NetRexx Supplement

21st December 1998

Mike Cowlishaw

mfc@uk.ibm.com
IBM UK Laboratories

Draft (1.148)

Contents

Part 1

NetRexx Language

This document is the supplement to *The NetRexx Language*.[1] Please see that book for background information about the language, collected syntax diagrams, *etc*. Page numbers in this supplement shown like [NRL 78] refer to page numbers in the book.

The supplement is in two parts:

1. New and experimental language features
2. The `Rexx` and other classes in the `netrexx.lang` package (see page 24).

The descriptions here assume that you have used NetRexx or have read an overview of the language; they also assume, and should be read in the context of, the NetRexx language definition.

This document may be found at the World Wide Web site

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

along with other NetRexx documentation and useful information.

---

[1] M. F. Cowlishaw, ISBN 0-13-806332-x, 197pp, Prentice-Hall, 1997

SECTION 1: MINOR AND DEPENDENT CLASSES

This section describes *minor classes* and *dependent classes* – features supported by the NetRexx reference implementation which are expected to be included in some future NetRexx language definition, should there be an updated definition.

Minor classes

A *minor class* in NetRexx is a class whose name is qualified by the name of another class, called its *parent*. This qualification is indicated by the form of the name of the class: the short name of the minor class is prefixed by the name of its parent class (separated by a period). For example, if the parent is called `Foo` then the full name of a minor class `Bar` would be written `Foo.Bar`. The short name, `Bar`, is used for the name of any constructor method for the class; outside the class it can only be used to identify the class in the context of the parent class (or from children of the minor class, see below).

The names of minor classes may be used in exactly the same way as other class names (types) in programs. For example, a property might be declared and initialized thus:

```
abar=Foo.Bar null   -- this has type Foo.Bar
```

or, if the class has a constructor, perhaps:

```
abar=Foo.Bar()      -- constructs a Foo.Bar object
```

Minor classes must be in the same program (and hence in the same package) as their parent. They are introduced by a **class** instruction that specifies their full name, for example:

```
class Foo.Bar extends SomeClass
```

Minor classes must immediately follow their parent class.[2]

Minor classes may have a parent which is itself a minor class, to any depth; the name and the positioning rules are extended as necessary. For example, the following classes might exist in a program:

```
class Foo
  class Foo.Bar
    class Foo.Bar.Nod
    class Foo.Bar.Pod
  class Foo.Car
```

[2] This allows compilers that generate Java source code to preserve line numbering.

As before, the children of `Foo.Bar` immediately follow their parent. The list of children of `Foo` can be continued after the children of `Foo.Bar` have all been specified.

Note that the short name (last part of the name) of a minor class may not be the same as the short name of any of its parents (a class `Foo.Bar.Foo` or a class `Foo.Bar.Bar` would be in error, for example). This allows minor classes to refer to their parent classes by their short name without ambiguity.

Constructing objects in minor classes

A parent class can construct an object of a child class in the usual manner, by simply specifying its constructor (identified by its short name, full name, or qualified name). For example, a method in the `Foo.Bar` class above could construct an object of type `Foo.Bar.Nod` using:

```
anod=Nod()
```

(assuming the `Foo.Bar.Nod` class has a constructor that takes no arguments).

Similarly, minor classes can refer to the types and constructors of any of its parents by simply using their short names. Hence, the `Foo.Bar.Nod` class could construct objects of its parents' types thus:

```
abar=Bar()
afoo=Foo()
```

(again assuming the parent classes have constructors that take no arguments).

Classes other than the parent or an immediate child must use the full name (if necessary, qualified by the package name) to refer to a minor class or its constructor.

Dependent classes

As described in the last section, minor classes provide an enhanced packaging (naming) mechanism for classes, allowing classes to be structured within packages. A stronger link between a child class and its parent is indicated by the modifier keyword **dependent** on the child class, which indicates that the child is a *dependent class*. For example:

```
class Foo.Dep dependent extends SomeClass
  method Dep   -- this is the constructor
```

An object constructed from a dependent class (a *dependent object*) is linked to the context of an object of its parent type (its *parent object*). The linkage thus provided allows the child object simplified access to the parent object and its properties.

In the example, an object of type `Foo.Dep` can only be constructed in the context of a parent object, which must be of type `Foo`.

Constructing dependent objects

A parent class can construct a dependent object in the same way as when constructing objects of other child types; that is, by simply specifying its constructor. In this case, however, the current object (`this`) becomes the parent object of the newly constructed object. For example, a method in the `Foo` class above could construct a dependent object of type `Foo.Dep` using:

```
adep=Dep()
```

(assuming the `Dep` class has a constructor that takes no arguments).

In general, for a class to construct an object from a dependent class, it must have a reference to an object of the parent class (which will become the parent of the new object), and the constructor must be called (by its short name) in the context of that parent object. For example:

```
parentObject=Foo()
adep=parentObject.Dep()
```

(In the same way, the first example could have been written:

```
adep=this.Dep()
```

within the parent class the `this.` is implied.)

In order to subclass a dependent class, the constructor of the dependent class must be invoked by the subclass constructor in a similar manner. In this case, a qualified call to the usual special constructor `super` is used, for example:

```
class ASub extends Foo.Dep
  method Asub(afoo=Foo)
    afoo.super()
```

The qualifier (`afoo` in this case) must be the name of an argument to the constructor. The call must be the first instruction in the method, as usual, and it must be present (it will not be generated automatically by the compiler).

Access to parent objects and their properties

Dependent classes have simplified access to their parent objects and their properties. In particular:

- The special word `parent` may be used to refer to the parent object of the current object. It can only be used in a dependent class and in contexts where the special word `this` could be used (that is, as a reference, after the current object has been constructed).

- In general, any of the objects in the chain of parents of a dependent object may be referred to by qualifying the special word `this` with the short name of the parent class. For example, extending the previous example, if the class `Foo.Dep.Ent` was a dependent class it could contain references to `Foo.this` (the parent of its parent) or `Dep.this` (the latter being the same as specifying `parent`). If preferred, the full name or the fully qualified name of the parent class may be used instead of the short name.

  Like `parent`, this construct can only be used in a dependent class and in contexts where the special word `this` could be used.

- As usual, properties external to the current class must always be qualified in some way (for example, the prefix `parent.` can be used in a term such as `parent.aprop`).

