetRexx source code are also options.

Using the translator as a Compiler

The installation instructions for the NetRexx translator describe how to use the package to compile and run a simple NetRexx program (`hello.nrx`). When using the translator in this way (as a compiler), the translator parses and checks the NetRexx source code, and if no errors were found then generates Java source code. This Java code (which is known to be correct) is then compiled into bytecodes (`.class` files) using a Java compiler. By default, the `javac` compiler in the Java toolkit is used.

This section explains more of the options available to you when using the translator as a compiler.

The translator command

The translator is invoked by running a Java program (class) which is called `COM.ibm.netrexx.process.NetRexxC` ("`NetRexxC`", for short). This can be run by using the Java interpreter, for example, by the command:

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

or by using a system-specific script (such as `NetRexxC.cmd`. or `nrc.bat`). In either case, the compiler invocation is followed by one or more file specifications (these are the names of the files containing the NetRexx source code for the programs to be compiled).

File specifications may include a path; if no path is given then NetRexxC will look in the current (working) directory for the file. NetRexxC will add the extension `.nrx` to input program names (file specifications) if no extension was given.

So, for example, to compile `hello.nrx` in the current directory, you could use any of:

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

(the first two should always work, the last two require that the system-specific script be available). The resulting `.class` file is placed in the current directory, and the `.crossref` (cross-reference) file is placed in the same directory as the source file (if there are any variables and the compilation has no errors).

Here's an example of compiling two programs, one of which is in the directory `D:\myprograms`:

```
nrc hello d:\myprograms\test2.nrx
```

In this case, again, the `.class` file for each program is placed in the current directory.

Note that when more than one program is specified, they are all compiled within the same class context. That is, they can "see" the classes, properties, and methods of the other programs being compiled, much as though they were all in one file.[6] This allows

[6] The programs do, however, maintain their independence (that is, they may have different **options**, **import**, and **package** instructions).

mutually interdependent programs and classes to be compiled in a single operation. Note that if you use the **package** instruction you should also read the more detailed *Compiling multiple programs* section (see page 23).

On completion, the NetRexxC class will exit with one of three return values: 0 if the compilation of all programs was successful, 1 if there were one or more Warnings, but no errors, and 2 if there were one or more Errors.

As well as file names, you can also specify various option words, which are distinguished by the word being prefixed with "–". These flagged words (or "flags") may be any of the option words allowed on the NetRexx **options** instruction (see the NetRexx language documentation). These options words can be freely mixed with file specifications. To see a full list of options, execute the NetRexxC command without specifying any files.

The translator also implements some additional option words, which control compilation features. These cannot be used on the **options** instruction, and are:

`-keep`

keep the intermediate `.java` file for each program. It is kept in the same directory as the NetRexx source file as `xxx.java.keep`, where `xxx` is the source file name. The file will also be kept automatically if the `javac` compilation fails for any reason.

`-nocompile`

do not compile (just translate). Use this option when you want to use a different Java compiler. The `.java` file for each program is kept in the same directory as the NetRexx source file, as the file `xxx.java.keep` (where `xxx` is the source file name).

`-noconsole`

do not display compiler messages on the console (command display screen). This is usually used with the `savelog` option.

`-savelog`

write compiler messages to the file `NetRexxC.log`, in the current directory. This is often used with the `noconsole` option.

`-time`

display translation, `javac` compile, and total times (for the sum of all programs processed).

`-run`

run the resulting Java class as a stand-alone application, provided that the compilation had no errors. (See note below.)

Here are some examples:

```
java COM.ibm.netrexx.process.NetRexxC hello –keep –strictargs
java COM.ibm.netrexx.process.NetRexxC –keep hello wordclock
java COM.ibm.netrexx.process.NetRexxC hello wordclock –nocompile
nrc hello
nrc hello.nrx
nrc –run hello
nrc –run Spectrum –keep
nrc hello –binary –verbose1
nrc hello –noconsole –savelog –format –keep
```

Option words may be specified in lowercase, mixed case, or uppercase. File specifications are platform-dependent and may be case sensitive, though NetRexxC will always prefer an exact case match over a mismatch.

Note: The `–run` option is implemented by a script (such as `nrc.bat` or `NetRexxC.cmd`), not by the translator; some scripts (such as the `.bat` scripts) may require that the `–run` be the first word of the command arguments, and/or be in lowercase. They may also require that only the name of the file be given if the `–run` option is used. Check the commentary at the beginning of the script for details.

