LOCAL ENVIRONMENT AND SCOPES IN OBJECT REXX

Rony G. Flatscher

Department of Management and Information Systems

Vienna University of Economics and Business Administration

„7th International Rexx Symposium“, Austin/Texas, May 13th-15th, 1996

ABSTRACT

Object Rexx extends Rexx with additional concepts of scopes. Firstmost the developers refer to the scope in which an individual object is executing in and describe „scope“ in the online manual (version 16th February, 96) as:

„A scope is the methods and object variables defined in a single class. Only methods defined in a particular scope can access object variables within that scope. This means that object variables in a subclass can have the same names as object variables in a superclass, because the object variables are at different scopes.“

This scope is relevant in discussing the possibilities of concurrent access to object variables, if multiple methods are supposed to execute in parallel for the same object.

But looking closer at Object Rexx it turns out that there are many more (non-object related) scopes available which are worthwhile to explore: scopes of procedures, routines and methods.

In addition to the scopes related to a single program the concept of the „local environment“ adds another dimension to the discussion of scopes: the local environment itself and the visibility - and therefore accessibility of classes and routines.

1 SCOPES OF TRADITIONAL REXX PROGRAMS

In traditional Rexx (abbreviated T-Rexx) the scope (also named *„standard scope“*) of variables and labels is global, i.e. variables and labels are visible and accessible thruout the entire Rexx program.

If the definition of a label is immediately followed by a „`PROCEDURE`“-statement and this label gets `CALL`ed, then T-Rexx defines a separate scope for variables at the entry of that label (*„procedure scope“*). This in effect hides all variables of the caller from the code constituting a procedure or a function, yet, labels remain still visible. Therefore it is possible to change the flow of control by `CALL`ing or `SIGNAL`ling any label in the T-Rexx-program.

```
/* */
a = 1
b = 2
SAY a      /* displays "1" */
SAY b      /* displays "2" */

CALL proc1
SAY a      /* displays "4" */
SAY b      /* displays "5" */

CALL proc2
SAY a      /* displays "4" */
SAY b      /* displays "2" */
EXIT

PROC1 :
  a = 4
  b = 5
  RETURN

    /* local scope with access to global "b" */
PROC2 : PROCEDURE EXPOSE b
  a = 1
  b = 2
  RETURN
```

Figure 1: T-Rexx global and local scopes

If the programmer so desires, it is possible to allow access to some variables of the caller from within the called procedure/function by using the `EXPOSE` keyword and denominating those variables which should get incorporated into the procedure's scope and thereby becoming accessible again.

Figure 1 depicts the scopes of T-Rexx programs, procedure `PROC1` changes the values of the globally visible variables „`a`" and „`b`". By contrast `PROC2` creates a procedure scope allowing access to the global variable „`b`" (via the `EXPOSE` keyword): changes to variable „`a`" remain local, changes to variable „`b`" occur globally in this case.

2 OBJECT REXX

2.1 Local Environment

The session in which an Object Rexx program executes constitutes a *„local environment"* in the sense that any Rexx program loaded by another Rexx program (via a `CALL`-statement or a `::REQUIRES` directive) is mereley added to the session's memory. All the Rexx programs build a tree whereby the root is simply the very first Rexx program invoked. Keeping all Rexx programs which form an application in memory allows for passing objects effectively by reference.

Routines and classes with the attribute „`PUBLIC`" are brought into the scope of the calling Rexx program (*„program scope"*) right after a `CALL` to an external Rexx program returns or right after a `::REQUIRES`-directive has been resolved. Only those public routines and classes of „children" are added to the program scope of the caller, which are not already defined there. Therefore it is not possible that called/required Rexx programs override routines and classes of the caller itself.