Restrictions

Minor classes may have any of the attributes (**public**, **interface**, *etc.*) of other classes, and behave in every way like other classes, with the following restrictions:

- If a class is a static class (that is, it contains only static or constant properties and methods) then any children cannot be dependent classes (because no object of the parent class can be constructed). Similarly, interface classes and abstract classes cannot have dependent classes.
- Dependent classes may not be interfaces.
- Dependent classes may not contain static or constant properties (or methods).[3] These must be placed in a parent which is not a dependent class.
- Minor classes may be public only if their parent is also public. (Note that this is the only case where more than one public class is permitted in a program.) In general: a minor class cannot be more visible than its parent.

[3] This restriction allows compilation for the Java platform.

SECTION 2: OTHER LANGUAGE ENHANCEMENTS

This section describes additional features supported by the NetRexx reference implementation that are expected to be included in some future NetRexx language definition, should there be an updated definition.

Adapter classes

The **class** instruction [NRL 74] is used to introduce a class. In NetRexx 1.0, this instruction could specify a *modifier* to indicate that it is **abstract**, **final**, or an **interface**. An alternative keyword is supported in NetRexx 1.1, **adapter**, which indicates that the class is an *adapter class.* For example:

```
class Macavity adapter implements MouseListener
```

An *adapter class* is a class that is guaranteed to implement all unimplemented abstract methods of its superclasses and interface classes that it inherits or lists as implemented on the **class** instruction.

If any unimplemented methods are found, they will be automatically generated by the language processor. Methods generated in this way will have the same visibility and signature as the abstract method they implement, and if a return value is expected then a default value is returned (as for the initial value of variables of the same type: that is, `null` or, for values of primitive type, an implementation-defined value, typically 0). Other than possibly returning a value, these methods are empty; that is, they have no side-effects.

An adapter class provides a concrete representation of its superclasses and the interface classes it implements. As such, it is especially useful for implementing event handlers and the like, where only a small number of event-handling methods are needed but many more might be specified in the interface class that describes the event model.[4]

An adapter class cannot be an interface or an abstract class (or have any abstract methods), and cannot be a final class.

Array initializers

A new form of *simple term* [NRL 41] is defined: the *array initializer.* The array initializer is recognized if it does not immediately follow (abut) a symbol, and has the form[5]

'[' *expression* [, *expression*] ... **']'**

---

4 For example, see the “Scribble” sample in the NetRexx package.

5 The notations **'['** and **']'** indicate square brackets appearing in the NetRexx program.

An array initializer therefore comprises a list of one or more expressions, separated by commas, within brackets. When an array initializer is evaluated, the expressions are evaluated in turn from left to right, and all must result in a value. An *array* [NRL 71] is then constructed, with a number of elements equal to the number of expressions in the list, with each element initialized by being assigned the result of the corresponding expression.

The type of the array is derived by adding one dimension to the type of the result of the first expression in the list, where the type of that expression is determined using the same rules as are used to select the type of a variable when it is first assigned a value [NRL 65]. All the other expressions in the list must have types that could be assigned to the chosen type without error.

For example, in

```
var1=['aa', 'bb', 'cc']
var2=[char 'a', 'b', 'c']
var3=[String 'a', 'bb', 'c']
var4=[1, 2, 3, 4, 5, 6]
var5=[[1,2], [3,4]]
```

the types of the variables would be `Rexx[]`, `char[]`, `String[]`, `Rexx[]`, and `Rexx[,]` respectively. In a binary class in the reference implementation, the types would be `String[]`, `char[]`, `String[]`, `int[]`, and `int[,]`.

Array initializers are most useful for initializing properties and variables, but like other simple terms, they may start a compound term. So, for example

```
say [1,1,1,1].length
```

would display `4`.

Notes:

1. An array of length zero cannot be constructed with an array initializer, as its type would be undefined. An explicitly typed array constructor (for example, `int[0]`) must be used.
2. Array initializers require Java 1.1.

Array partial terms

If a partial term results in a dimensioned array, its type is treated as type `Object` so that evaluation of the term can continue [NRL 46]. For example, in

```
ca=char[] "tosh"
say ca.toString()
```

the variable `ca` is an array of characters; in the expression on the second line the method `toString()` of the class `Object` will be found. (In the reference implementation, this would return an identifier for the object.)

Binary methods

The **method** instruction [NRL 93] is used to introduce a method. The **binary** keyword may be added to the instruction to indicate that the method is a *binary method*.

In binary methods, literal strings and numeric symbols are assigned native string or binary (primitive) types, rather than NetRexx types, and native binary operations are used to implement operators where possible. When **binary** is not in effect (the default), terms in expressions are converted to NetRexx types before use by operators. The section *Binary values and operations* [NRL 142] describes the implications of binary methods and classes in detail.

Notes:

1. Only the instructions inside the body of the method are affected by the **binary** keyword; any arguments and expressions on the method instruction are not affected (this ensures that a single rule applies to all the method signatures in a class).
2. All methods in a binary class are binary methods; the **binary** keyword on methods is provided for classes in which only the occasional method needs to be binary (perhaps for performance reasons). It is not an error to specify **binary** on a method in a binary class.

copyIndexed(source)

copies the collection of indexed sub-values [NRL 70] of *source* into the collection associated with *string*, and returns the modified *string*. The resulting collection is the union of the two collections (that is, it contains the indexes and their values from both collections). If a given index exists in both collections then the sub-value of *string* for that index is replaced by the sub-value from *source*.

The non-indexed value of *string* is not affected.

Example:

Following the instructions:

```
foo='def'
foo['a']=1
foo['b']=2
bar='ghi'
bar['b']='B'
bar['c']='C'
merged=foo.copyIndexed(bar)
```


then:

```
merged['a'] == '1'
merged['b'] == 'B'
merged['c'] == 'C'
merged['d'] == 'def'
```

Deprecation

Classes, methods, and properties may be designated as *deprecated*, which implies that a better alternative is available and documented. A compiler can use this information to warn of out-of-date or other use that is not recommended.

The **class** instruction [NRL 74] is used to introduce a class. The **deprecated** keyword may be added to the instruction to indicate that the class (and all its methods and properties) is deprecated. For example:

```
class action deprecated
```

The **method** instruction [NRL 93] is used to introduce a method. The **deprecated** keyword may be added to the instruction to indicate that the method is deprecated. For example:

```
method madness deprecated
```

Note that individual methods in interface classes cannot be deprecated; the whole class should be deprecated in this case.

