

# "ooRexx Tutorial"

37<sup>th</sup> International Rexx Symposium

May 3<sup>rd</sup> – May 6<sup>th</sup> 2026

Barcelona, Spain



- Brief History
- Rexx Basics
- Object Rexx
  - Some new features like
    - `USE ARG`
  - New: Directives
    - `::ROUTINE, ::REQUIRES`
    - `::CLASS, ::ATTRIBUTE, ::METHOD`
    - `(::ANNOTATE, ::CONSTANT, ::OPTIONS, ::RESOURCE)`
- Roundup

- Created for IBM mainframes to make programming easier compared to the rather awkward EXEC2
  - **Rexx design goals:** "human centric", "keep the language small", "easy to learn", "easy to understand hence easy to maintain"
  - Rexx is **still instrumental for IBM mainframe operating systems** today!
- Extremely successful in the 80'ies
  - Companies selling Rexx interpreters successfully, **ANSI/INCITS standard** (!)
- Object-oriented successor ("**Object Rexx**") in the 90'ies
  - **Open-sourced** in 2005 by RexxLA.org – "open object Rexx" (ooRexx)
    - Available for **all major operating systems**
    - Possible to programme even MS Windows applications via **OLE** ...



- "Everything is a string"
  - If a string represents a number, one can carry out arithmetic
- Three instruction types
  - 1) Assignment
    - Variable name followed by the assignment operator (=) and an expression
  - 2) Keyword instruction
    - Keywords are English words conveying the intent of the keyword instruction, e.g. **SAY, DO, IF, LOOP, CALL, PARSE, SELECT, ITERATE, LEAVE, INTERPRET, ...**
    - Makes Rexx code legible as if it was pseudocode
  - 3) Commands
    - A string passed to the operating system for execution (as if typed in a window)

- White space can be freely used to format code for better legibility
  - Space around operators gets removed
  - White space between symbols will be reduced to a single space serving as concatenation operator
  - Hence indentations with white space not significant
- Case of symbols irrelevant
  - Rexx uppercases everything outside of quoted strings
  - No (frustrating) casing errors for novices

```
sum = 17 + 19
      hint = "/ 17+19:" sum
say hint "/" upper("aü ß äöü ÄÖÜ A/ ? \\--// :-)")
```



```
SUM=17+19
HINT="/ 17+19:" SUM
SAY HINT "/" UPPER("aü ß äöü ÄÖÜ A/ ? \\--// :-)")
```

### Output:

```
/ 17+19: 36 / Aü ß äöü ÄÖÜ A/ ? \\--// :-)
```

- Rexx nutshell examples to stress fundamental concepts
  - Illustrate the Rexx language
    - Code intuitive and easy understandable as it looks like pseudo code
  - Same examples in the popular Python language to allow direct comparisons
    - Cannot be understood without an introduction to many concepts of the Python language



# Nutshell Example, 1

## Instructions



```

/* an assignment instruction:      */
a="hello world" /* assigns "hello world" to a variable named a */

/* a keyword instruction:         */
say a           /* output: hello world */

/* a command instruction:        */
/* a command (could be typed into a command line window) */
"echo Hello World 2" /* execute command */
/* variable RC contains the command's return code, 0 means success */
if rc=0 then say "success!"
else say "some problem occurred, rc="rc /* show return code */

```

### Output:

```

hello world
Hello World 2
Success!

```



```

# an assignment instruction
a="hello world" # assigns "hello world" to a variable named a

# no keyword instruction for output, using built-in function print()
print(a)

# no command instruction using module subprocess instead
import subprocess # import subprocess module
# execute command
completedProcess=subprocess.run("echo Hello World 2", shell=True) # run
rc=completedProcess.returncode # fetch return code, an int
if rc==0:
    print("found!") # indentation mandatory (forcing a block)
else: # must use + (concatenation operator) with str() function
    print("some problem occurred, rc="+str(rc)) # turn rc into a string

```

### Output:

```

hello world
Hello World 2
Success!

```



## Blocks, Selection, Multiple Selections



```
max=5          /* number of repetitions */
loop a=1 to max /* loop block */
  select      /* nested block # 1 */
    when a=1 then say a": first round"
    when a=2 then say a": second round"
    when a=3 then say a": third round"
    otherwise say "(a="a")"
  end

  if a=max then
  do          /* nested block # 2 */
    say "-> a=max"
    say "-> last round!"
    say "-> loop will end"
  end
end
```