In the case that several Rexx „children" contain public routines and classes by the same name those prevail, which are nearest to the caller with respect to the number of Rexx program „nodes" in between and in case of a tie, the program scope of the Rexx program is used which was called last. See figure 2 demonstrating the effects of `::REQUIRE`-directives and `CALL`-statements (output in figure 3).

```
/* main.cmd */

SAY
SAY "in MAIN.CMD (executing) ..."
CALL hi     /* call routine named HI (in PROC2.CMD) */
CALL proc3.cmd
CALL hi     /* call routine named HI (in PROC3.CMD) */
CALL proc4.cmd
CALL hi     /* call routine named HI (in PROC4.CMD) */
CALL hi     /* call routine named HI (in PROC4.CMD) */
SAY "leaving MAIN.CMD."

:: REQUIRES proc1.cmd  /* load PROC1.CMD into local environment */
```

```
/* proc1.cmd */

SAY
SAY "in PROC1.CMD (initializing) ..."
CALL hi         /* local routine prevails        */
SAY "... leaving PROC1.CMD."

:: REQUIRES proc2    /* load PROC2.CMD into local environment */

:: ROUTINE hi      /* non-public ! */
  SAY "... in routine 'HI' of PROC1.CMD (private!)"
```

```
/* proc2.cmd */

SAY
SAY "in PROC2.CMD (initializing) ..."
CALL hi         /* local routine prevails        */
SAY "... leaving PROC2.CMD."

:: ROUTINE hi  PUBLIC
  SAY "... in routine 'HI' of PROC2.CMD, which is public"
```

```
/* proc3.cmd */

SAY "in PROC3.CMD (initializing) ..."
CALL hi         /* local routine prevails        */
SAY "... leaving PROC3.CMD."

:: ROUTINE hi  PUBLIC
  SAY "... in routine 'HI' of PROC3.CMD, which is public"
```

```
/* proc4.cmd */

SAY "in PROC4.CMD (initializing) ..."
CALL hi         /* local routine prevails        */
SAY "... leaving PROC4.CMD."

:: ROUTINE hi  PUBLIC
  SAY "... in routine 'HI' of PROC4.CMD, which is public"
```

Figure 2: Requiring and calling Object Rexx programs (code)

```
in PROC2.CMD (initializing) ...
... in routine 'HI' of PROC2.CMD, which is public
... leaving PROC2.CMD.

in PROC1.CMD (initializing) ...
... in routine 'HI' of PROC1.CMD (private!)
... leaving PROC1.CMD.

in MAIN.CMD (executing) ...
... in routine 'HI' of PROC2.CMD, which is public
in PROC3.CMD (initializing) ...
... in routine 'HI' of PROC3.CMD, which is public
... leaving PROC3.CMD.
... in routine 'HI' of PROC3.CMD, which is public
in PROC4.CMD (initializing) ...
... in routine 'HI' of PROC4.CMD, which is public
... leaving PROC4.CMD.
... in routine 'HI' of PROC4.CMD, which is public
... in routine 'HI' of PROC4.CMD, which is public
leaving MAIN.CMD.
```

Figure 3: Requiring and calling Object Rexx programs (output)

2.3 Environment Symbols

Environment symbols allow programmers to access directory entries in Object Rexx supplied directories like `.LOCAL` or `.ENVIRONMENT`, without the need to know in which particular directory to look. Some maintenance directories are not even named and hence not directly accessible to programmers. E.g. all class objects defined in Rexx programs are directly accessible via environment symbols only.

An example for utilizing the session directory denominated by „`.LOCAL`“:

```
.local ~ rgf.test = "This is some string."
```

The easiest way to retrieve the directory entry for „`RGF.TEST`“ is by usage of another environment symbol: by simply prepending a dot to the entry name, i.e. „`.RGF.TEST`“. Alternatively „`.local ~ entry('RGF.TEST')`“ would work also.

Please note, environment symbols are *not* variables and they *must* start with a dot! They are a valid means of accessing entries of Object Rexx administered directories.

