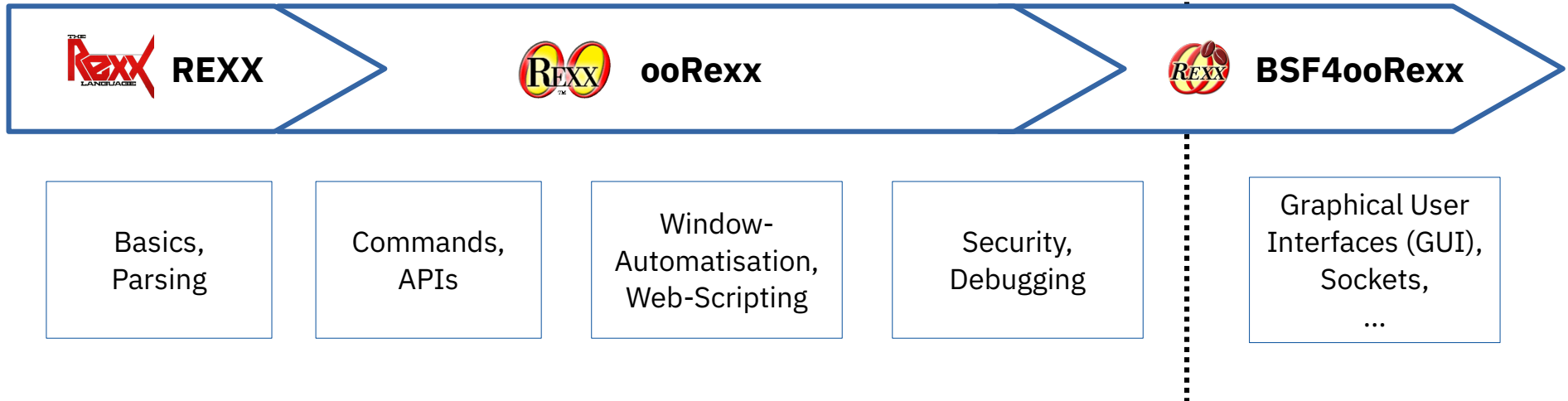


Procedural and Object-oriented Programming 1

Overview, Statements, Comparisons, Branches, Repetition

Business Programming 1

Business Programming 2



Why REXX and ooRexx?



- Human-centered language (simple syntax)
 - easy syntax and therefore *quick* to learn
- Powerful object-model
 - All important concepts of the object-oriented paradigm available
- Scripting language
 - Automation ("remote controlling") of applications and operating systems like Linux (D-Bus) or Windows (Windows Scripting Engine)
- BSF4ooRexx (Java-Bridge)
 - All of Java immediately available camouflaged as ooRexx
- Easy entry into other programming languages (e.g.: Java, Python)

- Course materials
 - Slides: http://wi.wu.ac.at/rgf/mcc/BP2025_26
- ooRexx 5 documentation
 - <https://sourceforge.net/projects/oorexx/files/oorexx-docs/5.2.0beta/rexxref.pdf>
- Related seminar, diploma, bachelor and master theses
 - <https://wi.wu.ac.at/rgf/diplomarbeiten/>
- Book
 - Flatscher R.G.: "Introduction to Rexx and ooRexx – From Rexx to Open Object Rexx (ooRexx)", facultas Management Book Service

Getting ooRexx (as of 2025-09-05)



- Rexx Language Association (non-profit SIG): <https://www.RexxLA.org>
- Latest ooRexx (beta versions are usually stable, fully productive)
 - Installation packages (needs administration rights)
 - <https://sourceforge.net/projects/oorexx/files/oorexx/5.2.0beta/>
 - **Considerations:**
 - **Operating System:** Linux, MacOS or Windows
 - Unblock after downloading with "powershell unblock-file *" (Windows)
 - Unblock by removing extended attributes (xattr) on MacOS
 - **Bitness:** 32bit (Linux, Windows) or 64bit (Linux, MacOS, Windows)
 - **Architecture:** x86, ARM (also Apple M1...), ...

