

An Introduction to Procedural and Object-oriented Programming (ooRexx) 1

Introduction, Overview,
Statement, Procedure, Function

Prof. Rony G. Flatscher

Overview, 1

- Course
 - Basic concepts of the object-oriented paradigm
 - Standard application systems
 - Scripting language
 - Automation ("remote controlling") of applications
 - Automation of operating systems like Linux or Windows
 - Foils
 - <http://wi.wu-wien.ac.at/rgf/wu/lehre/autowin/material/foils/>
 - <http://wi.wu-wien.ac.at/rgf/wu/lehre/autojava/material/foils/>
 - Exercises
 - <http://wi.wu-wien.ac.at/rgf/wu/lehre/autowin/material/exercises/>

Overview, 2

- Why Rexx? Why Object Rexx?
 - Simple syntax ("human-centric" language)
 - Easy and quick to learn
 - Powerful object-model
 - All important concepts of the OO-paradigm available
 - Windows Scripting Engine (WSE)
 - Full automation of Windows applications possible
- Availability of Software
 - <http://www.ooRexx.org>

Overview, 3

- "Interactive" lecture
 - Please ask questions!
 - Do not fear "wrong", "easy" or "ridiculous " questions
 - Questioner concentrates on the answer (easier and more thorough learning of new concepts)
 - Questioner usually can rely on the fact that there are others who would like to place the same question (but don't dare/bother to do so)

History, 1

<http://www2.hursley.ibm.com/rexx/rexxhist.htm>

- 1979 - IBM (Mike F. Cowlshaw, IBM-Fellow)
 - Successor of a rather cryptic script language ("EXEC") on IBM mainframes
 - Goal: to create a "human-centric" language
 - Interactive (Interpreter)
 - REXX: Acronym for "REstructured eXtended eXecutor"
- Since 1987 IBM's "SAA" (System Application Architecture) "Procedural Language"
 - Strategic script language for all IBM platforms
 - Numerous commercial and open source versions of the language, available for practically all operating systems there are
- ANSI Rexx Standard in 1996
 - ANSI "Programming Language - REXX", X3.274-1996

History, 2

- Since the beginning of the 90ies
 - Going back on an initiative of the powerful IBM user interest group "SHARE" development of an object-oriented version of REXX started
- "Object-based REXX" a.k.a. "Object REXX"
 - Fully compatible with classic ("procedural") Rexx
 - Internally fully object-oriented
 - *All classic Rexx statements are transformed into object-oriented ones internally!*
 - Powerful object model (e.g. meta-classes, multiple inheritance)
 - Still a simple syntax
 - Availability
 - 1997 part of OS/2 Warp 4 (free) and free for Warp 3 (with [SOM](#))
 - 1998 AIX (first evaluation version) and [Linux](#) (free)
 - 1998 for Windows 95 and Windows NT (with [OLEAutomation/ActiveX](#))

History, 3

- 1996 development of "NetRexx" by the original author of Rexx, Mike F. Cowlshaw
 - Java in the "clothes" of Rexx
 - NetRexx-programs are translated into Java byte code
 - Simpler programming of the Java VM due to the simpler Rexx syntax
 - ~30% less Code (syntactical elements) than Java
 - Due to the Rexx syntax, easier to learn for the programming novice
 - IBM handed over source code to RexxLA
 - June, 8th, 2011 opensource released by RexxLA
 - Kick-off for new developments
- URLs for Rexx, Object Rexx, NetRexx

<http://www.RexxLA.org/>

<http://www.ooRexx.org/>

<http://www.NetRexx.org/>

<news:comp.lang.rexx>

History, 4

- May 2004
 - The "Rexx Language Association" (RexxLA) and IBM start non-public negotiations about open-sourcing Object Rexx
 - IBM: Manfred Schweizer, manager of IBM's Rexx development team
 - RexxLA: Pam Taylor (experienced commercial manager, USA), Mark Hessling (maintainer of Regina and author of numerous Rexx libraries, Australia), Rony G. Flatscher (MIS professor, Austria/Europe)
- October 12th, 2004
 - IBM and the Rexx Language Association (<http://www.RexxLA.org>) announce that RexxLA will manage Object Rexx under the name "Open Object Rexx" ("ooRexx")
- April 2005
 - The Rexx Language Association releases the opensource version "Open Object Rexx" (ooRexx), homepage: <http://www.ooRexx.org>