Compiling multiple programs and using packages

When you specify more than one program for NetRexxC to compile, they are all compiled within the same class context: that is, they can "see" the classes, properties, and methods of the other programs being compiled, much as though they were all in one file.

This allows mutually interdependent programs and classes to be compiled in a single operation. For example, consider the following two programs (assumed to be in your current directory, as the files `X.nrx` and `Y.nrx`):

```
/* X.nrx */
class X
  why=Y null

/* Y.nrx */
class Y
  exe=X null
```

Each contains a reference to the other, so neither can be compiled in isolation. However, if you compile them together, using the command:

```
nrc X Y
```

then the cross-references will be resolved correctly.

The total elapsed time will be significantly less, too, as the classes on the CLASSPATH need to be located only once, and the class files used by the NetRexxC compiler or the programs themselves will also only be loaded (and JIT-compiled) once.

This example works as you would expect for programs that are not in packages. There's a restriction, though, if the classes you are compiling *are* in packages (that is, they include a **package** instruction). Currently, NetRexxC uses the `javac` compiler to generate the `.class` files, and for mutually-dependent files like these, `javac` requires that the

source files be in the Java CLASSPATH, in the sub-directory described by the **package** instruction.

So, for example, if your project is based on the tree:

```
D:\myproject
```

then if the two programs above specified a package, thus:

```
/* X.nrx */
package foo.bar
class X
  why=Y null

/* Y.nrx */
package foo.bar
class Y
  exe=X null
```

then:

1. You should put these source files in the directory: `D:\myproject\foo\bar`
2. The directory `D:\myproject` should appear in your CLASSPATH setting (if you don't do this, `javac` will complain that it cannot find one or other of the classes).
3. You should then make the current directory be `D:\myproject\foo\bar` and then compile the programs using the command "`nrc X Y`", as above.

With this procedure, you should end up with the `.class` files in the same directory as the `.nrx` (source) files, and therefore also on the CLASSPATH and immediately usable by other packages. In general, this arrangement is recommended whenever you are writing programs that reside in packages.

Notes:

1. When `javac` is used to generate the `.class` files, no new `.class` files will be created if any of the programs being compiled together had errors – this avoids accidentally generating mixtures of new and old `.class` files that cannot work with each other.
2. If a class is abstract or is an adapter class then it should be placed in the list before any classes that extend it (as otherwise any automatically generated methods will not be visible to the subclasses).

Compiling from another program

The translator may be called from a NetRexx or Java program directly, by invoking the `main` method in the `COM.ibm.netrexx.process.NetRexxC` class described as follows:

```
method main(arg=Rexx, log=PrintWriter null) static returns int
```

The `Rexx` string passed to the method can be any combination of program names and options (except `-run`), as described above. Program names may optionally be enclosed in double-quote characters (and must be if the name includes any blanks in its specification).

A sample NetRexx program that invokes the NetRexx compiler to compile a program called `test` is:

```
/* compiletest.nrx */
s='test -keep -verbose4 -utf8'
say COM.ibm.netrexx.process.NetRexxC.main(s)
```

Alternatively, the compiler may be called using the method:

```
method main2(arg=String[], log=PrintWriter null) static returns int
```

in which case each element of the `arg` array must contain either a name or an option (except `-run`, as before). In this case, names must *not* be enclosed in double-quote characters, and may contain blanks.

For both methods, the returned `int` value will be one of the return values described above, and the second argument to the method is an optional `PrintWriter` stream. If the `PrintWriter` stream is provided, translator messages will be written to that stream (in addition to displaying them on the console, unless `-noconsole` is specified). It is the responsibility of the caller to create the stream (autoflush is recommended) and to close it after calling the compiler. The `-savelog` compiler option is ignored if a `PrintWriter` is provided (the `-savelog` option normally creates a `PrintWriter` for the file `NetRexxC.log`).

Note: NetRexxC is thread-safe (the only static properties are constants), but it is not known whether `javac` is thread-safe. Hence the invocation of multiple instances of NetRexxC on different threads should probably specify `-nocompile`, for safety.

Using the prompt option

The **prompt** option may be be used for interactive invocation of the translator. This requests that the processor not be ended after a file (or set of files) has been processed. Instead, you will be prompted to enter a new request. This can either repeat the process (perhaps if you have altered the source in the meantime), specify a new set of files, or alter the processing options.

On the second and subsequent runs, the processor will re-use class information loaded on the first run. Also, the classes of the processor itself (and the javac compiler, if used) will not need to be verified and JIT-compiled again. These savings allow extremely fast processing, as much as fifty times faster than the first run for small programs.

