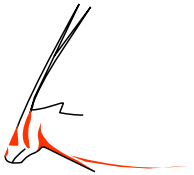
"From Rexx to ooRexx"



The 2021 International Rexx Symposium

Online ("Covid-19")

November 7th – November 11th 2021

© 2021 Rony G. Flatscher (Rony.Flatscher@wu.ac.at)

Wirtschaftsuniversität Wien, Austria (http://www.wu.ac.at)

Agenda

- Brief History
- Getting Object Rexx
- Some new features like
  - USE ARG
- New: Directives
  - ::ROUTINE, ::REQUIRES
  - ::CLASS, ::ATTRIBUTE, ::METHOD, ::CONSTANT
- Roundup

Brief History, 1

- Begin of the 90s
  - OO-version of Rexx (Object REXX) presented to the IBM user group "SHARE"
  - Developed since the beginning of the 90s
    - Originally conceived by a team led by Simon Nash
    - Rewritten product under the lead of Rick McGuire
  - 1997 Introduced with OS/2 Warp 4
    - *Support of SOM and WPS*
  - 1998 Free Linux version, trial version for AIX
  - 1998 Windows 95 and Windows/NT

Brief History, 2

- RexxLA and IBM negotiate
  - 2004 IBM handed over source code
  - "Open Object Rexx (ooRexx) 3.0"
    - Open source version of IBM's Object REXX
    - Released by RexxLA: 2005-03-25
  - ooRexx 4.0 (2009)
    - New kernel, 32- and 64-bit became possible
  - ooRexx 4.2 (2014)
  - ooRexx 5.0 currently in beta, *but better than 4.2!*

Some New Features

- Compatible with classic Rexx, TRL 2
  - New sequence of execution of Rexx programs:

    Phase 1: Full syntax check of the Rexx program upfront

    Phase 2: Interpreter carries out all directives (lead in with "::")

    Phase 3: Start of program execution with line # 1
- rexxc[.exe]: compiles Rexx programs
  - If same bitness and same endianness, on all platforms
- USE ARG in addition to PARSE ARG
  - among other things allows for retrieving stems *by reference (!)*
- Line comments, led in by two dashes ("--")

  -- *comment until the line ends*

Stem, Classic REXX Example

"stemclassic.rex"

```
s.1="Entry # 1"
s.2="Entry # 2"
s.0=2          /* total number of entries in stem           */

call add2stem  /* add to stem using an (internal) routine   */

do i=1 to s.0  /* iterate over all stem array entries       */
   say "#" i":" s.i
end
exit

add2stem: procedure expose s. -- allow access to stem
  n=s.0+1      /* add after last current entry              */
  s.n="Entry #" n "added in add2stem()"
  s.0=n        /* update total number of entries in stem    */
  return

/* yields:

   # 1: Entry # 1
   # 2: Entry # 2
   # 3: Entry # 3 added in add2stem()

*/
```

Stem, REXX with USE ARG Example "stemusearg.rex"

```
s.1="Entry # 1"
s.2="Entry # 2"
s.0=2              /* total number of entries in stem          */

call add2stem s.   /* supply stem as an argument!              */

do i=1 to s.0      /* iterate over all stem array entries      */
   say "#" i":" s.i
end
exit

add2stem: procedure  /* no "expose s." needed anymore !        */
  use arg s.     /* USE ARG allows to directly refer to the stem */
  n=s.0+1        /* add after last current entry               */
  s.n="Entry #" n "added in add2stem()"
  s.0=n          /* update total number of entries in stem     */
  return

/* yields:

   # 1: Entry # 1
   # 2: Entry # 2
   # 3: Entry # 3 added in add2stem()

*/
```

Stem, ooRexx USE ARG Example "stemroutine1.rex"

```
s.1="Entry # 1"
s.2="Entry # 2"
s.0=2               /* total number of entries in stem           */

call add2stem s.    /* supply stem as an argument!               */

do i=1 to s.0       /* iterate over all stem array entries       */
   say "#" i":" s.i
end

::routine add2stem
  use arg s.    /* USE ARG allows to directly refer to the stem */
  n=s.0+1       /* add after last current entry                 */
  s.n="Entry #" n "added in add2stem()"
  s.0=n         /* update total number of entries in stem       */
  return

/* yields:

   # 1: Entry # 1
   # 2: Entry # 2
   # 3: Entry # 3 added in add2stem()

*/
```