This brings us to the rules which Object Rexx follows in order to resolve environment symbols:

1) Object Rexx searches an unnamed program specific directory - let us call it „source" - for an entry, if this fails it

2) searches the directory „source" of the Rexx programs incorporated via „`::REQUIRES`" directives or `CALL`-statements recursively, if this fails it

3) searches the session wide available and named directory „`.LOCAL`", if this fails it

4) searches the operating system wide available and named directory „`.ENVIRONMENT`", if this fails it

5) searches another unnamed directory - let us call it „bottom" -, now if this fails

6) Object Rexx will return the environment symbol as a string constant, with all letters in uppercase.

Rules 1) and 2) are needed for making it possible e.g. to address class objects defined in Rexx programs, like

```
anObject = .MyClass ~ new
```

where „`.MYCLASS`" is an environment symbol relating to a class object which may have been defined via a directive like „`::CLASS MyClass`". Rule 2 follows the program scopes in that Object Rexx tries to find such public class objects in leaf Rexx programs.

The Directory „`.LOCAL`"

For every separate session (operating system process) the Rexx interpreter creates a directory object accessible via the environment symbol named „`.LOCAL`". This directory allows for storing and retrieving objects for *all* Rexx programs loaded into the local environment of that particular session.

As `.LOCAL` is visible to all executing Rexx-programs, it becomes possible to pass objects (data) between Rexx programs and even between *different* directives (i.e. `::ROUTINE`- and `::METHOD`-directives).

The Directory „`.ENVIRONMENT`"

In the system there exists one directory object which gets created by Object Rexx on startup of the system and is accessible via the environment symbol named „`.ENVIRONMENT`". Most of the documented Object Rexx classes are accessible via this directory.

Like `.LOCAL` it allows for exchanging objects among Object Rexx programs, in this case even across session (process) borders!

Once objects are stored in the `.ENVIRONMENT` directory all session related information is kept, so the object is guaranteed to have its valid working environment present (e.g. routines and other classes of the program scope where the object stems from). Therefore objects placed into the `.ENVIRONMENT`-directory should be removed once they are not needed anymore, so the garbage collecting system can reclaim the space.

2.6 Scopes of Object Rexx Programs and Directives

The T-Rexx scopes remain valid for Object Rexx programs as long as there are no directives defined with them. Directives, which are led in by two consecutive colons („`::`"), allow for defining:

- the merging of additional Rexx programs („`::Requires`"),
- routines („`::Routine`"),
- methods („`::Method`"), and
- classes („`::Class`").

Every single directive gets its own, individual scope, as if every single directive was an Object Rexx program of its own. This has the effect, that previously defined variables or labels (!) of the Rexx program in question are not available anymore.

On the other hand, because directives have their own scope, the basic T-Rexx scope rules apply for them. With other words, it is possible and very feasible to define „global" variables and labels *within* the scope of a directive, only visible from within it.

Coding of Rexx-statements is possible within the directives `::ROUTINE` and `:METHOD` only.

`::REQUIRES` Directive

When the Object Rexx interpreter first parses a Rexx program, it will check the code for syntactical errors and collect all directives. Then it will:

1) `CALL` (!) the Rexx program(s) denoted by the `::REQUIRES` directive(s) in the order they appear,

2) initialize the defined classes in the *called* Rexx program,

3) passing control to the very first statement in the Rexx program.

This resolution will work recursively on all required Rexx programs. The result are program scopes for all Rexx programs involved.

The same steps are taken at the moment an external Rexx program is called explicitly with a `CALL`-statement.

`::ROUTINE` Directive

A `::Routine`-directive defines a procedure or a function, which gets added to the program scope and may be therefore called from *any* part of an Object Rexx program, even from within directives. If the attribute „`PUBLIC`" is given, then parent Rexx programs may get these routines added to their program scopes too.

The routine constitutes its own scope („*routine scope*") working to the T-Rexx rules, i.e. it may have „global" variables and labels which may serve as targets for `CALL`- or `SIGNAL`-statements. Like in a normal program it is possible to use the keywords `PROCEDURE` and `EXPOSE` with labels within a routine definition.