The **properties** instruction [NRL 105] is used to define the attributes of following *property* variables. The **deprecated** keyword may be added to the instruction to indicate that the following properties are deprecated. For example:

```
properties public deprecated
  gaol=Months 11
  jail=Days 12
```

Note: In the reference implementation, the intermediate `.java` file must be compiled with a Java 1.1 compiler for the information about deprecation to be reflected in the resulting `.class` file.

Dollar sign in symbols

The dollar sign character ("`$`") may now be used in symbols (for instance, variable names). It is recommended that it only be used in mechanically generated programs or where otherwise essential.

Euro currency character

The euro character ('\u20ac') is now treated in the same way as the dollar character (that is, it may be used in the names of variables and other identifiers). It is recommended that it only be used in mechanically generated programs or where otherwise essential.

Note that only UTF8-encoded source files can currently use the euro character, and a 1.1.7 (or later) version of a Java compiler is needed to generate the class files.

Hexadecimal and binary numeric symbols

Numeric symbols (symbols in a NetRexx source program that start with a digit, [NRL 35]) may now be expressed in a hexadecimal or binary notation.

A *hexadecimal numeric symbol* describes a whole number, and is of the form *n*`X`*string*. Here, *n* is a simple number with no decimal part (and optional leading insignificant zeros) which describes the effective length of the hexadecimal string, the `X` (which may be in lowercase) indicates that the notation is hexadecimal, and *string* is a string of one or more hexadecimal characters (characters from the ranges “a-f”, “A-F”, and the digits “0-9”).

The *string* is taken as a signed number expressed in *n* hexadecimal characters. If necessary, *string* is padded on the left with `'0'` characters (note, not “sign-extended”) to length *n* characters.

If the most significant (left-most) bit of the resulting string is zero then the number is positive; otherwise it is a negative number in twos-complement form. In both cases it is converted to a NetRexx number which may, therefore, be negative. The result of the conversion is a number comprised of the Arabic digits 0-9, with no insignificant leading zeros but possibly with a leading `'-'`.

The value *n* may not be less than the number of characters in *string*, with the single exception that it may be zero, which indicates that the number is always positive (as though *n* were greater than the the length of *string*).

Examples:

```
1x8    == -8
2x8    == 8
2x08   == 8
0x08   == 8
0x10   == 16
0x81   == 129
2x81   == -127
3x81   == 129
4x81   == 129
04x81  == 129
16x81  == 129
4xF081 == -3967
8xF081 == 61569
0Xf081 == 61569
```

A *binary numeric symbol* describes a whole number using the same rules, except that the identifying character is `B` or `b`, and the digits of *string* must be either `0` or `1`, each representing a single bit.

Examples:

```
1b0    == 0
1b1    == -1
0b10   == 2
0b100  == 4
4b1000 == -8
8B1000 == 8
```

Note: Hexadecimal and binary numeric symbols are a purely syntactic device for representing decimal whole numbers. That is, they are recognized only within the source of a NetRexx program, and are not equivalent to a literal string with the same characters within quotes.

Imports, automatic (clarification)

In the reference implementation, the fundamental NetRexx and Java package hierarchies are imported by default [NRL 82], as though the instructions:

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

had been executed before the program begins.

In addition, classes in the current (working) directory are imported if no **package** instruction is specified. If a **package** instruction is specified then all classes in that package are imported.

[The Java packages are now listed explicitly, because new packages added in Java 1.1 or Java 1.2 are not included in the automatic import list.]

Imports, explicit

When a class is imported explicitly, for example, using

```
import java.awt.List
```

this indicates that the short name of the class (`List`, in this example) may be used to refer to the class unambiguously. That is, using this short name will not report an ambiguous reference warning (as it would without the **import** instruction, because a `java.util.List` class has been added in Java 1.2).

It follows that:

- Two classes imported explicitly cannot have the same short name.
- No class in a program being compiled can have the same short name as a class that is imported explicitly.

because in either of these situations a use of the short name would be ambiguous.

Numeric enhancement

The **numeric** instruction [NRL 98] is used to change the way in which arithmetic operations are carried out by a program.

One or more **numeric** instructions may now be placed before the first **class** instruction in a program; they no longer imply the start of a class. The settings they make then apply to all classes in the program (except interface classes), as though the **numeric** instructions were placed immediately following the **class** instruction in each class.

Select case

The **select** instruction [NRL 108] is used to conditionally execute one of several alternatives.

A new keyword, **case**, has been added to the **select** instruction; it must follow any **label** or **protect** phrase, and must be followed by an expression.

When **case** is used, the expression following it is evaluated at the start of the **select** construct. The result of the expression is then compared, using the strict equality operator (==), to the result of evaluating the expression in each of the **when** clauses in turn until a match is found. As usual, if no match is found then control will pass to the instruction list (if any) following **otherwise**, and in this situation the absense of an **otherwise** is a run-time error.

For example, in:

```
i=1
select case i+1
  when 1 then say 'one'
  when 2 then say 'two'
  when 3 then say 'three'
end
```

the message displayed would be “`two`”.

Notes:

1. When **case** is used, the result of evaluating the expression following each **when** no longer has to be 0 or 1. Instead, it must be possible to compare each result to the result of the **case** expression.
2. The **case** expression is evaluated only on entry to the **select** construct; it is not re-evaluated for each **when** clause.
3. An exception raised during evaluation of the **case** expression will be caught by a suitable **catch** clause in the construct, if one is present. Similarly, evaluation of the **case** expression is protected by the **protect** phrase, if one is present.

Shared classes, properties, and methods

Classes, properties, and methods have a specified *visibility*. A new form of visibility, known as *shared*, is available in NetRexx. A shared class, property, or method is one that is visible within the current package but is not visible outside the package. Shared properties and methods cannot be inherited by classes outside the package.