When you specify `-prompt` on a NetRexxC command, the NetRexx program (or programs) will initially be processed as usual, according to the other flags specified. Once processing is complete, you will be prompted thus:

```
Enter new files and additional options, '=' to repeat, 'exit' to end:
```

At this point, you may enter:

- One or more file names (with or without additional flags): the previous process, modified by any new flags, is repeated using the source file or files specified. Files named previously are not included in the process (unless they are named again in the new list of names).
- Additional flags (without any new files): the previous process, modified by the new flags, is repeated, on the same files as before. Note that flags are accumulated; that is, flags are not reset to defaults between prompts.
- The character "=": this simply repeats the previous process, on the same file or files (which may have had their contents changed since the last process) and using the same flags. This is especially useful when you simply wish to re-compile (or re-interpret, see below) the same file or files after editing.
- The word "`exit`", which causes NetRexxC to cease execution without any more prompts.
- Nothing (just press Enter or the equivalent) – usage hints, including the full list of possible options, *etc.*, are displayed and you are then prompted again.

Using the translator as an Interpreter

In addition to being used as a compiler, the translator also includes a true NetRexx interpreter, allowing NetRexx programs to be run on the Java 2 (1.2) platform without needing a compiler or generating .class files.

The startup time for running programs can therefore be significantly reduced as no Java source code or compilation is needed, and also the interpreter can give better runtime support (for example, exception tracebacks are localized to the programs being interpreted, and the location of an exception will be identified often to the nearest token in a term or expression).

Further, in a single run, a NetRexx program can be both interpreted and then compiled. This shares the parsing between the two processes, so the `.class` file is produced without the overhead of re-translating and re-checking the source.

Interpreting programs

The NetRexx interpreter is currently designed to be fully compatible with NetRexx programs compiled conventionally. There are some minor restrictions (see page 35), but in general any program that NetRexxC can compile without error should run. In particular, multiple programs, threads, event listeners, callbacks, and Minor (inner) classes are fully supported.

To use the interpreter, use the NetRexxC command as usual and specify either of the following command options (flags):

`-exec`

> after parsing, execute (interpret) the program or programs by calling the static `main(String[])` method on the first class, with an empty array of strings as the argument. (If there is no suitable `main` method an error will be reported.)

`-arg words...`

> as for `–exec`, except that the remainder of the command argument string passed to NetRexxC will be passed on to the main method as the array of argument strings, instead of being treated as file specifications or flags. Specifying `–noarg` is equivalent to specifying `–exec`; that is, an empty array of argument strings will be passed to the main method (and any remaining words in the command argument string are processed normally).

When any of `–exec`, `–arg`, or `–noarg` is specified, NetRexxC will first parse and check the programs listed on the command. If no error was found, it will then run them by invoking the main method of the first class interpretively.

Before the run starts, a line similar to:

```
===== Exec: hello =====
```

will be displayed (you can stop this and other progress indicators being displayed by using the `–verbose0` flag, as usual).

Finally, after interpretation is complete, the programs are compiled in the usual way, unless `-nojava`[7] or `-nocompile` was specified.

For example, to interpret the "hello world" program without compilation, the command:

```
nrc hello -exec -nojava
```

can be used. If you are likely to want to re-interpret the program (for example, after changing the source file) then also specify the `-prompt` flag, as described above. This will give very much better performance on the second and subsequent interpretations.

Similarly, the command:

```
nrc hello -nojava -arg Hi Fred!
```

would invoke the program, passing the words "`Hi Fred!`" as the argument to the program (you might want to add the line "`say arg`" to the program to demonstrate this).

You can also invoke the interpreter directly from another NetRexx or Java program, as described in the *Using the NetRexxA API* section (see page 30).

Interpreting – Hints and Tips

When using the translator as an interpreter, you may find these hints useful:

- If you can, use the `-prompt` command line option (see above). This will allow very rapid re-interpretation of programs after changing their source.
- If you don't want the programs to be compiled after interpretation, specify the `-nojava` option, unless you want the Java source code to be generated in any case (in which case specify `-nocompile`, which implies `-keep`).
- By default, NetRexxC runs fairly "noisily" (with a banner and logo display, and progress of parsing being shown). To turn off these messages during parsing (except error reports and warnings) use the `-verbose0` flag.
- If you are watching NetRexx trace output while interpreting, it is often a good idea to use the `-trace1` flag. This directs trace output to the standard output stream, which will ensure that trace output and other output (for example, from **say** instructions) are synchronized.
- Use the NetRexx **exit** instruction (rather than the `System.exit()` method call) to end windowing (AWT) applications which are to be interpreted. This will allow the interpreter to correctly determine when the application has ended. This is discussed further in the *Interpreter restrictions* section (see page 35).

