2023 RexxLA International Rexx Language Symposium Proceedings

René Vincent Jansen (ed.)

21 May 2023

THE REXX LANGUAGE ASSOCIATION
RexxLA Symposium Proceedings Series
ISSN 1534-8954

Publication Data

A publication of **RexxLA Press**

The responsible publisher of this edition is identified as *IBizz IT Services and Consultancy*, Amsteldijk 14, 1074 HR Amsterdam, a registered company governed by the laws of the Kingdom of The Netherlands.

The RexxLA Symposium Series is registered under ISSN 1534-8954
The 2023 edition is registered under ISBN 978-94-036-5010-4

2023-05-20 First printing

Introduction

History of the International Rexx Language Symposium

In 1990, Cathie Dager of SLAC[1] convened the organizing committee for the first independent Rexx[2] Symposium for Developers and Users. SLAC continued to organize this annual event until the middle of the 1990's when the RexxLA took over that responsibility. Symposia have been held annually since 1990.

About RexxLA

During the 1993 Symposium in La Jolla, California, plans for a Rexx User Group materialized. The Rexx Language Association (RexxLA), as it was called, is an independent, non-profit organization dedicated to promoting the use and understanding of the Rexx programming language. RexxLA manages several open source implementations of Rexx.

The selection procedure

Presentation proposals are solicited yearly using a CFP[3] procedure, after which the RexxLA symposium comittee reviews them and votes which presentations are selected for the symposium. The presentations are peer reviewed before being presented. Presenters are not compensated for their presentations.

Location

The 2023 symposium was held Amsterdam, and Online from 14 May 2023 to 17 May 2023.

Organizing Committee

- Chip Davis
- Jon Wolfers
- Mark Hessling
- René Jansen
- Terry Fuller

[1] Stanford Linear Accelerator Center, since 2008 SLAC National Accelerator Laboratory
[2] Cowlishaw, M. F., **The REXX Language** (second edition), ISBN 0-13-780651-5, Prentice-Hall, 1990.
[3] Call For Papers.

Contents

1

Tutorial: From Rexx to ooRexx – Rony G. Flatscher

Date and Time

14 May 2023, 13:15:00 UTC

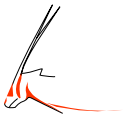
Presenter

Rony G. Flatscher

Presenter Details

Rony works as a professor for Business informatics ("Wirtschaftsinformatik") at the Vienna University of Economics and Business Administration (Wirtschaftsuniversität Wien) and uses Open Object Rexx for teaching Business Administration and MIS students the object-oriented paradigm, as well as remote-controlling (automating) Windows and Windows end-user applications (e.g. MS Office, Open Office) as well as Java and Java applications (he is the author of BSF4ooRexx, the ooRexx-Java bridge, which uses Apache BSF and had Rony invited to become an ASF member). He consults and trains in all of his research fields.

"From Rexx to ooRexx"



The 2023 International Rexx Symposium

Almere, The Netherlands

May 14th – May 17th 2023

© 2023 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
  - (::ANNOTATE, ::CONSTANT, ::OPTIONS, ::RESOURCE)
- 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 to RexxLA
  - "Open Object Rexx (ooRexx) 3.0"
    - Open source version of IBM's Object REXX
    - Released by RexxLA: 2005-03-25
  - ooRexx 3.1 (2006), ooRexx 3.2 (2008)
  - ooRexx 4.0 (2009)
    - New kernel, 32- and 64-bit became possible
  - ooRexx 4.1 (2011), ooRexx 4.2 (2014)
  - ooRexx 5.0 (2022)

Some New Features

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

    (Load) Phase 1: Full syntax check of the Rexx program upfront

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

    (Execution) 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 "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 "stemusearg.rex": *No* EXPOSE

```
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 "stemroutine1.rex": *No* EXPOSE

```
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 "stemroutine2.rex": *No* EXPOSE

```
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 the *setup phase* (phase 2) of startup
  - After all directives got carried out, the *execution phase of the Rexx program* starts by executing the first line
- An ooRexx program with directives
  - Defines a "package" of routines and classes
  - Rexx code before the first directive is also 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"
*/
```

8

::REQUIRES Directive

- Syntax

  ::requires "package.rex"

  – Interpreter in (setup) phase 2 will either
    - Call (execute) the Rexx program in the file named "package.rex" 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 has *required* that "package.rex" it will *immediately* make all its public routines and classes available to us
      – In this case "package.rex" 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 Smalltalk's "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*", "*type*") usually consists of
  - *Attributes* (data elements like "name", "salary"), a.k.a. "*object variables*", "*fields*", ...
  - *Method* routines (like "increaseSalary")