Basics

Minimal Rexx-Program

```
/* a comment */  
SAY "Hello, my beloved world"
```

Output:

```
Hello, my beloved world
```

Basics

RexxTry.rex

- "RexxTry.rex"
 - Rexx-Program which allows interactive execution of Rexx statements
 - Transfers every Rexx statement entered through the keyboard to the Rexx interpreter for execution
 - Displays console-output or errors of entered Rexx statements
 - Program ends when user enters **EXIT** through the keyboard
 - Invoking the program via a command in a command line window:

```
rex  RexxTry.rex
```

```
rex  rexxtry
```

Basics

Notation of Program Text

- Upper or lowercase spelling irrelevant
 - All characters of a Rexx statement will be translated into uppercase and executed
 - Exception: Contents of a string remains unchanged
 - Strings are delimited by apostrophes (') or by quotes ("), e.g.

"Richard" , *'Richard'* , *"\{[]}\gulp!öäüß!{niX }"*

- Multiple blank characters are reduced to one blank
 - Example

```
say      "\{[]}\gulp!öäüß!{niX }"      reverse(      Abc  )
```

becomes:

```
SAY "\{[]}\gulp!öäüß!{niX }" REVERSE( ABC )
```

Basics

Characters

- Characters outside of strings and comments must be from the following character set
 - Blank
 - **a** thru **z**
 - **A** thru **Z**
 - **0** thru **9**
 - Exclamation mark (**!**), backslash (****), question mark (**?**), equal sign (**=**), comma (**,**), dash/minus (**-**), plus (**+**), dot (**.**), Slash (**/**), parenthesis (**()**), square parentheses (**[]**), asterisk (*****), tilde (**~**), semicolon (**;**), colon (**:**) and underline (**_**)

Basics

Variables

- Variables allow storing, changing, and retrieving strings with the help of a discretionary name called an *identifier*

```
A = "Hello, my beloved world"  
a="Hello, my beloved variable"  
A      =      a      "- changed again."  
say a
```

Output:

```
Hello, my beloved variable - changed again.
```

- Identifiers must begin with a letter, an exclamation mark, a question mark or an underline character, followed by one or more of these characters, digits, and dots.

Basics

Constants

- Constants never get their values changed
- It is possible to use literals which are string constants appearing verbatim in an expression
 - If one wishes to name constants, then there are two possibilities available
 - The constant value is assigned to a variable, the value of which never gets changed in the entire program, e.g.,

```
Pi = 3.14159
```

- The constant value can be saved in the local (`.local`) or global (`.environment`) environment and will be referred to by an "environment symbol", which always start with a dot, e.g.,

```
.local~pi = 3.14159 /* Store value 3.14159 in .local */  
say .pi           /* retrieves value 3.14159 from .local*/
```

Basics

Comments

- Comments may be nested and are allowed to span multiple lines, e.g.

```
say 3 + /* This /**/ is
      a      /* nested
      /* aha*/ comment*/ which spans
      multiple lines */ 4
```

Output:

7

- Line comments: at the end of a statement, comments follow after two consecutive dashes:

```
say 3 + 4 -- this yields "7"
```

Output:

7

Basics

Statements, 1

- Statements consist of all characters up to and including the semi-colon (;)
- There may an arbitrary number of statements on a line
- If the semi-colon is missing, then the end of a statement is assumed by the end of a line

```
/* Convention: A comment begins in 1. line, 1. column */  
SAY "Hello, my dear world";
```

Output:

```
Hello, my dear world
```


Basics

Statements, 2

- Statements may span multiple lines, but you need to indicate this with the continuation character
 - Comma or Dash as the last character on the line

```
/* Convention: A comment begins in 1. line, 1. column */  
SAY "Hello," -  
    "my beloved world";
```

Output:

```
Hello, my beloved world
```

Basics

Block

- A block is a statement, which may comprise an arbitrary number of statements
- A block starts with the keyword **DO** and ends with **END**

```
DO
  SAY "Hello," ;
  SAY "world" ;
END;
```

```
DO
  SAY "Hello,"
  SAY "world"
END
```

Output:

```
Hello,
world
```

Basics

Comparisons (Test Expressions), 1

- Two values (constant, variable, results of function calls) can be compared with the following (Infix) operators (Result: 0=false or 1=true)

=		equal
<>	!= \=	unequal
<		smaller
<=		smaller than
>		greater
>=		greater than

- Negation of Boolean (0=false, 1=true) values

\ Negator

Basics

Comparisons (Test Expressions), 2

- Boolean values can be combined

& "and" (`true`: if both arguments are true)

| "or" (`true`: if either argument are true)

&& "exclusive or" (`true`: if one argument is true and the other is false)

- Boolean combinations can be evaluated in a specific order if enclosed in parentheses:

```
0 & 1 | 1 Result: 1 (= true)
(0 & 1) | 1 Result: 1 (= true)
0 & (1 | 1) Result: 0 (= false)
```

Basics

Comparisons (Test Expressions), 3

a=1

b=2

x="Anton"

y=" Anton "

If **a = 1** then ...