[7] The `-nojava` flag stops any Java source being produced, so prevents compilation. This flag may be used to force syntax-checking of a program while preventing compilation, and with optional interpretation.

Interpreting – Performance

The initial release of the interpreter, in the NetRexx 2.0 reference implementation, directly and efficiently interprets NetRexx instructions. However, to assure the stability of the code, terms and expressions within instructions are currently fully re-parsed and checked each time they are executed. This has the effect of slowing the execution of terms and expressions significantly; performance measurements on the initial release are therefore unlikely to be representative of later versions that might be released in the future.

For example, at present a loop controlled using "`loop for 1000`" will be interpreted around 50 times faster than a loop controlled by "`loop i=1 to 1000`", even in a binary method, because the latter requires an expression evaluation each time around the loop.

Using the NetRexxA API

As described elsewhere, the simplest way to use the NetRexx interpreter is to use the command interface (NetRexxC) with the `-exec` or `-arg` flags. There is a also a more direct way to use the interpreter when calling it from another NetRexx (or Java) program, as described here. This way is called the *NetRexxA Application Programming Interface* (API).

The `NetRexxA` class is in the same package as the translator (that is, `COM.ibm.netrexx.process`), and comprises a constructor and two methods. To interpret a NetRexx program (or, in general, call arbitrary methods on interpreted classes), the following steps are necessary:

1. Construct the interpreter object by invoking the constructor `NetRexxA()`. At this point, the environment's classpath is inspected and known compiled packages and extensions are identified.
2. Decide on the program(s) which are to be interpreted, and invoke the NetRexxA `parse` method to parse the programs. This parsing carries out syntax and other static checks on the programs specified, and prepares them for interpretation. A "stub" class is created and loaded for each class parsed, which allows access to the classes through the JVM reflection mechanisms.
3. At this point, the classes in the programs are ready for use. To invoke a method on one, or construct an instance of a class, or array, *etc.*, the Java reflection API (in `java.lang` and `java.lang.reflect`) is used in the usual way, working on the `Class` objects created by the interpreter. To locate these `Class` objects, the API's `getClassObject` method must be used.

Once step 2 has been completed, any combination or repetition of using the classes is allowed. At any time (provided that all methods invoked in step 3 have returned) a new or edited set of source files can be parsed as described in step 2, and after that, the new set of class objects can be located and used. Note that operation is undefined if any attempt is made to use a class object that was located before the most recent call to the `parse` method.

Here's a simple example, a program that invokes the main method of the hello.nrx program's class:

```
-- Try the NetRexxA interface
options binary
import COM.ibm.netrexx.process.NetRexxA

interpreter=NetRexxA()             -- make interpreter

files=['hello.nrx']                -- a file to interpret
flags=['nocrossref', 'verbose0']   -- flags, for example
interpreter.parse(files, flags)    -- parse the file(s), using the flags

helloClass=interpreter.getClassObject(null, 'hello') -- find the hello Class

-- find the 'main' method; it takes an array of Strings as its argument
classes=[interpreter.getClassObject('java.lang', 'String', 1)]
mainMethod=helloClass.getMethod('main', classes)

-- now invoke it, with a null instance (it's static) and an empty String array
values=[Object String[0]]

loop for 10    -- let's call it ten times, for fun...
  mainMethod.invoke(null, values)
end
```

Compiling and running (or interpreting!) this example program will illustrate some important points, especially if a **trace all** instruction is added near the top. First, the performance of the interpreter (or indeed the compiler) is dominated by JVM and other start-up costs; constructing the interpreter is expensive as the classpath has to be searched for duplicate classes, *etc.* Similarly, the first call to the parse method is slow because of the time taken to load, verify, and JIT-compile the classes that comprise the interpreter. After that point, however, only newly-referenced classes require loading, and execution will be very much faster.

The remainder of this section describes the constructor and the two methods of the NetRexxA class in more detail.

The NetRexxA constructor

Syntax:

```
NetRexxA()
```

This constructor takes no arguments and builds an interpeter object. This process includes checking the classpath and other libraries known to the JVM and identifying classes and packages which are available.