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[15]" (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! :-)***

29

Links

- RexxLA-Homepage (non-profit SIG, owner of ooRexx, BSF4ooRexx)
  <http://www.rexxla.org/>
- ooRexx 5.0 on Sourceforge
  <https://sourceforge.net/projects/oorexx/files/oorexx/5.0.0/>
  - Introduction to ooRexx on Windows, Slides ("Business Programming 1")
    - <http://wi.wu.ac.at/rgf/wu/lehre/autowin/material/foils/>
- BSF4ooRexx850 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: 2023-05-09)
    - <https://sourceforge.net/projects/bsf4oorexx/files/Sandbox/aseik/ooRexxIDEA/GA/2.2.0/>
- Introduction to ooRexx (254 pages, covers ooRexx 4.2)
  <https://www.facultas.at>

30

2

Tutorial: Stems a Different Way - Introducing 'oo' in ooRexx – Rony G. Flatscher

Date and Time

14 May 2023, 14:15:00 UTC

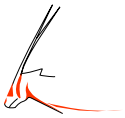
Presenter

Rony G. Flatscher

Presenter Details

Rony works as a professor for Business informatics ("Wirtschaftsinformatik") at the Vienna University of Economics and Business Administration (Wirtschaftsuniversität Wien) and uses Open Object Rexx for teaching Business Administration and MIS students the object-oriented paradigm, as well as remote-controlling (automating) Windows and Windows end-user applications (e.g. MS Office, Open Office) as well as Java and Java applications (he is the author of BSF4ooRexx, the ooRexx-Java bridge, which uses Apache BSF and had Rony invited to become an ASF member). He consults and trains in all of his research fields.

Stems a Different Way - Introducing 'oo' in 'ooRexx



The 2023 International Rexx Symposium

Almere, The Netherlands

May 14th – May 17th 2023

© 2023 Rony G. Flatscher (Rony.Flatscher@wu.ac.at, http://www.ronyRexx.net)

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

Overview

- Data type, abstract data type
  - REXX: strings, stem variables ("stems")
  - ooRexx in addition: Classes, Attributes, Methods
- Collecting values
  - REXX (and ooRexx): "Stem arrays"
  - ooRexx: *real* arrays
- Roundup

Data Type (DT), 1

- Data type
  - Defines set of valid values
  - Defines operations with those values (e.g. addition, concatenation)
  - Example 1
    - Data type Birthday
      - Defined values consist of a combination of
        - A valid date attribute and a valid time attribute
      - Defined operations
        - Set, query and change its date and time attributes

Data Type (DT), 2

  - Example 2
    - Data type Person
      - Defined values consist of a combination of
        - firstName, lastName, salary attributes
      - Defined operations
        - Set, query and change its firstName, lastName, salary attributes
        - increaseSalary

Data Type (DT), 3
REXX-Problems

- No means to *explicitly* define *data structures*
- No means to *explicitly* define *operations* restricted to certain data types
- *Data structures* can be mimicked with
  - Strings
  - Stem variables

Data Type (DT), 4
REXX, Possible Solution, 1

- Encode a data structure in a string
  - E.g. for the data type Birthday

```
"2005-09-01 16:00"
"2008-02-29 19:19"
```

  - E.g. for the data type Person

```
"Albert Einstein 45000"
"Vera WithAnyName 25000"
```

- Processing possible *only* if *everyone* knows
  - Number and sequence of encoded fields/attributes
  - Where the fields/attributes start and end

Data Type (DT), 5
REXX, Possible Solution, 2

- Represent a data structure with a stem variable
  - E.g. for the data type Birthday

```
birthday.0date="2005-09-01"; birthday.0time="16:00"
birthday.0date="2008-02-29"; birthday.0time="19:19"
```

  - E.g. using a "stem-array" for data type Person

```
person.1.firstname="Albert"; person.1.lastname="Einstein";
person.1.salary="45000"
person.2.firstname="Vera"; person.2.lastname="WithAnyName";
person.2.salary=25000
```

- Processing possible if name of fields/attributes is known!

Data Type (DT), 6
REXX, Considerations

- DT-Structure
  - Encoding as strings or in stems
    - Crook, as implementation dependent!
    - Error-prone!
- DT-Operations
  - No means to define operations restricted to data types!
- No means to hide values/instances of data types from the programmer in order to shelter them from programming errors!
  - *Everyone* must know implementation (encoding) details!

