Department of Information Systems and Operations Management

Procedural and Object-oriented Programming 3

Exceptions, References, Directives (::routine, ::requires)

Business Programming 1 | **Business Programming 2**

Basics, Parsing	Commands, APIs	Window-Automatisation, Web-Scripting	Security, Debugging	Graphical User Interfaces (GUI), Sockets, ...

Prof. Rony G. Flatscher

Execution of Programs

- Phase 1 (loading): file containing the program ("package") gets loaded
  - ***All lines*** are read
  - ***All statements*** are syntactically checked and compiled
- Phase 2 (setup): **::REQUIRES** directives are carried out
  - Remaining directives (e.g. **::ROUTINE**, **::CLASS**, ...) are carried out
- Phase 3 (execution): Program starts with the very first statement
  - This section is called "main program" or "prolog"
  - At this point in time access to all defined routines and classes of the program ("package") and access to all public routines and public classes of required or called programs is available and can be used

Cf. rexxref.pdf (3. Directives)

Conditions (Exceptions), 1

- Conditions
  - **SYNTAX** Statement not syntactically correct
  - **FAILURE** Error in external program
  - **ERROR** Error in external program, not trapped with "FAILURE" or "ANY"
  - **HALT** Ctl-C (Ctl-Break): user aborts program
  - **NOVALUE** Do not allow non-initialised variables
  - **USER usercondition** User-defined condition named "usercondition"
  - **LOSTDIGITS** Needs more digits than **NUMERIC DIGITS**
  - **NOMETHOD**, **NOSTRING**, **NOTREADY** (later ...)
  - **ANY** Intercepts (traps) all conditions

Cf. rexxref.pdf (11. Conditions and Condition Traps)

Conditions (Exceptions), 2

- Activating/deactivating the condition handling statements with
  - **CALL {on|off} condition [NAME label]**
    calls an internal routine that handles the condition from which one can return
  - **SIGNAL {on|off} condition [NAME label]**
    jumps to the internal routine that handles the condition
- Intercepting ("trapping", "catching") conditions can be activated with the keyword **on**, and deactivated with the keyword **off**
- **condition** is the name of one the aforementioned condition
  - The condition **USER** needs to be followed by its **usercondition** word
- **NAME** optional, allows for defining a **label** which serves as the **CALL** or **SIGNAL** target
  - If omitted, then the interpreter looks for a label which has the same name as the condition
    - In the case of a **USER** condition the interpreter looks for a label named **usercondition**

Cf. rexxref.pdf (11. Conditions and Condition Traps)

Conditions (Exceptions), 3

- Activate **SYNTAX** condition handling ("**SIGNAL ON SYNTAX**"), jump to label "**ANY:**"

```
/* */
SIGNAL ON SYNTAX NAME ANY /* target name "ANY" given   */
SAY Nix                  /* Variable not initialized! */
EXIT 0
ANY:  /* target for any exception     */
  exc_rc   = RC   /* save return code */
  exc_sigl = SIGL /* save line number */
  exc_type = CONDITION("C") /* get exception type */
  CALL say2stderr "REXX 'RC':" exc_rc
  CALL say2stderr "     type:" exc_type
  CALL say2stderr
  CALL say2stderr "  in line:" exc_sigl
  CALL say2stderr "          " SOURCELINE(exc_sigl)
  EXIT -1             /* indicate error   */
SAY2STDERR:           /* write to STDERR: */
  CALL LINEOUT "STDERR", ARG(1)
  RETURN
```

Output:

```
NIX
```


Cf. rexxref.pdf (11. Conditions and Condition Traps)

Conditions (Exceptions), 4

```
/* */
SIGNAL ON NOVALUE NAME ANY
SAY Nix                    /* Variable not initialized! */
EXIT 0
ANY:  /* target for any exception     */
  exc_rc   = RC   /* save return code */
  exc_sigl = SIGL /* save line number */
  exc_type = CONDITION("C") /* get exception type */
  CALL say2stderr "REXX 'RC':" exc_rc
  CALL say2stderr "     type:" exc_type
  CALL say2stderr
  CALL say2stderr "  in line:" exc_sigl
  CALL say2stderr "          " SOURCELINE(exc_sigl)
  EXIT -1             /* indicate error   */
SAY2STDERR:           /* write to STDERR: */
  CALL LINEOUT "STDERR", ARG(1)
  RETURN
```