The parse method

Syntax:

```
parse(files=String[], flags=String[]) returns boolean
```

The parse method takes two arrays of Strings. The first array contains a list of one or more file specifications, one in each element of the array; these specify the files that are to be parsed and made ready for interpretation.

The second array is a list of zero or more option words; these may be any option words understood by the interpreter (but excluding those known only to the NetRexxC command interface, such as `time`).[8] The parse method prefixes the `nojava` flag automatically, to prevent `.java` files being created inadvertently. In the example, `nocrossref` is supplied to stop a cross-reference file being written, and `verbose0` is added to prevent the logo and other progress displays appearing.

The `parse` method returns a boolean value; this will be 1 (true) if the parsing completed without errors, or 0 (false) otherwise. Normally a program using the API should test this result an take appropriate action; it will not be possible to interpret a program or class whose parsing failed with an error.

The getClassObject method

Syntax:

```
getClassObject(package=String, name=String [,dimension=int]) returns Class
```

This method lets you obtain a Class object (an object of type `java.lang.Class`) representing a class (or array) known to the interpreter, including those newly parsed by a parse instruction.

The first argument, `package`, specifies the package name (for example, "`com.ibm.math`"). For a class which is not in a package, `null` should be used (not the empty string, `''`).

The second argument, `name`, specifies the class name (for example, "`BigDecimal`"). For a minor (inner) class, this may have more than one part, separated by dots.

The third, optional, argument, specifies the number of dimensions of the requested class object. If greater than zero, the returned class object will describe an array with the specified number of dimensions. This argument defaults to the value 0.

An example of using the `dimension` argument is shown above where the `java.lang.String[]` array Class object is requested.

Once a Class object has been retrieved from the interpreter it may be used with the Java reflection API as usual. The Class objects returned are only valid until the parse method is next invoked.

---

[8] Note that the option words are not prefixed with a "`-`".

Using NetRexx for Web applets

Web applets can be written one of two styles:

- "Lean and mean", where binary arithmetic is used, and only core Java classes (such as `java.lang.String`) are used. This is recommended for World Wide Web pages, which may be accessed by people using a slow dial-up connection. Several examples using this style are included in the NetRexx package (*e.g.*, `NervousTexxt.nrx` or `ArchText.nrx`).
- "Full-function", where decimal arithmetic is used, and advantage is taken of the full power of the NetRexx runtime (Rexx) class. This is appropriate for intranets, where most users will have fast connections to servers. An example using this style is included in the NetRexx package (`WordClock.nrx`).

If you write applets which use the NetRexx runtime (or any other Java classes that might not be on the client browser), the rest of this section may help in setting up your Web server.

A good way of setting up an HTTP (Web) server for this is to keep all your applets in one subdirectory. You can then make the NetRexx runtime classes (that is, the classes in the package known to the Java Virtual Machine as `netrexx.lang`) available to all the applets by unzipping NetRexxR.jar into a subdirectory `netrexx/lang` below your applets directory.

For example, if the root of your server data tree is

```
D:/mydata
```

then you might put your applets into

```
D:/mydata/applets
```

and then the NetRexx classes (unzipped from NetRexxR.jar) should be in the directory

```
D:/mydata/applets/netrexx/lang
```

The same principle is applied if you have any other non-core Java packages that you want to make available to your applets: the classes in a package called `iris.sort.quicksorts` would go in a subdirectory below `applets` called `iris/sort/quicksorts`, for example.

Note that with Java 1.1 or later it should be possible to use the classes direct from the NetRexxR.jar file providing that the browser being used is at a Java 1.1 level. This may also depend on your server being set up correctly. Please see the Java documentation for details.

Appendix: Current restrictions

The NetRexx translator is now functionally complete, though work continues on usability and performance improvements. As of this version there are still a number of restrictions, listed below.

Please note that the presence of an item in this section is not a commitment to remove a restriction in some future update; NetRexx enhancements are dependent on on-going research, your feedback, and available resources. You should treat this list as a "wish-list" (and please send in your wishes).

General restrictions

- The translator requires that Java **1.1.2** or later be installed. To use the interpreter functions, Java **1.2** (Java 2) is required.
- Certain forward references (in particular, references to methods later in a program from the argument list of an earlier method) are not handled by the translator. For these, try reordering the methods.

Compiler restrictions

The following restrictions are due to the use of a translator for compiling, and would probably only be lifted if a direct-to-bytecodes NetRexx compiler were built.