Figure 2 demonstrates the effect of `::REQUIRES`-directives intermixed with `CALL`-statements. `MAIN.CMD` requires `PROC1.CMD`, which defines a local routine named '`Hi`' and requires `PROC2.CMD` in turn, which defines a public routine by the same name '`Hi`'. `PROC3.CMD` and `PROC4.CMD` both contain a public routine '`Hi`' and will get called by `MAIN.CMD` once the initialisation phase has terminated.

Figure 3 shows the generated output from which it becomes clear how the program scope for `MAIN.CMD` with respect to the routines named '`Hi`' is setup in the different stages of execution. The Rexx code before the first directive in all programs starting with „`PROC`" serves in effect as initialisation code for the appropriate programs. This effect can be seen if looking for the routine '`Hi`' of `PROC4`: the very first time the code before the first directive is run; the second time that routine is called only the '`Hi`' routine in `PROC4` will be executed.

As long as `PROC3` and `PROC4` are not called the routine '`Hi`' of `PROC2` will be used. `PROC1`'s '`Hi`' routine is private to `PROC1` hence not visible to `MAIN`. `PROC3` then replaces the '`Hi`' routine of `PROC2` in `MAIN`'s program scope and in turn gets replaced by `PROC4`'s routine '`Hi`'.

`::METHOD` Directive

The `::METHOD` directive defines a method consisting of Rexx statements, which implement a particular „behavior" of a class. The method constitutes its own scope („*method scope*") working to the T-Rexx rules, i.e. it may have „global" variables and labels which may serve as targets for `CALL`- or `SIGNAL`-statements. Like in a normal program it is possible to use the keywords `PROCEDURE` and `EXPOSE` for *labels* within a method definition.

In addition it is possible for a method to gain access to the object variables of the class it belongs to by using the `EXPOSE` keyword *immediately* after the `::METHOD` directive, denoting the names of the object variables the method needs access to.

As methods merely define the blueprint for the behavior of all objects of a particular class, the scope determining which methods may access which object variables at object-runtime is called „*object scope*“. It is this scope which the online help of Object Rexx is referring to.

It is interesting to note that methods may be defined *outside* of a class' context, i.e. before a class directive is encountered. In this case the Object Rexx interpreter gathers all „floating“ methods into a directory and makes it available to the program via the environment symbol „`.METHODS`“ (which gets stored in the unnamed „source“ directory, see above). Therefore *every* Rexx program defining floating methods has its own, specific directory of floating methods which may be retrieved via the environment symbol „`.METHODS`“. If floating methods are used e.g. to enhance a class' object instance with additional instance methods, they define their own scope, with the pseudo variable „`super`“ pointing to the original class methods and object variables.

Figure 4 shows a little program which has floating methods defined. The enhanced object receives these floating methods as instance methods, which in turn form a unique object scope, totally isolated from the object scope of the instance methods at the level of the `.Test1`-class. This is to say, if a particular object is executing within a floating method it has access to a different object scope (different object variables) than when executing within the methods defined for the `.Test1`-class, which has its own unique set of object variables.

```
/* meth.cmd */

anObject = .test1 ~ new
anObject ~ one
anObject ~ two
anObject ~ showVar1
SAY COPIES( "-", 50 )

anObject = .test1 ~ ENHANCED( .methods )
anObject ~ one
anObject ~ two
anObject ~ showVar1

:: ROUTINE pp
  RETURN "[" || ARG( 1 ) || "]"

/* ----------------- floating methods ------------------- */
:: METHOD one
  EXPOSE var1

  IF \ VAR( "VAR1" ) THEN var1 = "---"
  SAY "floating.ONE -    var1:" pp( var1 )
  var1 = "floating.1"
  SAY "floating.ONE - var1-method:" pp( self ~ var1 )
  self ~ one:super

:: METHOD two
  EXPOSE var1

  SAY "floating.TWO -    var1:" pp( var1 )
  var1 = "floating.2"
  SAY "floating.TWO - var1-method:" pp( self ~ var1 )
  self ~ two:super

:: METHOD var1 ATTRIBUTE


/* ----------------- class with methods ----------------- */
:: CLASS test1

:: METHOD one
  EXPOSE var1

  IF \ VAR( "VAR1" ) THEN var1 = "???"
  SAY "Test1.ONE -      var1:" pp( var1 )
  var1 = "Test1.1"
  SAY "Test1.ONE -    var1-new:" pp( var1 )
  SAY

:: METHOD two
  EXPOSE var1

  SAY "Test1.TWO -      var1:" pp( var1 )
  var1 = "Test1.2"
  SAY "Test1.TWO -    var1-new:" pp( var1 )
  SAY
```

```
:: METHOD ShowVar1
  EXPOSE var1

  SAY "Test1.Show -     var1:" pp( var1 )

:: METHOD var1 ATTRIBUTE
```