History, 1



- 1979 – Mike F. Cowlshaw (IBM-Fellow)
 - **REXX**: Acronym for "**RE**structured **eX**tended **eX**ecutor"
 - Human-centric successor of “EXEC” language on IBM mainframes
 - Interactive (interpreter)
 - Keywords are English, resulting code looks like pseudo-code!
 - No reserved keywords unlike many other programming languages!
- 1987 – IBM’s System Application Architecture (SAA)
 - Procedural script language for all IBM platforms
 - Commercial and open source versions available for all operating systems
- 1996 – ANSI/INCITS "Programming Language – REXX" (INCITS 274:1996[S2008])

- Since 1990s – Development of an object-oriented REXX
 - Fully compatible with classic ("procedural") REXX but still with a simple syntax
 - Internally fully object-oriented (classic REXX statements are transformed)
 - Powerful object model (e.g. meta-classes, multiple inheritance)
- 1996 – Mike F. Cowlshaw development of NetREXX
 - NetRexx-programs are translated into Java byte code
 - Simpler programming of the Java VM (~30% less Code)
 - IBM handed over source code to RexxLA (<http://www.RexxLA.org/>)
 - June, 8th, 2011 opensource released by RexxLA

History, 3



- May 2004 – 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 2004 – Object REXX → "Open Object REXX" (ooRexx)
- April 2005 – RexxLA releases the opensource version of ooRexx
- ooRexx 5.1.0 released on 2025-05-04
 - BSF4ooRexx850 a bridge between ooRexx and Java is available
 - Work on ooRexx 5.2.0 has started

- The Hello World program is a tradition that dates back to 1974.

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

Output:

```
Hello, my beloved world
```

Notation of Program Text

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

- 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 (**,**), minus (**-**), plus (**+**), dot (**.**), Slash (**/**), parentheses (**()**), square brackets (**[]**), asterisk (*****), tilde (**~**), semicolon (**;**), colon (**:**) and underscore (**_**)

- 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.

- 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 a few possibilities available, e.g.
 - a) The constant value is assigned to a variable, the value of which never gets changed in the entire program (after all, it is a constant!)

```
Pi = 3.14159
```

- b) In ooRexx use the `::constant name value` directive

Comments

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

```
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
```

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

```
/* Some comment */  
SAY "Hello, my dear world";
```

Output:

```
Hello, my dear world
```

Statements, 2

- Statements may span multiple lines, but you need to indicate this with the continuation character
 - A dash (-) or comma (,) as the very last character on the line

```
/* Some comment */  
SAY "Hello," -  
    "my beloved world";
```

Output:

```
Hello, my beloved world
```

- 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
```

Comparisons (test_expression), 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
----------	----------------

Comparisons (**test_expression**), 2

- Boolean values can be combined

&	" and " (true : if both arguments are true)
 	" or " (true : if either argument is 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)

Comparisons (test_expression), 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)

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;`
- 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;`

```
IF age < 19 THEN SAY "Young."
```

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

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

- 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
```

Repetition, 1

- A block can be executed repeatedly

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

Output:

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

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!
```

Repetition, 3

- Repetition using a loop 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
```

Repetition, 4

- Repetition using a loop 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
```

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
```

Repetition, 6

- Conditional repetition (evaluated at the beginning of the block)

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

Output:

```
Aua!
Oh! 2
```

Repetition, 7

- Conditional repetition (evaluated at the beginning of the block)

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

Output:



→ No output, because block is not executed!

Repetition, 8

- Exit condition (evaluated at the end of the block)

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

Output:

```
Aua!
Oh! 3
```

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

Output:

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

Execution (Commands), 2

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

or:

```
/* */  
ADDRESS SYSTEM "del *.*"
```

or:

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

or:

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