The **properties** instruction [NRL 105] is used to define the attributes of following *property* variables. The *visibility* for properties may include a new alternative, **shared**, which indicates that the following properties are shared and therefore cannot be inherited by classes outside the package. For example:

```
properties shared constant
  hours=int 24
  minutes=int 60
```

The **method** instruction [NRL 93] is used to introduce a method. The *visibility* for methods may include a new alternative, **shared**, which indicates that the method is shared and therefore cannot be inherited by classes outside the package. For example:


```
-- This method is only visible to the current package
method inpackage shared
```

The **class** instruction [NRL 74] is used to introduce a class. The **shared** keyword on the class instruction means exactly the same as the keyword **private**, and is accepted for consistency with the other meanings of **shared**. For clarity, it is recommended that, if required, **shared** be specified rather than **private**.

Special names

The following special names are added to the list [NRL 118]:

`class`

The object of type `Class` that describes a specific type. This word is only recognized as the second part of a compound term, where the evaluation of the first part of the term resulted in a type or qualified type.

Example:

```
obj=String.class
say obj.isInterface /* would say '0' */
```

Note that this special name requires Java 1.1.

`sourceline`

The line number of the first token [NRL 33] of the current clause in the NetRexx program, returned as a string of type `Rexx`. This will be one or more Arabic numerals, with no leading blanks, zeros, or sign, and no trailing blanks or exponent.

`sourceline` can only appear alone, or at the start of a compound term.

Trace enhancements

The **trace** instruction [NRL 111] is used to control the tracing of the execution of NetRexx methods. Three enhancements have been made to tracing:

1. The **trace** instruction has a new option, **var**, which allows variables to be traced selectively. This option is described in detail in the next section.
2. One or more **trace** instructions may now be placed before the first **class** instruction in a program; they no longer imply the start of a class. The settings they make (both overall and **trace var** selections) then apply to all classes in the program (except interface classes), as though the **trace** instructions were placed immediately following the **class** instruction in each class.

3. If a trace line is produced in a different context (program or thread) from the preceding trace line (if any) then a *trace context* line is shown. This shows the name of the program that produced the trace line, and also the name of the thread (and thread group) of the context.

   The thread group name is not shown if it is `main`, and in this case the thread name is then also suppressed if its name is `main`.

Trace Var

The **trace** instruction [NRL 111] is used to control the tracing of the execution of NetRexx methods. A new option, **var**, is now available.

The **trace var** instruction adds names to a list of monitored variables; it can also remove names from the list. If the name of a variable in the current class or method is in the list, then **trace results** is turned on for any assignment, **loop**, or **parse** instruction that assigns a new value to the named variable.

Variable names are specified by listing them after the **var** keyword. Each name may be optionally prefixed by a + or a − sign. A + sign indicates that the variable is to be added to the list of monitored variables (the default), and a − sign indicates that the variable is to be removed from the list. Blanks may be added before and after variable names and signs to separate the tokens and to improve readability.

For example:

```
trace var a b c
-- now variables a, b, and c will be traced
trace var -b -c d
-- now variables a and d will be traced
```

Notes:

1. Names in the list following the **var** keyword are simple symbols that name variables in the current class or current method. The variables may be properties, method arguments, or local variables, and may be of any type, including arrays. The names are not case-sensitive; any variables whose names match, independent of case, will be monitored.

2. No variable name can appear more than once in the list on one **trace var** instruction. However, it is not an error to add the name of a variable which does not exist or is not then assigned a value. Similarly, it is not an error to remove a name which is not currently being monitored.

3. One or more **trace var** instructions (along with one other **trace** instruction) are allowed before the first method in a class. They all modify an initial list of monitored variables which is then used for all methods in the class. Similarly, **trace var** instructions are allowed before the first class in a program, in which case they apply to all classes (except interface classes).

4. Other **trace** instructions do not affect the list of monitored variables. The **trace off** instruction may be used to turn off tracing completely; in this case **trace var** (with or without any variable names) will then turn the tracing of variables back on, using the current (or modified) variable list.

5. For a **parse** instruction, only monitored variables have their assignments traced (unless **trace results** is in effect).

Transient properties

The **properties** instruction [NRL 105] is used to define the attributes of following *property* variables. The *modifier* for properties may include a new alternative: **transient**. Properties with this modifier are known as *transient properties*. For example:

```
properties public transient
  wayfarer=int 7
```

Transient properties are properties which should not be saved when an instance of the class is saved (made persistent).

Type operation enhancement

A type on the left hand side of an operator that could be a prefix operator (+, −, or \) is now assumed to imply a cast (type concatentation) after the prefix operator is applied to the right-hand operand, rather than being an error.

For example:

```
x=int -y
```

now means that `-y` is evaluated, and then the result is cast to `int` before being assigned to `x`. It would previously need to have been written `x=int (-y)` to avoid being treated as subtracting `y` from the type `int` (which is not possible).

SECTION 3: NEW AND ENHANCED OPTIONS

The **options** instruction is used to pass special requests to the language processor (for example, an interpreter or compiler). See [NRL 100].

This section describes new or enhanced *option words* supported by the NetRexx reference implementation. Unless stated otherwise, these may be used either with an **options** instruction or as "flags" passed to the implementation with a leading "–".

comments

Comments from the NetRexx source program will be passed through to the the Java output file (which may be saved with a `.java.keep` extension by using the **–keep** command option).

Line comments become Java line comments (introduced by "`//`"). Block comments become Java block comments (delimited by "`/*`" and "`*/`"), with nested block comments having their delimiters changed to "`(-`" and "`-)`").

console

Requests that compiler messages be written to console (the default). Use **–noconsole** to prevent messages being written to the console.

This option must be used only as a compiler option, and applies to all programs being compiled.

decimal

Decimal arithmetic may be used in the program. If **nodecimal** is specified, the language processor will report operations that use (or, like normal string comparison, might use) decimal arithmetic as an error. This option is intended for performance-critical programs where the overhead of inadvertent use of decimal arithmetic is unacceptable.

explicit

Requires that all local variables must be explicitly declared (by assigning them a type but no value) before assigning any value to them. This option is intended to permit the enforcement of "house styles" (but note that the NetRexx compiler always checks for variables which are referenced before their first assignment, and warns of variables which are set but not used).

savelog

Requests that compiler messages be written to the file NetRexxC.log in the current directory. The messages are also displayed on the console, unless **–noconsole** is specified.