Stem, ooRexx USE ARG Example "stemroutine2.rex"

```
s.1="Entry # 1"
s.2="Entry # 2"
s.0=2              /* total number of entries in stem           */

call add2stem s.   /* supply stem as an argument!               */

do i=1 to s.0      /* iterate over all stem array entries       */
   say "#" i":" s.i
end

::routine add2stem   /* we can even use a different stem name  */
  use arg abc. /* USE ARG allows to directly refer to the stem */
  n=abc.0+1    /* add after last current entry                 */
  abc.n="Entry #" n "added in add2stem()"
  abc.0=n      /* update total number of entries in stem       */
  return

/* yields:

   # 1: Entry # 1
   # 2: Entry # 2
   # 3: Entry # 3 added in add2stem()

*/
```

About Directives in ooRexx

- Always placed at the end of a Rexx program
  - led in by "**::**" followed by the name of the directive
    - "routine", "class", "attribute", "method", ...
- Instructions to the ooRexx interpreter before program starts
  - Interpreter sequentially processes and carries out directives in *phase 2* of startup (*phase 1* is the syntax checking phase)
  - After all directives got carried out, *phase 3* starts, *the execution of the Rexx program* with line # 1
- An ooRexx program with directives
  - Defines a "package" of routines and classes
  - Rexx code before the first directive is named "prolog"

::Routine Directive

- Syntax

  ::routine name [public]

  – Interpreter maintains routines (and classes) per Rexx program ("package")

  – If optional keyword public is present, the routine can be also *directly invoked by another (!) Rexx program*

::ROUTINE Directive, Example "routine.rex"

```
r=" 1 "
s=2
say "r="pp(r)
say "s="pp(s)
say
say "The result of 'r || 3 ' is:" pp(r || 3 )
say "The result of 's || 3 ' is:" pp(s || 3 )
say "The result of 'r + 3'   is:" pp(r +  3)
say "The result of 's + 3'   is:" pp(s +  3)
say
say "The result of 'r s'     is:" pp(r s)
say "The result of 'r || s'  is:" pp(r || s)
say "The result of 'r+s'     is:" pp(r+s)

::routine pp              -- enclose argument in square brackets
  parse arg value
  return "["value"]"

/* yields:

   r=[ 1 ]
   s=[2]

   The result of 'r || 3 ' is: [ 1 3]
   The result of 's || 3 ' is: [23]
   The result of 'r + 3'   is: [4]
   The result of 's + 3'   is: [5]

   The result of 'r s'     is: [ 1  2]
   The result of 'r || s'  is: [ 1 2]
   The result of 'r+s'     is: [3]
*/
```

::ROUTINE Directive, Example "toolpackage.rex"

```
-- collection of useful little Rexx routines

::routine pp public -- enclose argument in square brackets
  parse arg value
  return "["value"]"

::routine quote public -- enclose argument in double-quotes
  parse arg value
  return '"' || value || '"'
```

::ROUTINE Directive, Example "call_package.rex"

```
call toolpackage.rex   -- get access to public routines in "toolpackage.rex"
say quote('hello, my beloved world')

r=" 1 "
s=2
say "r="pp(r)
say "s="pp(s)
say
say "r="quote(r)
say "s="quote(s)
say
say "The result of 'r || 3 ' is:" pp(r || 3 )
say "The result of 's || 3 ' is:" quote(s || 3 )
say "The result of 'r + 3'   is:" pp(r +  3)
say "The result of 's + 3'   is:" quote(s +  3)

/* yields:

   "hello, my beloved world"
   r=[ 1 ]
   s=[2]

   r=" 1 "
   s="2"

   The result of 'r || 3 ' is: [ 1 3]
   The result of 's || 3 ' is: "23"
   The result of 'r + 3'   is: [4]
   The result of 's + 3'   is: "5"
*/
```