Abstract Data Type (ADT), 1

- Abstract Data type (ADT)
  - *Schema* for implementing data types
    - Definition of *attributes*
      - Yields the data structure
    - Definition of *operations* ("methods")
      - Yields the *behaviour*
  - *Schema* must be implemented
    - REXX is *not designed* for it, hence not suitable!
    - ooRexx is an object-oriented language and *hence predestined !* :-)

Abstract Data Type (ADT), 2

- Implement any ADT in ooRexx with *directives*

  ::CLASS name

  ::ATTRIBUTE name

  ::METHOD name

  Hint: Rexx method routines are able to directly access attributes of its class by using as their first instruction the EXPOSE keyword instruction listing the attributes

- "Instances" ("objects", "values")
  - Distinct to any other instance/object/value
  - Possess all the same structure and behaviour

Abstract Data Type (ADT), 3
Implementing ADT "Birthday", 1

```
/* an ooRexx program that implements an ADT! */

::CLASS     BirthDay       /* name of the structure/class              */
::ATTRIBUTE date
::ATTRIBUTE time
```

- Creating values/instances/objects
  - Simply send the message NEW to the Rexx-Class named .Birthday
  - Message operator is the tilde (~), hence e.g.

```
bd1=.Birthday~new /* create a value        */
bd2=.Birthday~new /* create another value */
...
```


Abstract Data Type (ADT), 4
Implementing ADT "Birthday", 2

```
/* an ooRexx program that implements an ADT! */
bd1=.BirthDay~new
bd1~date="2005-09-01"
bd1~time ="16:00"

bd2=.BirthDay~new
bd2~date="2008-02-29"
bd2~time ="19:19"

say "BirthDay 1:" bd1~date bd1~time
say "BirthDay 2:" bd2~date bd2~time
```

```
::CLASS     BirthDay       /* name of the structure/class              */
::ATTRIBUTE date
::ATTRIBUTE time
```

Output:

```
BirthDay 1: 2005-09-01 16:00
BirthDay 2: 2008-02-29 19:19
```

Excursus: Scopes, 1
REXX

- Scopes
  - Determine the visibility of variables, attributes, routines and classes
- REXX-Scopes
  - *Standard-Scope*
    - Labels and variables are visible throughout the program
  - *Procedure-Scope*
    - Variables of internal routines followed by the PROCEDURE keyword statement are locally visible only

Excursus: Scopes, 2
ooRexx, 1

- Additional ooREXX-Scopes
  - *Program-Scope*
    - All Routine-directives and Class-directives of a program are visible in the entire program
    - In addition all public routines and public classes defined in another program become visible and directly accessible after that program got invoked/required !

Excursus: Scopes, 3 ooRexx, 2

- Additional ooREXX-Scopes
  - *Routine-Scope*
    - Managed as if it was a proper REXX-Programm
      - *Standard-Scope*
    - Therefore can include internal routines
      - *Procedure-Scope*
    - Can access all the routines and classes of the program
      - *Program-Scope*

Excursus: Scopes, 4 ooRexx, 3

- Additional ooREXX-Scopes
  - *Method-Scope*
    - Like *Routine-Scope*
    - In addition
      - Direct access to attributes of its class possible
        - First instruction must be the EXPOSE-keyword instruction with blank delimited attribute names

Excursus: Scopes, 5 Overview

- REXX and ooRexx
  - *Standard-scope*: labels, variables
  - *Procedure-scope*: local variables
- ooRexx
  - *Programm-scope*: routines, classes
  - *Routine-scope*
    - Like a proper program
    - Scopes: *Standard*, *Procedure*, *Program*
  - *Method-Scope*
    - Like *Routine-Scope*
    - Additionally: EXPOSE allows direct access to attributes of the class

Abstract Data Type (ADT), 5 Implementing ADT "Person", 1

```
p1=.person~new              /* create an instance/value/object        */
p1~firstName ="Albert"
p1~lastName="Einstein"
p1~salary  =45000

p2=.person~new              /* create an instance/value/object        */
p2~firstName ="Vera"
p2~lastName="WithAnyName"
p2~salary  =25000

say "Person 1:       " p1~firstName p1~lastName p1~salary
say "Person 2:       " p2~firstName p2~lastName p2~salary
say "sum of salaries:" p1~salary + p2~salary
```

```
::CLASS Person              /* name of the structure/class             */
::ATTRIBUTE firstName
::ATTRIBUTE lastName
::ATTRIBUTE salary
```

Output:

```
Person 1:        Albert Einstein 45000
Person 2:        Vera WithAnyName 25000
sum of salaries: 70000
```

