Getting Ready for Object Rexx

Rick McGuire IBM Endicott

Pages 194-218

Getting Ready for Object REXX

Rick McGuire
Object REXX Development
IBM Endicott





Answers to Questions without Answers

- A major goal of Object REXX is removing limitations of the existing REXX language.
- Many of the limitations are seen in some of the most frequently asked (and frequently unanswered) questions on bulletin boards.







Let's Practice

- Question: How do I convert dates from on e REXX format to another?
- Current Answer: Well, you don't....
- Object REXX Answer: Just specify the input date as the second argument to the Date() function. A third option argument tells Date() what input format you are using:
 - Date('b', '28 Feb 1995')
 - Date('n', '02/28/1995', 'U')





Passing Stems

- Question: How do I pass a stem to a function or subroutine
- Answer: Just specify the stem in the argument list and access the argument with the USE ARG instruction.

call StemSort stem., count

StemSort: procedure use arg x., count

return



19



Returning Multiple Values

- Question: How do I return more than just a single string value from a function?
- Answer: Just return a stem or other "composite" object

```
lines. = ReadFile(filename)
```

ReadFile: procedure
parse arg filename
count = 0
do while lines(filename) <> 0
count = count + 1
x.count = linein(filename)
end
x.0 = count
return x.





Expressions in Compound Tails

- Question: How do I specify that A.i = A.i+1?
- Answer: Specify the variable part of the tail within square brackets ("[]")

lines. = ReadFile(filename)

ReadFile: procedure
parse arg filename
x.0 = 0
do while lines(filename) <> 0
x.0 = x.0 + 1
x.[x.0] = linein(filename)
end
return x.





Traversing Stems

• Question: How do I traverse all of the tails currently assigned to a stem?

Answer: Use the DO OVER instruction Do tail over stem. say stem.tail end









Packaging Multiple Functions

• Question: Now do I distribute a "bunch" of external functions without creating a file for each function?

 Answer: Package the routines in a "Requires" file ::requires sitefunc.cmd

::routine function1 public

::routine function2 public

::routine function3 public





Bonus Function

 Requires files can also perform needed global setup /* load required functions */
call rxfuncadd 'a', 'b', 'c'

::routine function1 public

::routine function2 public







Sharing Variables Between Programs

- Question: How can I share "global variables" between multiple programs?
- Answer: Access the variables as a REXX "environment" variable

.environment-setentry(,
 'MY.PROGRAM',,
 .directory new

.my.program~name = "xyz"





The "Procedure Expose" Dilemma

- Question: How can I share variables between related subroutines without doing a PROCEDURE EXPOSE for every variable through all of the caller's levels?
- Answer: Structure the related routines as an object and share the variables with the EXPOSE instruction

::class data_manager

::method x

expose name time type

::method y expose time type attributes

::method z expose attributes





Computed CALL instructions

- Question: How do I make a call to a routine whose name is contained in a variable?
- Answer: Use an indirect CALL instruction, placing the routine variable name in parentheses

parse arg name, argument call (name) argument



Replacing Common Idioms

 Some common REXX idioms can be made easier using features of Object REXX or by replacing stems with other REXX objects.





Stems vs. Arrays

- A REXX array may be the more appropriate choice
 - Variable size
 - Automatically tracks the size
 - DO OVER traverses in order

lines = ReadFile(filename)

ReadFile: procedure
parse arg filename
output = .queue-new
do while lines(filename) <> 0
output-add(linein(filename))
end
return output-makearray





Stems vs. Directories

 Compound variables can be "vulnerable" to other variable usage in a program employee.name

Can fail if name is used as a variable, but

employee = .directory-new employee-name = "Rick"

is always safe!



Stems vs. Directories

 Using compound variables as both "collections" and "structures" simultaneously can be awkward employees.i.name = "Rick"
employees.i.salary = "???"

VS.

employees[i] = nextWorker()

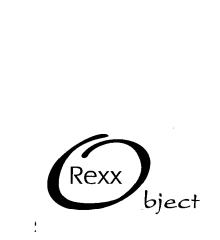






Consider Building You Own Objects

- While many problems can be adequately solved by stems, arrays, directory, etc., consider building your own objects:
 - Hide the processing logic
 - Can be placed in a REQUIRES file for better reuse.







A Common Problem

- Customer wants to process a group of records contained in a flat file, with the data fields organized in columns.
 - Records must be easily accessed, updated, and written out to a new file in the same format.
 - Record formats are subject to change, so updates must be easily performed.
 - Multiple programs will be written to perform updates against the same files.





A Solution

::class employee ::method init expose name id address salary manager parse arg name 25 id 32 address 100 salary , 106 manager 131

::method name attribute

::method id attribute

::method address attribute

::method salary attribute

::method manager attribute

::method string return left(name, 25) || left(id, 7) || left(address, 68) || , right(salary, 6) || left(manager, 25)





A Solution (continued)

```
/* Give everybody a raise! */
parse arg oldFile newFile

do while lines(oldFile) <> 0
employee = .employee-new(linein(oldFile))
employee-salary = employee-salary + ,
employee-salary * .10
call lineout newFile, employee
end
```

::requires employ /* include the employee records */





Building New Idioms

- Over the years, many common REXX idioms have been developed
- These idioms are still valid, but...
 - New Object REXX idioms may replace some existing ones
 - New Object REXX programming idioms will be added to existing ones





For Your Consideration...

- A new Object REXX programming idiom, the "caching directory"
 - Keep a cache of items read from a disk file
 - Caching is done on first reference to an item
 - Subsequent requests pull the item from the cache







The Caching Directory

::method unknown
expose dataFile
parse arg employeeld
if \var(dataFile) then dataFile = .stream new('emp.rec')
record = dataFile linein(Employeeld%100)
record = .employee new(record)
self[employeeld] = record
return record

::requires employ