Figure 4: Floating methods, a class with methods and enhancing an object instance with floating methods

```
Test1.ONE -      var1: [???]
Test1.ONE -    var1-new: [Test1.1]

Test1.TWO -      var1: [Test1.1]
Test1.TWO -    var1-new: [Test1.2]

Test1.Show -     var1: [Test1.2]
--------------------------------------------------
floating.ONE -    var1: [---]
floating.ONE - var1-method: [floating.1]
Test1.ONE -      var1: [???]
Test1.ONE -    var1-new: [Test1.1]

floating.TWO -    var1: [floating.1]
floating.TWO - var1-method: [floating.2]
Test1.TWO -      var1: [Test1.1]
Test1.TWO -    var1-new: [Test1.2]

Test1.Show -     var1: [Test1.2]
```

Figure 5: The object scope for floating methods is different to the object scope of the class .Test1 (output)

There is an attribute method defined for the floating methods and one for the class `.Test1`. Both of these object variables exist at *different* object scopes, hence they may store different values. The output is given in figure 5.

`::CLASS` Directive

A `::CLASS`-directive merely defines the properties of a class and serves as the „glue point“ for all immediately following `::Method`-directives, which implement the behavior of the methods themselves. All the methods at the instance level with their *instance* object variables define a separate object scope, as well as all the methods at the class level with their *class* object variables.

Looking at the class hierarchy, as a matter of fact, each single class forms its own object scope, defining the methods and object variables level by level. So if an instance of a class gets asynchronous messages - e.g. by using the `START`-method of `.Object` - it becomes possible that different messages at different object scopes (e.g. different class levels) execute concurrently without problems on the same object.

If a *second* method should get invoked at an object scope in which another method is already running on the *same* object, the second method will have to wait by default until the first method finishes. This way Object Rexx by default prevents methods from changing object variables of the same object concurrently, which usually is undesired and sometimes dangerous. If the programmer wishes, that some methods execute concurrently at the same object scope for the same object, he or she can do so by either using the `UNGUARDED` attribute for the appropriate method directive or use the keyword instruction `GUARD ON`/`OFF`.

Another side effect of object scopes is the ability for programmers to define names for object variables which may already have been used someplace up in the hierarchy tree without interfering with them.

```
/* test_scope.cmd */

anObject = .Test2 ~ new( "one", "two" )
msg = anObject ~ start( "work1" )     /* object scope of .Test2 */
msg = anObject ~ start( "work0" )     /* object scope of .Test1 */
CALL SysSleep 1                       /* sleep a bit            */
anObject ~ work2                      /* object scope of .Test2 */
SAY "program ended."
EXIT

/* ----------------- class with methods ----------------- */
:: CLASS Test1

:: METHOD work0
  SAY "in Test1::work0 (object scope of class .Test1)"
  var1 = "in work2"
  SAY "in Test1::work0, leaving."

:: CLASS Test2 SUBCLASS Test1

:: METHOD init
  EXPOSE var1 var2   /* define instance object variables   */
  USE ARG var1, var2 /* assign arguments as values      */

:: METHOD work1     /* UNGUARDED  */
  EXPOSE var1

  var1 = "in work1"
  SAY "in Test2::work1 (object scope of class .Test2)"
  DO 5
   SAY "in Test2::work1, sleeping a second ..."
   CALL SysSleep 1  /* wait 1 second            */
  END
  SAY "in Test2::work1, slept 5 times, just woke up !"

:: METHOD work2

  SAY "in Test2::work2 (object scope of class .Test2)"
  SAY "in Test2::work2, leaving."
```