Output:

```
C:\temp>rexx test.rex
REXX 'RC': RC
     type: NOVALUE

  in line: 3
           SAY Nix                    /* Variable not initialized! */
```


Cf. rexxref.pdf (11. Conditions and Condition Traps)

Conditions (Exceptions), 5

```
/* */
SIGNAL ON NOVALUE NAME ANY
SAY Nix                    /* Variable not initialized! */
EXIT 0
ANY:  /* target for any exception     */
  exc_rc   = RC   /* save return code */
  exc_sigl = SIGL /* save line number */
  exc_type = CONDITION("C") /* get exception type */
  CALL say2stderr "REXX 'RC':" exc_rc
  CALL say2stderr "     type:" exc_type
  CALL say2stderr
  CALL say2stderr "  in line:" exc_sigl
  CALL say2stderr "          " SOURCELINE(exc_sigl)
  EXIT -1             /* indicate error   */
SAY2STDERR:           /* write to STDERR: */
  CALL LINEOUT "STDERR", ARG(1)
  RETURN
```

Running program with redirecting error output to a file:

```
C:\temp>rexx test.rex 2>myerrors.txt
```

myerrors.txt (contains stderr output):

```
REXX 'RC': RC
     type: NOVALUE

  in line: 3
           SAY Nix                    /* Variable not initialized! */
```


Cf. rexxref.pdf (11. Conditions and Condition Traps)

Excursus: Operating System "Process"

- For each program that needs to be executed Unix or Windows
  - Creates a management unit named "*process*" and
    - Reserves memory
    - Reserves processor time
    - Sets up the following "standard files"
      - "standard input file ('*stdin*', file descriptor number 0)": default input via the keyboard
      - "standard output file ('*stdout*', file descriptor number 1)": default output to the screen (window)
      - "standard error file ('*stderr*', file descriptor number 2): default output to the screen (window)
    - Sets up the process "environment"
      - Defines environment variables like
        - *PATH* ... determines the directories and the order for seeking programs
        - *JAVA_HOME* ... determines the Java directory to use
        - *CLASSPATH* ... determines the directories and the order for seeking Java classes
  - Manages and supervises the program execution (using single or multiple threads)

Excursus: "Redirecting Standard Files", 1

- Redirecting the standard files when running a program in a process
  - Redirection operators: '**<**' (*stdin*), '**>**' (*stdout*, *stderr*) and '**>>**' (*stdout*, *stderr*)
    - '**>**' will delete the output file, whereas '**>>**' will append output to the output file
  - Examples

```
rexx myprogram.rex  0<myinput.txt
rexx myprogram.rex   <myinput.txt
```

    - Redirect input from the keyboard to the file "*myinput.txt*"

```
rexx myprogram.rex  1>myoutput.txt
rexx myprogram.rex   >myoutput.txt
```

    - Redirect output from the screen to the file "*myoutput.txt*" (will delete file if it exists)

```
rexx myprogram.rex  2>myerrors.txt
```

    - Redirect error output from the screen to the file "*myerrors.txt*" (will delete file if it exists)

Cf. rexxref.pdf (14.1.4. Default Stream Names)

Excursus: "Redirecting Standard Files", 2

- Redirecting standard files when running a program in a process

```
rexx myprogram.rex 0<myinput.txt 1>myoutput.txt 2>myerrors.txt
rexx myprogram.rex  <myinput.txt  >myoutput.txt 2>myerrors.txt
```

  - Input from file "*myinput.txt*" instead of keyboard, output to "*myoutput.txt*" instead of screen and error output to "*myerrors.txt*" instead of screen

- Redirecting *stderr* to *stdout*
  - Redirect output to a file and also redirect error output to the same file

```
rexx myprogram.rex >myoutput.txt 2>&1
```