This option must be used only as a compiler option, and applies to all programs being compiled.

sourcedir

Requests that all `.class` files be placed in the same directory as the source file from which they are compiled. Other output files are already placed in that directory. Note that using this option will prevent the **–run** command option from working unless the source directory is the current directory.

strictargs

Requires that method invocations always specify parentheses, even when no arguments are supplied. Also, if **strictargs** is in effect, method arguments are checked for usage – a warning is given if no reference to the argument is made in the method.

strictimport

Requires that all imported packages and classes be imported explicitly using **import** instructions. That is, if in effect, there will be no automatic imports (see page 11), except those related to the **package** instruction.

This option must be used only as a compiler option, and applies to all programs being compiled.

symbols

Symbol table information (names of local variables, *etc.*) will be included in any generated `.class` file. This option is provided to aid the production of classes that are easy to analyse with tools that can understand the symbol table information. The use of this option increases the size of `.class` files.

trace, traceX

If given as **trace**, **trace1**, or **trace2**, then **trace** instructions are accepted. The trace output is directed according to the option word: **trace1** requests that trace output is written to the standard output stream, **trace** or **trace2** imply that the output should be written to the standard error stream (the default).

If **notrace** is given, then trace instructions are ignored. The latter can be useful to prevent tracing overheads while leaving **trace** instructions in a program.

utf8

As of NetRexx 1.1 (July 1997), this option must be used as a compiler option, and applies to all programs being compiled. If present on an **options** instruction, it is checked and must match the compiler option (this allows processing with or without the **utf8** option to be enforced).

Prefixing any of the above with "**no**" turns the selected option off.

The default settings of these options are:


```
nocomments console decimal noexplicit nosavelog
nosourcedir nostrictargs nostrictimport nosymbols
trace2 noutf8
```

SECTION 4: EXPERIMENTAL ENHANCEMENTS

This section describes features supported by the NetRexx reference implementation that are *experimental*. That is, they will not necessarily be included in a future NetRexx language definition, should there be an updated definition.

JavaBean properties

Almost all JavaBeans will have *properties*, which are data items that a user of a JavaBean is expected to be able to customize (for example, the text on a pushbutton). The names and types of the properties of a JavaBean are inferred from "*design patterns*" (in this context, conventions for naming methods) or from PropertyDescriptor objects associated with the JavaBean. The JavaBean properties do not necessarily correspond to instance variables in the class – though very often they do. The JavaBean specification does not guarantee that JavaBean properties that can be set can also be inspected, nor does it describe how ambiguities of naming and method signatures are to be handled.

The NetRexxC compiler (as of 15 Feb 1997) allows a more rigorous treatment of JavaBean properties, by allowing an optional attribute of properties in a class that declares them to be *indirect properties.* Indirect properties are properties of a known type that are private to the class, but which are expected to be publicly accessible indirectly, though certain conventional method calls.

Declaring properties to be indirect offers the following advantages:

- For many simple cases, the access methods for the properties can be generated automatically; there is no need to explicitly code them in the source file for the class. This is especially helpful for Indexed Properties (where four methods are needed, in general).
- Where access methods are explicitly provided in the class, they can be checked for correct form, signature and accessibility. This detects errors at compile time that otherwise would only be determined by testing.
- Similarly, attention can be drawn to the presence of methods that may be intended to be an access method for an indirect property, but will not be recognized as such by builders.

The next section describes the use of indirect properties in more detail.

Indirect properties

The **properties** instruction [NRL 105] is used to define the attributes of following *property* variables. The *visibility* of properties may include a new alternative: **indirect**. Properties with this form of visibility are known as *indirect properties*. These are properties of a known type that are private to the class, but which are expected to be publicly accessible indirectly, though certain conventional method calls.

For example, consider the simple program:

```
class Sandwich extends Canvas implements Serializable
  properties indirect
    slices=Color.gray
    filling=Color.red

  method Sandwich
    resize(100,30)

  method paint(g=Graphics)
    g.setColor(slices)
    g.fillRect(0, 0, size.width, size.height)
    g.setColor(filling)
    g.fillRect(12, 12, size.width-12, size.height-12)
```

This declares the `Sandwich` class as having two indirect properties, called `slices` and `filling`, both being of type `java.awt.Color`.

In the example, no access methods are provided for the properties, so the compiler will add them. By implementation-dependent convention, the names are prefixed with verbs such as `get` and `set`, *etc.*, and have the first character of their name uppercased to form the method names. Hence, in this Java-based example, the following four methods are added:

```
method getSlices  returns java.awt.Color
  return slices
method getFilling returns java.awt.Color
  return filling
method setSlices($1=java.awt.Color)
  slices=$1
method setFilling($2=java.awt.Color)
  filling=$2
```

(where `$1` and `$2` are “hidden” names used for accessing the method arguments).

Note that the **indirect** attribute for a property is an alternative to the **public**, **private**, and **inheritable** attributes. Like private properties, indirect properties can only be accessed directly by name from within the class in which they occur; other classes can only access them using the access methods (or other methods that may use, or have a side-effect on, the properties).

Indirect properties may be **constant** (implying that only a `get` method is generated or allowed, though the private property may be changed by methods within the class) or **transient** (see page 16). They may not be **static** or **volatile**.

In detail, the rules used for generating automatic methods for a property whose name is `xxxx` are as follows:

1. A method called `getXxxx` which returns the value of the property is generated. The returned value will have the same type as `xxxx`.
2. If the type of `xxxx` is `boolean` then the generated method will be called `isXxxx` instead of `getXxxx`.
3. If the property is not **constant** then a method for setting the property will also be generated. This will be called `setXxxx`, and take a single argument of the same type as `xxxx`. This assigns the argument to the property and returns no value.

If the property has an array type (for example, `char[]`), then it must only have a single dimension. Two further methods may then be generated, according to the rules:

1. A method called `getXxxx` which takes a single `int` as an argument and which returns an item from the property array is generated. The returned value will have the same type as `xxxx`, without the `[]`. The integer argument is used to index into the array.
2. As before, if the result type of the method would be `boolean` then the name of the method will be `isXxxx` instead of `getXxxx`.
3. If the property is not **constant** then a method for setting an item in the property array will also be generated. This will be called `setXxxx`, and take two arguments: the first is an `int` that is used to select the item to be changed, and the second is an undimensioned argument of the same type as `xxxx`. It assigns the second argument to the item in the property array indexed by the first argument, and returns no value.