Abstract Data Type (ADT), 6
Implementing ADT "Person", 2

```
p1=.person~new              /* create an instance/value/object        */
p1~firstName ="Albert"
p1~lastName="Einstein"
p1~salary  =45000

p2=.person~new              /* create an instance/value/object        */
p2~firstName ="Vera"
p2~lastName="WithAnyName"
p2~salary  =25000

say "Person 1:           " p1~firstName p1~lastName p1~salary
say "Person 2:           " p2~firstName p2~lastName p2~salary
p1~increaseSalary(10000)    /* increase salary                        */
say "Person 1:        ->" p1~firstName p1~lastName p1~salary
say "sum of salaries: ->" p1~salary + p2~salary

::CLASS Person              /* name of the structure/class            */
::ATTRIBUTE firstName
::ATTRIBUTE lastName
::ATTRIBUTE salary

::METHOD increaseSalary     /* increaseSalary method                  */
  EXPOSE salary               /* access "salary" attribute directly   */
  USE ARG increaseBy          /* fetch increase amount                */
  salary=salary+increaseBy    /* add and save result in attribute     */
```

Output:

```
Person 1:           Albert Einstein 45000
Person 2:           Vera WithAnyName 25000
Person 1:        -> Albert Einstein 55000
sum of salaries: -> 80000
```


Fun with Methods: INIT, 1
Creating Objects/Instances/Values

- Objects/instances/values
  - Can be simply created by sending the message NEW to the class which will return a newly created value
- If a method INIT exists in the class then it will be invoked from the NEW method
  - If one supplies arguments to the NEW-message, then they will be forwarded to INIT in the same order!
  - The INIT-method carries also the name "constructor method" or short: "constructor"

Fun with Methods: INIT, 2
Creating Objects/Instances/Values

```
p1=.person~new("Albert", "Einstein", 45000)  /* create with values       */
p2=.person~new("Vera", "WithAnyName", 25000) /* create with values       */

say "Person 1:        " p1~firstName p1~lastName p1~salary
say "Person 2:        " p2~firstName p2~lastName p2~salary
say "sum of salaries:" p1~salary + p2~salary
```

```
::CLASS Person              /* name of the structure/class               */
::ATTRIBUTE firstName
::ATTRIBUTE lastName
::ATTRIBUTE salary

::METHOD increaseSalary     /* increaseSalary method                     */
  EXPOSE salary               /* access "salary" attribute directly      */
  USE ARG increaseBy          /* fetch increase amount                   */
  salary=salary+increaseBy    /* add and save result in attribute        */
```

```
::METHOD INIT               /* constructor method                        */
  EXPOSE firstName lastName salary    /* access attributes directly      */
  USE ARG firstName, lastName, salary /* assign arguments to attributes */
```

Output:

```
Person 1:         Albert Einstein 45000
Person 2:         Vera WithAnyName 25000
sum of salaries: 70000
```


Fun with Methods: UNINIT, 1
Destroying Objects/Instances/Values

- Objects/instances/values
  - If values are not referenced anymore then the "garbage collector" destroys them
- If a method with the name UNINIT exists in a class, then the garbage collector will invoke it right before destroying the value
  - E.g. useful to release global locks, writing logs etc.
  - The UNINIT-method is also known as the "destructor method" or short: "destructor"

Fun with Methods: UNINIT, 2

Destroying Objects/Instances/Values

```
p1=.person~new("Albert", "Einstein", 45000)  /* create with values       */
p2=.person~new("Vera", "WithAnyName", 25000) /* create with values       */

say "Person 1:       " p1~firstName p1~lastName p1~salary
say "Person 2:       " p2~firstName p2~lastName p2~salary
say "sum of salaries:" p1~salary + p2~salary
drop p2; drop p1             /* delete variables, objects become garbage */
call sysSleep 5              /* sleep five seconds                       */
say "end of main program!"

::CLASS Person               /* name of the structure/class              */
::ATTRIBUTE firstName
::ATTRIBUTE lastName
::ATTRIBUTE salary

::METHOD increaseSalary      /* increaseSalary method                    */
  EXPOSE salary              /* access "salary" attribute directly       */
  USE ARG increaseBy         /* fetch increase amount                    */
  salary=salary+increaseBy   /* add and save result in attribute         */

::METHOD INIT                /* constructor method                       */
  EXPOSE firstName lastName salary    /* access attributes directly        */
  USE ARG firstName, lastName, salary /* assign arguments to attributes    */

::METHOD UNINIT              /* destructor method                        */
  EXPOSE firstName lastName salary  /* access attributes directly         */
  say 'Object <'firstName lastName salary'> about to be destroyed...'
```