### Output:

```
1: first round
2: second round
3: third round
(a=4)
(a=5)
-> a=max
-> last round!
-> loop will end
```



```
max=5          # number of repetitions
for a in range(1,max+1): # loop with range() function, must add 1 to max
    # must use str() function with + (concatenation operator)
    match a:      # must be indented, "match" needs Python 3.10 or higher
      case 1: print(str(a)+": first round") # nested block # 1
      case 2: print(str(a)+": second round") # nested block # 1
      case 3: print(str(a)+": third round") # nested block # 1
      case _: print("(a="+str(a)+")") # default, nested block # 1

    if a==max: # must be indented, must use == instead of =
      print("-> a==max") # nested block # 2
      print("-> last round!") # nested block # 2
      print("-> loop will end") # nested block # 2
```

### Output:

```
1: first round
2: second round
3: third round
(a=4)
(a=5)
-> a==max
-> last round!
-> loop will end
```



```
text = " John   Doe   Vienna Austria"
parse var text firstName lastName city country
say "first name:" firstName", last name:" lastName", city:" city
```

```
text = "Mary Doe Tokyo Japan"
parse var text firstName lastName city . /* ignore country */
say "first name:" firstName", last name:" lastName", city:" city
```

### Output:

```
first name: John, last name: Doe, city: Vienna
first name: Mary, last name: Doe, city: Tokyo
```

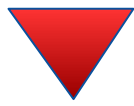


```
text      = " John   Doe   Vienna Austria"
words     = text.split() # create list of words
firstName = words[0]     # assign to variable
lastName  = words[1]     # assign to variable
city      = words[2]     # assign to variable
print("first name:",firstName+",", "last name:",lastName+",", "city:",city)
```

```
text = "Mary Doe Tokyo Japan"
words = text.split() # create list of words
# assign multiple elements in a single statement
firstName, lastName, city = [words[i] for i in (0, 1, 2)]
print("first name:",firstName+",", "last name:",lastName+",", "city:",city)
```

### Output:

```
first name: John, last name: Doe, city: Vienna
first name: Mary, last name: Doe, city: Tokyo
```



# ooRexx: Some New Features



- Compatible with classic Rexx, TRL 2
  - **New** sequence of execution of Rexx programs:
    - Phase **1 (load)**: **Full syntax check** of the Rexx program upfront
    - Phase **2 (setup)**: Interpreter carries out all directives (lead in with ":::")
    - Phase **3 (execution)**: 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! */

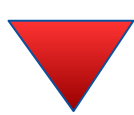
do i=1 to s.0   /* iterate over all stem array entries */
  say "#" i:" s.i
end

::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()

*/
```

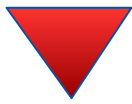
# ▼ About Directives in ooRexx

- 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"

- 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]
```

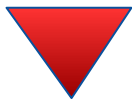
```
The result of 'r s' is: [ 1 2]  
The result of 'r || s' is: [ 1 2]  
The result of 'r+s' is: [3]
```

# ▼ ::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.rex"`

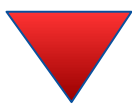
- Interpreter in (setup) phase 2 will either

- 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!

20



# ::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"
```

- 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](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, ...

- 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
```

- 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*")

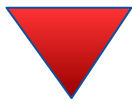
## ▼ About ooREXX *Structures*, 2 Classes (Types, Structures)

- **::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
```

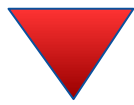
```
::attribute salary -- define a data element, field, object variable
```

```
::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" ([rexxref.pdf](#))

- 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! :-)***

- REXXLA-Homepage (non-profit SIG, owner of ooRexx, BSF4ooRexx)  
<<http://www.rexxla.org/>>
- OoRexx 5.1.0 on Sourceforge  
<<https://sourceforge.net/projects/ooorexx/files/ooorexx/5.1.0/>>
  - 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/bsf4ooorexx/>>
  - 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: 2025-05-07)
    - <<https://sourceforge.net/projects/bsf4ooorexx/files/Sandbox/aseik/ooRexxIDEA/GA/2.5.0/>>
- "Introduction to Rexx and ooRexx" (254 pages, covers ooRexx 4.2)  
Google et.al., or, <<https://www.facultas.at>>