For example, for an indirect property declared thus:

```
properties indirect
  fred=foo.Bar[]
```

the four methods generated would be:

```
method getFred returns foo.Bar[]; return fred
method getFred($1=int) returns foo.Bar; return fred $1”
method setFred($2=foo.Bar[]); fred=$2
method setFred($3=int, $4=foo.Bar); fred $3”=$4
```

Note that in all cases a method will only be generated if it would not exactly match a method explicitly coded in the current class.

Explicit provision of access methods

Often, for example when an indirect property has an on-screen representation, it is desirable to redraw the property when the property is changed (and in more complicated cases, there may be interactions between properties). These and other actions will require extra processing which will not be carried out by automatically generated methods. To add this processing the access methods will have to be coded explicitly. In the "Sandwich" example, we only need to supply the `set` methods, perhaps by adding the following to the example class above:

```
method setSlices(col=Color)
  slices=col      -- update the property
  this.repaint    -- redraw the component

method setFilling(col=Color)
  filling=col
  this.repaint
```

If we add these two methods, they will no longer be added automatically (the two `get` methods will continue to be provided automatically, however). Further, since the names match possible access methods for properties that are declared to be indirect, the compiler will check the method declaration: the method signatures and return type (if any) must be correct, for example. Also, since the names of access methods are case-sensitive (in a Java environment), you will be warned if a method appears to be intended to be an access method but the case of one or more letters is wrong.

Specifically, the checks carried out are as follows:

1. For methods whose names exactly match a potential access method for an indirect property (that is, start with `is`, `get`, or `set`, which is then followed by the name of an indirect property with the first character of the name uppercased):
   - The argument list for (signature of) the method must match one of those that could possibly be automatically generated for the property.
   - The returns type (if any) must match the expected returns type for that method.
   - If the returns type is simply `boolean`, then the method name must start with `is`. Conversely, if the method name starts with `is` then the returns type must be just `boolean`.
   - If the property is **constant** then the name of the method cannot start with `set`.
   - A warning is given if the method is not **public** (the default).

2. For methods whose names match a potential access method, as above, except in case:
    - A warning is given that the method in question may be intended to be an indirect property access method, but will not be recognized as such by builders.

These checks detect a wide variety of errors at compile time, hence speeding the development of classes that use indirect properties.

Part 2

The netrexx.lang package

This part of the supplement documents the `netrexx.lang` package, which includes the classes used for creating string objects of type `Rexx` along with several classes that are often used while running NetRexx programs.

This section describes the public methods and properties of these classes, as implemented by the reference implementation. It does not document those "built-in" methods of the `Rexx` class that form part of the NetRexx language [NRL 148], or those classes and methods that are internal "helper" components (which, for example, are used as repositories for rarely-executed code).

The classes in the netrexx.lang package are:

- The Exception classes (see page 25)
- Rexx (see page 26)
- RexxIO (helper class, for **say** and **ask**)
- RexxNode (helper class, for indexed strings)
- RexxOperators interface (see page 34)
- RexxParse (helper class, for **parse**)
- RexxSet (see page 35)
- RexxTrace (helper class, for **trace**)
- RexxUtil (helper class, for the Rexx class)
- RexxWords (helper class, for the Rexx class)

SECTION 1: EXCEPTION CLASSES

The classes provided for exceptions in the `netrexx.lang` package are all subclasses of `java.lang.RuntimeException` and all have the same content. Each has two constructors: one taking no argument and the other taking a string of type `java.lang.String`, which is used for additional detail describing the exception.

The Exceptions are signalled as follows.

BadArgumentException

signalled when an argument to a method is incorrect.

BadColumnException

signalled when a column number in a parsing template is not valid (for example, not a number).

BadNumericException

signalled when a **numeric digits** instruction tries to set a value that is not a whole number, or is not positive, or is more than nine digits.

DivideException

signalled when an error occurs during a division. This may be due to an attempt to divide by zero, or when the intermediate result of an integer divide or remainder operation is not valid.

ExponentOverflowException

signalled when the exponent resulting from an operation would require more than nine digits.

NoOtherwiseException

signalled when a **select** construct does not supply an **otherwise** clause and none of expressions on the **when** clauses resulted in `'1'`.

NotCharacterException

signalled when a conversion from a string to a single character was attempted but the string was not exactly one character long.

NotLogicException

signalled when a conversion from a string to a boolean was attempted but the string was neither the string `'0'` nor the string `'1'`.

Other exceptions, from the `java.lang` package, may also be signalled, for example `NumberFormatException` or `NullPointerException`.

SECTION 2: THE REXX CLASS

The class `netrexx.lang.Rexx` implements the NetRexx string class, and includes the "built-in" methods (described in [NRL 148]). Described here are the platform-dependent methods as provided in the reference implementation: constructors (this page) for the class, the methods for arithmetic operations (see page 28), and miscellaneous methods (see page 31) intended for general use.

The class `netrexx.lang.Rexx` is serializable.

Rexx constructors

These constructors all create a string of type `netrexx.lang.Rexx`.

Rexx(arg=boolean)

Constructs a string which will have the value `'1'` if *arg* is 1 (*true*) or the value `'0'` if *arg* is 0 (*false*).

Rexx(arg=byte)

Constructs a string which is the decimal representation of the 8-bit signed binary integer *arg*. The string will contain only decimal digits, prefixed with a leading minus sign (hyphen) if *arg* is negative. A leading zero will be present only if *arg* is zero.

Rexx(arg=char)

Constructs a string of length 1 whose first and only character is a copy of *arg*.

Rexx(arg=char[])

Constructs a string by copying the characters of the character array *arg* in sequence. The length of the string is the number of elements in the character array (that is, `arg.length`).

Rexx(arg=int)

Constructs a string which is the decimal representation of the 32-bit signed binary integer *arg*. The string will contain only decimal digits, prefixed with a leading minus sign (hyphen) if *arg* is negative. A leading zero will be present only if *arg* is zero.

Rexx(arg=double)

Constructs a string which is the decimal representation of the 64-bit signed binary floating point number *arg*.

(The precise format of the result may change for Java 1.1 and will be defined later.)

Rexx(arg=float)

Constructs a string which is the decimal representation of the 32-bit signed binary floating point number *arg*.

(The precise format of the result may change for Java 1.1 and will be defined later.)

Rexx(arg=long)

Constructs a string which is the decimal representation of the 64-bit signed binary integer *arg*. The string will contain only decimal digits, prefixed with a leading minus sign (hyphen) if *arg* is negative. A leading zero will be present only if *arg* is zero.