    - Output goes to "*myoutput.txt*" instead of screen
    - Error output goes to where *stdout* goes to, i.e. "*myoutput.txt*" as well
    - Important note: always redirect *stdout* first and then redirect *stderr* to *stdout*!
  - Redirecting and *appending* to *stdout* and to *stderr*

```
rexx myprogram.rex >>myoutput.txt 2>>myerrors.txt
```

Excursus: "Redirecting Standard Files", 3

- "Piping"
  - Redirect *stdout* of one program to *stdin* of the next program
    - The output of one program becomes the input of another program
  - Piping operator: vertical bar ("**|**")
  - Example

```
rexx myprogram.rex | rexx myfilter.rex
```

    - The output (*stdout*) of "*myprogram.rex*" becomes the input (*stdin*) of "*myfilter.rex*"
  - One can redirect and pipe to/from multiple programs

```
rexx myprogram.rex < myinput.txt | rexx myfilter.rex 2>removed.txt | sort
```

    - "*myprogram.rex*": input is taken from the text file "*myinput.txt*", its output (*stdout*) gets piped to the next program's ("*myfilter.rex*") *stdin*
    - "*myfilter.rex*": input is taken from *stdout* of the previous program, *stderr* gets redirected to "removed.*txt*", *stdout* gets piped to next program's ("*sort*") *stdin*
    - "*sort*": input is taken from *stdout* of the previous program, output (*stdout*) goes to the screen

Excursus: "Redirecting Standard Files", 4

- "Null device"
  - Any output redirected to the "null device" gets discarded!
    - Unix null device, a file named

      /dev/null
    - Windows null device, a pseudo file named

      nul
  - Used in redirections, examples

    `rexx myprogram.rex >nul` (Windows)

    `rexx myprogram.rex >/dev/null` (Unix)
    - Show errors only (discard *stdout* output)

    `rexx myprogram.rex 2>nul` (Windows)

    `rexx myprogram.rex 2>/dev/null` (Unix)
    - Show output only (discard *stderr* output, i.e. do not show error messages)

"Redirection", Example, 1

```
/* myprogram.rex: write input lines (stdin) to output (stdout) */
do until value=""     -- if empty input, leave
   parse pull value   -- read line from stdin, assign to variable "value"
   say value          -- write to stdout
end
```

Command (Unix, Windows):

```
rexx myprogram.rex < myinput.txt
```

myinput.txt:

```
Max
und
Moritz
konnt
der
Lehrer ...
```

Output:

```
Max
und
Moritz
konnt
der
Lehrer ...
```

"Redirection", Example, 2

```
/* myprogram.rex: write input lines (stdin) to output (stdout) */
do until value=""    -- if empty input, leave
   parse pull value  -- read line from stdin, assign to variable "value"
   say value         -- write to stdout
end
```

```
/* myfilter.rex: if value contains 'r' then write it to stdout, else to stderr */
do until value=""
   parse pull value           -- read line from stdin, assign to variable "value"
   if pos('r',value)>0 then say value -- write value to stdout
                       else call lineout 'stderr',value   -- write value to stderr
end
```

Command (Windows):

```
rexx myprogram.rex < myinput.txt | rexx myfilter.rex >nul
```

Command (Unix):

```
rexx myprogram.rex < myinput.txt | rexx myfilter.rex >/dev/null
```

Output:

```
Max
und
konnt
```

myinput.txt:

```
Max
und
Moritz
konnt
der
Lehrer ...
```

"Redirection", Example, 3

```
/* myprogram.rex: write input lines (stdin) to output (stdout) */
do until value=""    -- if empty input, leave
   parse pull value  -- read line from stdin, assign to variable "value"
   say value         -- write to stdout
end
```

```
/* myfilter.rex: if value contains 'r' then write it to stdout, else to stderr */
do until value=""
   parse pull value           -- read line from stdin, assign to variable "value"
   if pos('r',value)>0 then say value -- write value to stdout
                       else call lineout 'stderr',value   -- write value to stderr
end
```

Command (Windows):

```
rexx myprogram.rex < myinput.txt | rexx myfilter.rex 2>nul
```