::REQUIRES Directive

- Syntax

  ::requires package

  – Interpreter in phase 2 will either

    - Call (execute) the Rexx program named "package" on behalf of the current Rexx program and make all its public routines and classes upon return directly available to us
    - Or if the interpreter already required that "package" will *immediately* make all its public routines and classes available to us

      – In this case "package" will ***not*** *be called/executed anymore!*

::REQUIRES-Directive, Example "requires_package.rex"

```
say quote('hello, my beloved world')

r=" 1 "
s=2
say "r="pp(r)
say "s="pp(s)
say
say "r="quote(r)
say "s="quote(s)
say
say "The result of 'r || 3 ' is:" pp(r || 3 )
say "The result of 's || 3 ' is:" quote(s || 3 )
say "The result of 'r + 3'   is:" pp(r +  3)
say "The result of 's + 3'   is:" quote(s +  3)

::requires toolpackage.rex -- get access to public routines in "toolpackage.rex"

/* yields:

   "hello, my beloved world"
   r=[ 1 ]
   s=[2]

   r=" 1 "
   s="2"

   The result of 'r || 3 ' is: [ 1 3]
   The result of 's || 3 ' is: "23"
   The result of 'r + 3'   is: [4]
   The result of 's + 3'   is: "5"
*/
```

The Message Paradigm, 1

- A programmer sends messages to objects
  - The *object* looks for a method routine with the same name as the received message
  - If arguments were sent the *object* forwards them
  - The *object* returns any value the method routine returns
- *C.f. <https://en.wikipedia.org/wiki/Alan_Kay>*
  - One of the fathers of "object-orientation"
- Programming languages with this paradigm, e.g.
  - Smalltalk, Objective C, ...

The Message Paradigm, 2

ooRexx

- Proper message operator "~" (tilde, "twiddle")
- In ooRexx everything is an *"object"*
  - Hence one can send messages to everything!
- Example

```
say "hi, Rexx!"~reverse

-- same as in classic REXX:

say reverse("hi, Rexx!")

-- both yield (actually run the same code):
!xxeR ,ih
```

The Message Paradigm, 3
ooRexx

- Creating "*values*" a.k.a. "*objects*", "*instances*"

  Classic Rexx-style (strings only)

  ```
  str="this is a string"
  ```

  ooRexx-style (*any* class/type including .string class)

  ```
  str=.string~new("this is a string")
  ```

About Classic REXX Structures, 1
Important Usage of Stems

- Whenever structures ("records") are needed, *stems* get used in classic REXX
- Example
  - A person may have a name and a salary, e.g.

    p.name = "Doe, John"

    p.salary= "10500"

  - E.g. a collection of data with a person structure

    p.1.name = "Doe, John"; p.1.salary=10500

    p.2.name = "Doe, Mary"; p.2.salary=8500

    p.0 = 2

About Classic REXX Structures, 2
Important Usage of Stems

- Whenever *structures* ("*records*") need to be processed, *every* Rexx programmer *must* know the *exact stem encoding!*
- *Everyone* must implement routines like increasing the salary *exactly* like everyone else!
- If *structures* are simple and not used in many places, this is o.k., but the more complex the more places the *structure* needs to be accessed, the more error prone this becomes!

About ooREXX *Structures*, 1

Classes (Types, Structures)

- Any object-oriented language makes it easy to define and implement *structures*!
  - That is what they were designed for!
- The *structure* ("*class*") usually consists of
  - *Attributes* (data elements like "name", "salary"), a.k.a. "*object variables*", "*fields*", ...
  - Routines (like "increaseSalary"), a.k.a. "*methods*", "*method routines*", ...

About ooREXX *Structures*, 2

Classes (Types, Structures)

- **::CLASS** Directive
  - Denotes the name of the *structure*
  - Can optionally be public
- **::ATTRIBUTE** Directive
  - Denotes the name of a *data element*, *field*
- **::METHOD** Directive
  - Denotes the name of a routine of the *structure*
  - Defines the *Rexx code* to be run, when invoked

About ooREXX *Structures*, 3

Classes (Types, Structures)