Rexx(arg=Rexx)

Constructs a string which is copy of *arg*, which is of type `netrexx.lang.Rexx`. *arg* must not be `null`. Any sub-values [NRL 70] are ignored (that is, they are not present in the object returned by the constructor).

Rexx(arg=short)

Constructs a string which is the decimal representation of the 16-bit signed binary integer *arg*. The string will contain only decimal digits, prefixed with a leading minus sign (hyphen) if *arg* is negative. A leading zero will be present only if *arg* is zero.

Rexx(arg=String)

Constructs a NetRexx string by copying the characters of *arg*, which is of type `java.lang.String`, in sequence. The length of the string is same as the length of *arg* (that is, `arg.length()`). *arg* must not be `null`.

Rexx(arg=String[])

Constructs a NetRexx string by concatenating the elements of the `java.lang.String` array *arg* together in sequence with a blank between each pair of elements. This may be used for converting the argument word array passed to the `main` method of a Java application into a single string.

If the number of elements of *arg* is zero then an empty string (of length 0) is returned. Otherwise, the length of the string is the sum of the lengths of the elements of *arg*, plus the number of elements of *arg*, less one.

arg must not be `null`.

Rexx arithmetic methods

These methods implement the NetRexx arithmetic operators, as described in [NRL 130]. Each corresponds to and implements a method in the RexxOperators interface class (see page 34).

Each of the methods here takes a RexxSet (see page 35) object as an argument. This argument provides the **numeric** settings for the operation; if `null` is provided for the argument then the default settings are used (**digits**=`9`, **form**=**scientific**).

For monadic operators, only the `RexxSet` argument is present; the operation acts upon the current object. For dyadic operators, the `RexxSet` argument and a `Rexx` argument are present; the operation acts with the current object being the left-hand operand and the second argument being the right-hand operand. For example, under default numeric settings, the expression:

```
award+extra
```

(where *award* and *extra* are references to objects of type `Rexx`) could be written as:

```
award.OpAdd(null, extra)
```

which would return the result of adding *award* and *extra* under the default numeric settings.

OpAdd(set=RexxSet, rhs=Rexx)

Implements the NetRexx + (Add) operator [NRL 57], and returns the result as a string of type `Rexx`.

OpAnd(set=RexxSet, rhs=Rexx)

Implements the NetRexx `&` (And) operator [NRL 59], and returns a result (0 or 1) of type `boolean`.

OpCc(set=RexxSet, rhs=Rexx)

Implements the NetRexx || or *abuttal* (Concatenate without blank) operator [NRL 57], and returns the result as a string of type `Rexx`.

OpCcblank(set=RexxSet, rhs=Rexx)

Implements the NetRexx *blank* (Concatenate with blank) operator [NRL 57], and returns the result as a string of type `Rexx`.

OpDiv(set=RexxSet, rhs=Rexx)

Implements the NetRexx / (Divide) operator [NRL 57], and returns the result as a string of type `Rexx`.

OpDivI(set=RexxSet, rhs=Rexx)

Implements the NetRexx % (Integer divide) operator [NRL 57], and returns the result as a string of type `Rexx`.

OpEq(set=RexxSet, rhs=Rexx)

Implements the NetRexx = (Equal) operator [NRL 58], and returns a result (0 or 1) of type `boolean`.

OpEqS(set=RexxSet, rhs=Rexx)

Implements the NetRexx == (Strictly equal) operator [NRL 59], and returns a result (0 or 1) of type `boolean`.

OpGt(set=RexxSet, rhs=Rexx)

Implements the NetRexx > (Greater than) operator [NRL 58], and returns a result (0 or 1) of type `boolean`.

OpGtEq(set=RexxSet, rhs=Rexx)

Implements the NetRexx >= (Greater than or equal) operator [NRL 59], and returns a result (0 or 1) of type `boolean`.

OpGtEqS(set=RexxSet, rhs=Rexx)

Implements the NetRexx >>= (Strictly greater than or equal) operator [NRL 59], and returns a result (0 or 1) of type `boolean`.

OpGtS(set=RexxSet, rhs=Rexx)

Implements the NetRexx >> (Strictly greater than) operator [NRL 59], and returns a result (0 or 1) of type `boolean`.

OpLt(set=RexxSet, rhs=Rexx)

Implements the NetRexx < (Less than) operator [NRL 59], and returns a result (0 or 1) of type `boolean`.

OpLtEq(set=RexxSet, rhs=Rexx)

Implements the NetRexx <= (Less than or equal) operator [NRL 59], and returns a result (0 or 1) of type `boolean`.

OpLtEqS(set=RexxSet, rhs=Rexx)

Implements the NetRexx <<= (Strictly less than or equal) operator [NRL 59], and returns a result (0 or 1) of type `boolean`.

OpLtS(set=RexxSet, rhs=Rexx)

Implements the NetRexx << (Strictly less than) operator [NRL 59], and returns a result (0 or 1) of type `boolean`.

OpMinus(set=RexxSet)

Implements the NetRexx `Prefix -` (Minus) operator [NRL 57], and returns the result as a string of type `Rexx`.

OpMult(set=RexxSet, rhs=Rexx)

Implements the NetRexx `*` (Multiply) operator [NRL 57], and returns the result as a string of type `Rexx`.

OpNot(set=RexxSet)

Implements the NetRexx `Prefix \` (Not) operator [NRL 60], and returns a result (0 or 1) of type `boolean`.

OpNotEq(set=RexxSet, rhs=Rexx)

Implements the NetRexx \= (Not equal) operator [NRL 58], and returns a result (0 or 1) of type `boolean`.

OpNotEqS(set=RexxSet, rhs=Rexx)

Implements the NetRexx \== (Strictly not equal) operator [NRL 59], and returns a result (0 or 1) of type `boolean`.

OpOr(set=RexxSet, rhs=Rexx)

Implements the NetRexx | (Inclusive or) operator [NRL 59], and returns a result (0 or 1) of type `boolean`.

OpPlus(set=RexxSet)

Implements the NetRexx `Prefix +` (Plus) operator [NRL 57], and returns the result as a string of type `Rexx`.

OpPow(set=RexxSet, rhs=Rexx)

Implements the NetRexx `**` (Power) operator [NRL 57], and returns the result as a string of type `Rexx`.