Command (Unix):

```
rexx myprogram.rex < myinput.txt | rexx myfilter.rex 2>/dev/null
```

Output:

```
Moritz
der
Lehrer ...
```

myinput.txt:

```
Max
und
Moritz
konnt
der
Lehrer ...
```

"Redirection", Example, 4

```
/* myprogram.rex: write input lines (stdin) to output (stdout) */
do until value=""    -- if empty input, leave
   parse pull value  -- read line from stdin, assign to variable "value"
   say value         -- write to stdout
end
```

```
/* myfilter.rex: if value contains 'r' then write it to stdout, else to stderr */
do until value=""
   parse pull value           -- read line from stdin, assign to variable "value"
   if pos('r',value)>0 then say value -- write value to stdout
                       else call lineout 'stderr',value   -- write value to stderr
end
```

Command (Windows):

```
rexx myprogram.rex < myinput.txt | rexx myfilter.rex 2>nul | sort
```

Command (Unix):

```
rexx myprogram.rex < myinput.txt | rexx myfilter.rex 2>/dev/null | sort -f
```

myinput.txt:

```
Max
und
Moritz
konnt
der
Lehrer ...
```

Output:

```
der
Lehrer ...
Moritz
```

Conditions (Exceptions), 5 (Repeated)

```
/* */
SIGNAL ON NOVALUE NAME ANY
SAY Nix                    /* Variable not initialized! */
EXIT 0
ANY:  /* target for any exception     */
  exc_rc   = RC   /* save return code */
  exc_sigl = SIGL /* save line number */
  exc_type = CONDITION("C") /* get exception type */
  CALL say2stderr "REXX 'RC':" exc_rc
  CALL say2stderr "     type:" exc_type
  CALL say2stderr
  CALL say2stderr "  in line:" exc_sigl
  CALL say2stderr "          " SOURCELINE(exc_sigl)
  EXIT -1             /* indicate error   */
SAY2STDERR:           /* write to STDERR: */
  CALL LINEOUT "STDERR", ARG(1)
  RETURN
```

Output:

```
C:\temp>rexx test.rex 2>myerrors.txt
```

myerrors.txt (contains stderr output):

```
REXX 'RC': RC
     type: NOVALUE

  in line: 3
           SAY Nix                    /* Variable not initialized! */
```


Cf. rexxref.pdf (11. Conditions and Condition Traps)

Raising Conditions (Exceptions), 1

- Usually, the Rexx-Interpreter raises conditions ...

  ... but you can also do it! :)

- **RAISE** statement
  - **RAISE** condition ... creates ("raises") the given condition
- **RAISE PROPAGATE**
  - Can only be given **during** condition (exception) handling
  - Re-creates the same exception in the caller, which allows the caller to also intercept it!

```
/* demoRaiseSyntax.rex */
SAY "hallo"
RAISE SYNTAX 9.1 /* Pretend syntax error # 9.1 */
EXIT 0
```

Output:

```
hallo
      3 *-* RAISE SYNTAX 9.1 /* Pretend syntax error # 9.1 */
Error 9 running C:\temp\demoRaiseSyntax.rex line 3:
    Unexpected WHEN or OTHERWISE
Error 9.1: WHEN has no corresponding SELECT
```


Cf. rexxref.pdf (2.23. RAISE)

Raising Conditions (Exceptions), 2

```
/**/
SIGNAL ON SYNTAX /* no label, hence "SYNTAX"   */
SAY "hallo"
RAISE SYNTAX 9.1 /* Pretend syntax error # 9.1 */
EXIT 0

SYNTAX:          /* target for any exception   */
  SAY "In SYNTAX-exception handling code."
  EXIT -1
```

Output:

```
hallo
In SYNTAX-exception handling code.
```


Cf. rexxref.pdf (2.23. RAISE)