- Once
  - A *structure* ("*class*", "*type*" both of which are synonyms of each other) got defined
  - One can create an *unlimited (!) number* of persons ("*instances*", "*objects*", "*values*", all of which are synonyms)
    - *Each* person will have its *own copy of attributes (data elements, fields)*
    - *All* persons will share/use the *same method routines* that got defined for the structure (class, type)

ooRexx Structure "Person"
"personstructure.rex"

```
p=.person~new("Doe, John", 10500)
say "name:  " p~name
say "salary:" p~salary

::class person              -- define the name

::attribute name            -- define a data element, field, object variable
::attribute salary          -- define a data element, field, object variable

::method    init            -- constructor method routine (to set the attribute values)
  expose name salary        -- establish direct access to attributes
  use arg name, salary      -- fetch and assign attribute values

/* yields:

   name:   Doe, John
   salary: 10500

*/
```

Defining the ooRexx Class (Type) "person.cls"

```
::class person PUBLIC    -- define the name, this time PUBLIC

::attribute name         -- define a data element, field, object variable
::attribute salary       -- define a data element, field, object variable

::method    init         -- constructor method routine (to set the attribute values)
  expose name salary     -- establish direct access to attributes
  use arg name, salary   -- fetch and assign attribute values
```

Defining the ooRexx Class (Type) "requires_person.rex"

```
p.1 = .person~new("Doe, John", 10500)
p.2 = .person~new("Doe, Mary",  8500)
p.0 = 2

sum=0
do i=1 to p.0
   say p.i~name "earns:" p.i~salary
   sum=sum+p.i~salary
end
say
say "Sum of salaries:" sum

::requires person.cls    -- get access to the public class "person" in "person.cls"

/* yields:

   Doe, John earns: 10500
   Doe, Mary earns: 8500

   Sum of salaries: 19000
*/
```

ooRexx *Classes* and Beyond ...

- ooRexx comes with a wealth of *classes*
  - A lot of tested functionality for "free" ;-)
  - E.g., the collection classes augment what stems are capable of doing!
    - Explore the collection classes and you will immediately be much more productive!
    - If seeking arrays, you have them: .Array class
  - Consult the pdf-books coming with ooRexx, e.g.,
    - "ooRexx Programming Guide" (rexxpg.pdf)
    - "ooRexx Reference Guide" (rexxref.pdf)

Roundup

- ooRexx is great and compatible to classic REXX
  - You can continue to program in classic REXX, yet use ooRexx on Linux, MacOS, Windows, s390x...
- ooRexx adds a lot of flexibility and power to the REXX language and to your fingertips
  - One can take advantage of all of it immediately
  - Simple to use because of the *message paradigm*
    - Send ooRexx *messages* to Windows and MS Office ...
    - Send ooRexx *messages* to Java ...
    - Send ooRexx *messages* to ...
- ***Get it and have fun! :-)***

Links

- RexxLA-Homepage (non-profit SIG, owner of ooRexx, BSF4ooRexx)
  <http://www.rexxla.org/>
- ooRexx 5.0 beta on Sourceforge
  <https://sourceforge.net/projects/oorexx/files/oorexx/5.0.0beta/>
  - Introduction to ooRexx on Windows, Slides ("Business Programming 1")
    - <http://wi.wu.ac.at/rgf/wu/lehre/autowin/material/foils/>
- BSF4ooRexx on Sourceforge (ooRexx-Java bridge)
  <https://sourceforge.net/projects/bsf4oorexx/>
  - Introduction to BSF4ooRexx (Windows, Mac, Unix), Slides ("Business Programming 2")
    - <http://wi.wu.ac.at/rgf/wu/lehre/autojava/material/foils/>
- Student's work, including ooRexx, BSF4ooRexx
  <http://wi.wu.ac.at/rgf/diplomarbeiten/>
- JetBrains "IntelliJ IDEA", powerful IDE for all operating systems
  - <https://www.jetbrains.com/idea/download>, free "Community-Edition"
    - Students and lecturers can use the professional edition for free
  - Alexander Seik's ooRexx-Plugin with readme (as of: 2021-11-07)
    - <https://sourceforge.net/projects/bsf4oorexx/files/Sandbox/aseik/ooRexxIDEA/GA/2.0.4/>
- Introduction to ooRexx (254 pages, covers ooRexx 4.2)
  <https://www.facultas.at/Flatscher>