- Externally-visible names (property, method, and class names) cannot be Java reserved words (you probably want to avoid these anyway, as people who have to write in Java cannot refer to them), and cannot start with "`$0`".
- There are various restrictions on naming and the contents of programs (the first class name must match the program name, *etc.*), required to meet Java rules.
- The `javac` compiler requires that mutually-dependent source files be on the CLASSPATH, so it can find the source files. NetRexxC does not have this restriction, but when using `javac` for the final compilation you will need to follow the convention described in the *Compiling multiple programs and using packages* section (see page 23).
- The `symbols` option (which requests that debugging information be added to generated `.class` files) applies to all programs compiled together if any of them specify that option.
- Some binary floating point underflows may be treated as zero instead of being trapped as errors.

- When **trace** is used, side-effects of calls to `this()` and `super()` in constructors may be seen before the method and method call instructions are traced – this is because the Java language does not permit tracing instructions to be added before the call to `this()` or `super()`.
- The results of expressions consisting of the single term "`null`" are not traced.
- When a minor (inner) class is explicitly imported, its parent class or classes must also be explicitly imported, or `javac` will report that the class cannot be found.
- If you have a **loop** construct with a large number (perhaps hundreds) of instructions inside it, running the compiled class may fail with an **illegal target of jump or branch** verification error (or, under Java 1.1, simply terminate execution after one iteration of the loop). This is due to a bug in `javac`;[9] one workaround is to move some of the code out of the loop, perhaps into a private method.
- (The following problem may occur in larger methods, with Java 1.1.2; it seems to have been fixed in later versions of Java): NetRexxC does not restrict the number of local variables used or generated. However, the 1.1.2 `javac` compiler fails with unrelated error messages (such as **statement unreachable** or **variable may be uninitialized**) if asked to handle more than 63 local variables.

Interpreter restrictions

Interpreting Java-based programs is complex, and is constrained by various security issues and the architecture of the Java Virtual Machine. As a result, the following restrictions apply; these will not affect most uses of the interpreter.

- For interpretation to proceed, when any of `-exec`, `-arg`, or `-noarg` is specified, you must be running a Java 2 JVM (Java Virtual Machine). That is, the command "`java -version`" should report a version of **1.2** or later. Parsing and compilation, however, only require Java 1.1.2.
- Certain "built-in" Java classes (notably `java.lang.Object`, `java.lang.String`, and `java.lang.Throwable`) are constrained by the JVM in that they are assumed to be pre-loaded. An attempt to interpret them is allowed, but will cause the later loading of any other classes to fail with a class cast exception.
- Interpreted classes have a stub which is loaded by a private class loader. This means that they will usually not be visible to external (non-interpreted) classes which attempt to find them explicitly using reflection, `Class.forName()`, *etc.* Instead, these calls may find compiled versions of the classes from the classpath. Therefore, to find the "live" classes being interpreted, use the NetRexxA interpreter API interface (described below).
- An interpreter cannot completely emulate the actions taken by the Java Virtual Machine as it closes down. Therefore, special rules are followed to determine when an application is assumed to have ended when interpreting (that is, when any of `-exec`, `-arg`, or `-noarg` is specified):
    - If the application being interpreted invokes the `exit` method of the `java.lang.System` class, the run ends immediately (even if `-prompt` was speci-

[9] A `goto` bytecode instruction is being generated instead of a `goto_w` instruction.

fied). The call cannot be intercepted by the interpreter, and is assumed to be an explicit request by the application to terminate the process and release all resources.

In other cases, NetRexxC has to decide when the application ends and hence when to leave NetRexxC (or display the prompt, if `-prompt` was specified). The following rules apply:

- If any of the programs being interpreted contains the NetRexx **exit** instruction and the application leaves extra user threads active after the `main` method ends then NetRexxC will wait for an exit instruction to be executed before assuming the application has ended and exiting (or re-prompting).
- Otherwise (that is, there are no extra threads, or no **exit** instruction was seen) the application is assumed to have ended as soon as the `main` method returns and in this case the run ends (or the prompt is shown) immediately. This rule allows a program such as "hello world" to be run after a windowing application (which leaves threads active) without a deadlocked wait.

These rules normally "do the right thing". Applications which create windows may, however, appear to exit prematurely unless they use the NetRexx **exit** instruction to end their execution, because of the last rule.

Applications which include both thread creation and an **exit** instruction which is never executed will wait indefinitely and will need to be interrupted by an external "break" request, or equivalent, just as they would if run from compiled classes.

- Interpreting programs which set up their own security managers may prevent correct operation of the interpreter.

Index

G

H

I

J

K

L

N

O

P

R

S

T

U

V

W

Z