Output (maybe):

```
Person 1:        Albert Einstein 45000
Person 2:        Vera WithAnyName 25000
sum of salaries: 70000
end of main program!
Object <Vera WithAnyName 25000> about to be destroyed...
Object <Albert Einstein 45000> about to be destroyed...
```


Collecting Values, 1

- "Stem-arrays"
  - Convention
    - Stem variable with the tail "0" contains the sum of stored values starting with the tail "1"
  - Only possibility in REXX to collect and to process values
  - ooRexx allows for collecting any kind of values in such stem arrays

Collecting Values, 2 "Stem-Arrays", 1

```
person.1.firstName ="Albert"
person.1.lastName="Einstein"
person.1.slary   =45000        /* <-- typical typing error!              */

person.2.firstName ="Vera"
person.2.lastName="WithAnyName"
person.2.salary  =25000
person.0=2

do i=1 to person.0   /* iterate over all persons                         */
   say "Person #" i":" person.i.firstName person.i.lastName person.i.salary
end
```

Output:

```
Person # 1: Albert Einstein PERSON.1.SALARY
Person # 2: Vera WithAnyName 25000
```


Collecting Values, 3 "Stem-Arrays", 2

```
person.1=.person~new("Albert", "Einstein", 45000)
person.2=.person~new("Vera", "WithAnyName", 25000)
person.0=2

do i=1 to person.0   /* iterate over all persons                         */
   say "Person #" i":" person.i~firstName person.i~lastName person.i~salary
end

::CLASS Person              /* name of the structure/class               */
::ATTRIBUTE firstName
::ATTRIBUTE lastName
::ATTRIBUTE salary

::METHOD INIT               /* constructor method                        */
  EXPOSE firstName lastName salary     /* access attributes directly     */
  USE ARG firstName, lastName, salary  /* assign arguments to attributes*/
```

Output:

```
Person # 1: Albert Einstein 45000
Person # 2: Vera WithAnyName 25000
```

Collecting Values, 4 ooRexx

- ooRexx has *real* arrays !
  - Simple to create
    - ooRexx 5.0 even allows creating them from a list
  - Easy to use and to iterate over the collection
    - E.g. DO...OVER
- Hint
  - ooRexx comes with many different kinds of classes/types that allow one to collect and process values!

Collecting Values, 5 ooRexx Has *Real* Arrays, 1

```
persons=.Array~new          /* create an array                          */
persons[1]=.person~new("Albert", "Einstein", 45000)
persons[2]=.person~new("Vera", "WithAnyName", 25000)

do p over persons           /* iterate over all persons                 */
   say "Person:" p~firstName p~lastName p~salary
end
```

```
::CLASS Person              /* name of the structure/class              */
::ATTRIBUTE firstName
::ATTRIBUTE lastName
::ATTRIBUTE salary

::METHOD INIT         /* constructor method                             */
  EXPOSE firstName lastName salary    /* access attributes directly     */
  USE ARG firstName, lastName, salary /* assign arguments to attributes */
```

Output:

```
Person: Albert Einstein 45000
Person: Vera WithAnyName 25000
```

Collecting Values, 6
ooRexx

- Arrays can be *sorted*! :)
  - Simply define a method named compareTo
    - Will receive the other value to compare to by the sort method defined in the Array class
    - Method must return the value
      - "1", if our value is regarded to be larger
      - "0", if both values are regarded to be the same
      - "-1", if other value is regarded to be larger

Collecting Values, 7
ooRexx Has *Real* Arrays, 2

```
persons=.Array~new          /* create an array                          */
persons[1]=.person~new("Albert", "Einstein", 45000)
persons[2]=.person~new("Vera", "WithAnyName", 25000)

do p over persons~sort      /* iterate over all persons in sorted order */
   say "Person:" p~firstName p~lastName p~salary
end
```

```
::CLASS Person              /* name of the structure/class              */
::ATTRIBUTE firstName
::ATTRIBUTE lastName
::ATTRIBUTE salary

::METHOD INIT         /* constructor method                             */
  EXPOSE firstName lastName salary     /* access attributes directly     */
  USE ARG firstName, lastName, salary  /* assign arguments to attributes*/
```

```
::METHOD compareTo          /* comparison method for sorting            */
  EXPOSE salary             /* access attribute directly                */
  use arg other             /* other person to compare to               */
  if other~salary<salary then return 1 /* our salary is greater          */
  if other~salary=salary then return 0 /* salaries are the same          */
  return -1                 /* other salary is greater                  */
```


Output:

```
Person: Vera WithAnyName 25000
Person: Albert Einstein 45000
```