Raising Conditions (Exceptions), 3

```
/**/
SIGNAL ON ANY    /* no label, hence "ANY"      */
SAY "hallo"
RAISE SYNTAX 9.1 /* Pretend syntax error # 9.1 */
EXIT 0
ANY:  /* target for any exception     */
  exc_rc   = RC   /* save return code */
  exc_sigl = SIGL /* save line number */
  exc_type = CONDITION("C") /* get exception type */
  CALL say2stderr "REXX 'RC':" exc_rc
  CALL say2stderr "     type:" exc_type
  CALL say2stderr
  CALL say2stderr "  in line:" exc_sigl
  CALL say2stderr "          " SOURCELINE(exc_sigl)
  EXIT -1             /* indicate error   */
SAY2STDERR:           /* write to STDERR: */
  CALL LINEOUT "STDERR", ARG(1)
  RETURN
```

Output:

```
hallo
REXX 'RC': 9
     type: SYNTAX

  in line: 4
           RAISE SYNTAX 9.1 /* Pretend syntax error # 9.1 */
```


Cf. rexxref.pdf (2.23. RAISE)

Routine Directive (ooRexx), 1

- Routine directives
  - Start with a double-colon (**::**)
  - Routine directives represent procedures and functions (= returning a value)
    - There is no **EXPOSE** statement available to the routine directive!
  - A routine directive is visible from everywhere in the program that defines it
  - If a routine directive has the subkeyword **PUBLIC** then it is made available to all Rexx programs that require or call the program that defines it
- Routine directives are regarded as if they were programs of their own!
  - Therefore labels, i.e. internal routines, are available **within** routine directives

Cf. rexxref.pdf (3.9. ::ROUTINE)

Routine Directive (ooRexx), 2

```
/**/
SAY  pp("hello")
CALL oha          /* routine is called */
SAY  pp("hello")

EXIT 0
pp : RETURN "<<<" || ARG(1) || ">>>"

:: ROUTINE oha PUBLIC
  SAY  pp("holla")
  EXIT 0
  pp : RETURN "[" || ARG(1) || "]"
```

Output:

```
<<<hello>>>
[holla]
<<<hello>>>
```


Cf. rexxref.pdf (3.9. ::ROUTINE)

Routine Directive and Exceptions, 1

- Routine directives are treated as if they were proper programs

```
/**/
SIGNAL ON USER TOO_SMALL /* intercept a user exception */
CALL checkAge 10
CALL checkAge 3
CALL checkAge 7
EXIT 0
TOO_SMALL:               /* dealing with the user exception */
  SAY "// caught exception 'TOO_SMALL' \\"
  EXIT -1
::ROUTINE checkAge
  PARSE ARG age
  SAY "--> age:" age
  IF age < 6 THEN RAISE USER too_small
             ELSE SAY "--> checked o.k."
  EXIT 0
```

Output:

```
--> age: 10
--> checked o.k.
--> age: 3
// caught exception 'TOO_SMALL' \\
```

Routine Directive and Exceptions, 2

```
/**/
CALL ON USER TOO_SMALL   /* intercept a user exception */
CALL checkAge 10
CALL checkAge 3
CALL checkAge 7
EXIT 0
TOO_SMALL:               /* dealing with the user exception */
  SAY "// caught exception 'TOO_SMALL' \\"
  RETURN
::ROUTINE checkAge
  PARSE ARG age
  SAY "--> age:" age
  IF age < 6 THEN RAISE USER too_small
             ELSE SAY "--> checked o.k."
  EXIT 0
```

Output:

```
 --> age: 10
 --> checked o.k.
 --> age: 3
 // caught exception 'TOO_SMALL' \\
 --> age: 7
 --> checked o.k.
```

Routine Directive and Exceptions, 3

```
CALL ON ANY           /* intercept anything that is not caught explicitly */
CALL ON USER TOO_SMALL /* intercept a user exception */
CALL ON USER too_big   /* intercept a user exception */
CALL checkAge 10
CALL checkAge 3
CALL checkAge 7        /* <- this is line # 6        */
EXIT 0

ANY      : SAY "in line:" SIGL "exception:" CONDITION("C"); RETURN
Too_small: SAY "// caught exception 'TOO_SMALL' \\";        RETURN
TOO_BIG  : SAY "// caught exception 'TOO_BIG' \\";          RETURN
::ROUTINE checkAge
  PARSE ARG age
  SAY '--> age:' age
  IF age < 6 THEN RAISE USER too_small
             ELSE IF age > 9 THEN RAISE USER too_big
                             ELSE SAY '--> checked o.k.'

  RAISE USER something_raised
  EXIT 0
```