OpRem(set=RexxSet, rhs=Rexx)

Implements the NetRexx `//` (Remainder) operator [NRL 57], and returns the result as a string of type `Rexx`.

OpSub(set=RexxSet, rhs=Rexx)

Implements the NetRexx `-` (Subtract) operator [NRL 57], and returns the result as a string of type `Rexx`.

OpXor(set=RexxSet, rhs=Rexx)

Implements the NetRexx `&&` (Exclusive or) operator [NRL 60], and returns a result (0 or 1) of type `boolean`.

Rexx miscellaneous methods

These methods provide standard Java methods for the class, together with various conversions.

charAt(offset=int)

Returns the character from the string at *offset* (that is, if *offset* is 0 then the first character is returned, *etc.*). The character is returned as type `char`.

If *offset* is negative, or is greater than or equal to the length of the string, an exception is signalled.

equals(item=Object)

Compares the string with the value of *item*, using a strict character-by-character comparison, and returns a result of type `boolean`.

If *item* is `null` or is not an instance of one of the types `Rexx`, `java.lang.String`, or `char[]`, then 0 is returned. Otherwise, *item* is converted to type `Rexx` and the OpEqS (see page 29) method (or equivalent) is used to compare the current string with the converted string, and its result is returned.

hashCode()

Returns a hashcode of type `int` for the string. This hashcode is suitable for use by the `java.util.Hashtable` class.

toboolean()

Converts the string to type `boolean`. If the string is neither "`0`" nor "`1`" then a `NotLogicException` (see page 25) is signalled.

tobyte()

Converts the string to type `byte`. If the string is not a number, has a non-zero decimal part, or is out of the possible range for a `byte` (8-bit signed integer) result then a `NumberFormatException` is signalled.

tochar()

Converts the string to type `char`. If the string is not exactly one character in length then a `NotCharacterException` (see page 25) is signalled.

toCharArray()

Converts the string to type `char[]`. A character array object of the same length as the string is created, and the characters of the string are copied to the array in sequence. The character array is then returned.

todouble()

Converts the string to type `double`. If the string is not a number, or is out of the possible range for a `double` (64-bit signed floating point) result then a `NumberFormatException` is signalled.

tofloat()

Converts the string to type `float`. If the string is not a number, or is out of the possible range for a `float` (32-bit signed floating point) result then a `NumberFormatException` is signalled.

toint()

Converts the string to type `int`. If the string is not a number, has a non-zero decimal part, or is out of the possible range for an `int` (32-bit signed integer) result then a `NumberFormatException` is signalled.

tolong()

Converts the string to type `long`. If the string is not a number, has a non-zero decimal part, or is out of the possible range for a `long` (64-bit signed integer) result then a `NumberFormatException` is signalled.

toshort()

Converts the string to type `short`. If the string is not a number, has a non-zero decimal part, or is out of the possible range for a `short` (16-bit signed) result then a `NumberFormatException` is signalled.

toString()

Converts the string to type `java.lang.String`. A String object of the same length as the string is created, and the characters of the string are copied to the new string in sequence. The String is then returned.

SECTION 3: THE REXXOPERATORS INTERFACE CLASS

The `RexxOperators` interface class defines the signatures of the methods that implement the NetRexx (and Rexx) operators. These methods are described in the section *Rexx arithmetic methods (see page 28)* .

In the future this interface may be used to allow the overloading of operators for objects of types other than `Rexx`. The current NetRexx language definition does not permit operator overloading.

SECTION 4: THE REXXSET CLASS

The `RexxSet` class is used to provide the numeric settings for the methods described in the section *Rexx arithmetic methods (see page 28)* . When provided, a RexxSet Object supplies the **numeric** settings for the operation; when `null` is provided then the default settings are used (**digits**=`9`, **form**=**SCIENTIFIC**).

Public properties

These properties supply the numeric settings and certain values they may take. After construction, the **digits** and **form** values should only be changed by using the **setDigits** and **setForm** methods.

DEFAULT_DIGITS

A constant of type `int` that describes the default number of digits for a numeric operation (9).

DEFAULT_FORM

A constant of type `byte` that describes the default exponential format for a numeric operation (**SCIENTIFIC**).

digits

A value of type `int` that describes the numeric digits to be used for a numeric operation. The Rexx arithmetic methods (see page 28) use this value to determine the significance of results. **digits** must always be greater than zero.

ENGINEERING

A constant of type `byte` that signifies that engineering exponential formatting should be used for a numeric operation.

form

A value of type `byte` that describes the exponential format to be used for a numeric operation. The Rexx arithmetic methods (see page 28) use this value to determine the formatting of results that require an exponent. **form** must be either **ENGINEERING** or **SCIENTIFIC**.

SCIENTIFIC

A constant of type `byte` that signifies that scientific exponential formatting should be used for a numeric operation.

Constructors

These constructors are used to set the initial values of a RexxSet object.

RexxSet()

Constructs a RexxSet object which has default **digits** and **form** properties.

RexxSet(newdigits=int)

Constructs a RexxSet object which has its **digits** property set to *newdigits* and its **form** property set to **DEFAULT_DIGITS**.

RexxSet(newdigits=int, newform=byte)

Constructs a RexxSet object which has its **digits** property set to *newdigits* and its **form** property set to *newform*.

RexxSet(arg=RexxSet)

Constructs a RexxSet object which is copy of *arg*, which is of type `netrexx.lang.RexxSet`. *arg* must not be `null`.

Methods

The RexxSet class has the following additional methods:

formword()

Returns a string of type `netrexx.lang.Rexx` that describes the **form** property. This will either be the string `'engineering'` or the string `'scientific'`, corresponding to the **form** value **ENGINEERING** or **SCIENTIFIC**, respectively.

setDigits(newdigits=Rexx)

Sets the **digits** value for the `RexxSet` object, from *newdigits*, after rounding and checking as defined for the **numeric** instruction; *newdigits* must be a positive whole number with no more than nine digits. No value is returned.

setForm(newformword=Rexx)

Sets the **form** value for the `RexxSet` object, from *newformword*. This must equal either the string `'engineering'` or the string `'scientific'`, corresponding to the **form** value **ENGINEERING** or **SCIENTIFIC**, respectively. No value is returned.

Index

J

L

M

N

O

P

R

S

T

U

V

W