Result: **1** (= true)

If **a = a** then ...

Result: **1** (= true)

If **a >= b** then ...

Result: **0** (= false)

If **x = y** then ...

Result: **1** (= true)

If **x == y** then ...

Result: **0** (= false)

a <= b & (a = 1 | b > a)

Result: **1** (= true)

\(a <= b & (a = 1 | b > a))

Result: **0** (= false)

\a

Result: **0** (= false)

Basics

Branch, 1

- A branch determines which statement (block) should be executed as a result of a comparison (of a Boolean value)
 - **IF** test_expression=.true **THEN** statement;
 - Example:

```
IF age < 19 THEN SAY "Young."
```
 - A branch can also determine what alternative statement (block) should be executed, in case the Boolean value is false
 - **IF** test_expression=.true **THEN** statement; **ELSE** statement;
 - Examples:

```
IF age < 19 THEN SAY "Young.";  
                ELSE SAY "Old."
```

```
IF age < 1 THEN  
DO  
    SAY "Hello,"  
    SAY "my beloved world"  
END
```

Basics

Branch, 2

- Multiple selections (**SELECT**)

SELECT

```
WHEN test_expression THEN statement ;  
WHEN test_expression THEN statement ;  
/* ... additional WHEN-statements */  
OTHERWISE statement ;
```

END

Example:

```
SELECT
```

```
WHEN age = 1 THEN SAY "Baby." ;  
WHEN age = 6 THEN SAY "Elementary school kid." ;  
WHEN age >= 10 THEN SAY "Big kid." ;  
OTHERWISE SAY "Unimportant." ;
```

```
END
```

Basics

Repetition, 1

- Principally a block can be executed repeatedly

```
DO 3  
    SAY "Aua! "  
    SAY "Oh! "  
END
```

Output:

```
Aua!  
Oh!  
Aua!  
Oh!  
Aua!  
Oh!
```


Basics

Repetition, 2

- Using a variable to control the number of repetitions

```
a = 3
...
DO a
    SAY "Aua! "; SAY "Oh! "
END
```

Output:

```
Aua!
Oh!
Aua!
Oh!
Aua!
Oh!
```

Basics

Repetition, 3

- Repetition using a control variable ("i" in this example)

```
DO i = 1 TO 3
  SAY "Aua! "; SAY "Oh! " i
END
```

Output:

```
Aua!  
Oh! 1  
Aua!  
Oh! 2  
Aua!  
Oh! 3
```

Basics

Repetition, 4

- Repetition using a control variable ("i" in this example)

```
DO i = 1 TO 3 BY 2  
    SAY "Aua! "; SAY "Oh! " i  
END
```

Output:

```
Aua!  
Oh! 1  
Aua!  
Oh! 3
```

Basics

Repetition, 5

- Repetition using a control variable ("i" in this example)

```
DO i = 3.1 TO 5.7 BY 2.1
    SAY "Aua!"; SAY "Oh!" i
END
```

Output:

```
Aua!  
Oh! 3.1  
Aua!  
Oh! 5.2
```

Basics

Repetition, 6

- Conditional repetition

```
i = 2
DO WHILE i < 3
    SAY "Aua!"; SAY "Oha!" i
    i = i + 1
END
```

Output:

Aua!

Oha! 2

Basics

Repetition, 7

- Conditional repetition

```
i = 3
DO WHILE i < 3
    SAY "Aua!"; SAY "Oha!" i
    i = i + 1
END
```

➔ No output, because block is not executed!

Basics

Repetition, 8

- Conditional repetition

```
i = 3
DO UNTIL i > 1
    SAY "Aua!"; SAY "Oha!" i
    i = i + 1
END
```

Output:

Aua!

Oha! 3

Basics

Execution, 1

```
/* */  
a = 3  
b = "4"  
say a b  
say a b  
say a || b  
say a + b
```

Output:

```
3 4  
3 4  
34  
7
```


Basics

Execution, 2

```
/* */  
"del *.*"
```

or:

```
/* */  
ADDRESS CMD "del *.*"
```

or:

```
/* */  
a = "del *.*"  
a
```

or:

```
/* */  
a = "del *.*"  
ADDRESS CMD a
```