Output:

```
 --> age: 10
 // caught exception 'TOO_BIG' \\
 --> age: 3
 // caught exception 'TOO_SMALL' \\
 --> age: 7
 --> checked o.k.
 in line: 6 exception: USER SOMETHING_RAISED
```

Requires Directive (ooRexx)

- **::Requires** "rexxprogram.rex"
  - Denotes a Rexx program that is required (e.g. "rexxprogram.rex")
    - Hint: for portable purposes, enclose the filename in quotes (Unix is case sensitive)
  - The interpreter will call the required program before carrying out any of the other directives (**::Routine**, **::Class**, **::Method**, ...)
  - Thereafter all of its public routines (and public classes!) are made available
    - If another program requires the same required Rexx program later, then the interpreter will immediately make its (now already known) public routines and public classes available, without calling it a second time

Cf. rexxref.pdf (3.7. ::REQUIRES)

CALL-Statement and Public Routines

```
/* cmd1.rex */
SAY "In" "cmd1.rex"
CALL cmd2
SAY "In" pp("cmd1.rex")
```

```
/* cmd3.rex */
SAY "    \1\ In" pp("cmd3.rex")
CALL cmd4
SAY "    \2\ In" pp("cmd3.rex")
EXIT 0

::ROUTINE pp
  RETURN "c3<<" || ARG(1) || ">>c3"
```

Output:

```
In cmd1.rex
  /1/ In c2[cmd2.rex]c2
    \1\ In c3<<cmd3.rex>>c3
          In c4<cmd4.rex>c4
    \2\ In c3<<cmd3.rex>>c3
  /2/ In c2[cmd2.rex]c2
In c4<<cmd1.rex>>c4
```

```
/* cmd2.rex */
SAY "  /1/ In" pp("cmd2.rex")
CALL cmd3
SAY "  /2/ In" pp("cmd2.rex")
EXIT 0

pp :
  RETURN "c2[" || ARG(1) || "]c2"
```

```
/* cmd4.rex */
SAY "          In" pp("cmd4.rex")
EXIT 0

pp :
  RETURN "c4<" || ARG(1) || ">c4"

::ROUTINE pp PUBLIC
  RETURN "c4<<" || ARG(1) || ">>c4"
```

Requires-Directive and Public Routines

```
/* cmd1.rex */
SAY "In" pp("cmd1.rex")

::REQUIRES cmd2.rex
```

```
/* cmd3.rex */
SAY "    \1\ In" pp("cmd3.rex")
EXIT

:: requires cmd4.rex

::ROUTINE pp
  RETURN "c3<<" || ARG(1) || ">>c3"
```

Output:

```
          In c4<cmd4.rex>c4
    \1\ In c3<<cmd3.rex>>c3
  /1/ In c2[cmd2.rex]c2
In c4<<cmd1.rex>>c4
```

```
/* cmd2.rex */
SAY "  /1/ In" pp("cmd2.rex")
EXIT 0

pp :
  RETURN "c2[" || ARG(1) || "]c2"

::Requires cmd3.rex
```

```
/* cmd4.rex */
SAY "          In" pp("cmd4.rex")
EXIT

pp :
  RETURN "c4<" || ARG(1) || ">c4"

::ROUTINE pp PUBLIC
  RETURN "c4<<" || ARG(1) || ">>c4"
```

Variables and Values

- Rexx
  - Variables hold references to string values ("strings") only
    - If a variable is not set, its uppercase name is used as its string value
  - Arguments for routines (procedures, functions)
    - Only string values (strings) allowed in classic Rexx
- ooRexx
  - Variables hold references to any values (objects) including string values
  - Arguments for routines, methods
    - **USE ARG** statement
      - Introduced with ooRexx
      - Fetches arguments by reference (!)
    - **PARSE ARG** statement
      - Fetches arguments by their string value only, if necessary requests a string value from objects