Figure 6: Object Scope and Concurrency

Figure 7 demonstrates the output of the Object Rexx program in figure 6 executing concurrently methods of *different* object scopes for the *same* object. While method „`work1`“, defined at the object scope of class `.Test2`, is asynchroneously executing for object „`anObject`“, method „`work0`“ defined in the superclass `.Test1` is able to run concurrently. Method „`work2`“ defined at the object scope of class `.Test2` is blocked until method „`work1`“ executing in the same object scope finishes.

```
in Test2::work1 (object scope of class .Test2)
in Test2::work1, sleeping a second ...
in Test1::work0 (object scope of class .Test1)
in Test1::work0, leaving.
in Test2::work1, sleeping a second ...
in Test2::work1, sleeping a second ...
in Test2::work1, sleeping a second ...
in Test2::work1, sleeping a second ...
in Test2::work1, slept 5 times, just woke up !
in Test2::work2 (object scope of class .Test2)
in Test2::work2, leaving.
program ended.
```

Figure 7: Object Scope and Concurrency (output)

3 SUMMARY

This paper discussed various ascpects of scopes as present in Object Rexx. As many of these scopes do not have names so far, the author attempted to coin some:

- *Standard scope*: scope rules as established in T-Rexx which determine the visibility of variables and labels in a plain Rexx program, i.e. in a program not containing any directives.

- *Procedure scope*: determines the visibility of variables within called labels, which use the `PROCEDURE` keyword.

- *Program scope*: determines which routines and classes are visible to a particular Rexx program.

- *Routine scope*: determines the visibility of variables and labels within a routine directive. It follows the T-Rexx scope rules.

- *Method scope*: determines the visibility of variables and labels within a method directive. It follows the T-Rexx scope rules. In addition methods gain access to object variables according to the object scope the method is running in.

- *Object scope*: determines which methods may directly access the same object variables, where object variables may be defined for instance, class and floating methods. Every class constitutes its own object scope. All methods of the same object scope are by default prevented from executing concurrently for the *same* object, unless the programmer specifies otherwise.

The environment symbols allow for retrieving objects stored in different Object Rexx supplied directories, the two most important being `.LOCAL` and `.ENVIRONMENT`. `.LOCAL` is meant for storing or exchanging objects (data) among Object Rexx programs executing within the same session and thus defining the „local environment". `.ENVIRONMENT` is meant for retrieving class objects and exchanging objects (data) across session (process) boundaries, i.e. on a global basis.

4 ACKNOWLEDGEMENTS

The author wishes to thank Rick McGuire, one of the former lead developers for Object Rexx, for his great hints and explanations given on the Internet-newsgroup „`comp.lang.rexx`" thru the past years.

5 REFERENCES

Online documentations of various beta versions of Object Rexx (the version used for this paper stems from February 16th, 1996).

Various postings on the internet newsgroup „`comp.lang.rexx`", 1995-1996.

Cowlishaw, M.F.: „The REXX Language", Prentice-Hall (Second edition), 1990..

Date of Article: 1996-05-30.

Published in: Proceedings of the "7th International REXX Symposium", Texas/Austin, May 12th-15th, 1996, The Rexx Language Association, Raleigh N.C. 1996.

Presented at: "7th International REXX Symposium", Texas/Austin, May 12th-15th, 1996, The Rexx Language Association.