Procedural and Object-oriented Programming 6

Commands

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

Commands, 1

- Directed to the operating system
- Typed on the command line, e.g.
  - "dir" (Windows), "ls" (Unix) ... list entries
  - "copy" (Windows), "cp" (Unix) ... copy files
  - "sort" (Windows, Unix) ... sort files
- Commands may have switches and arguments
  - "dir /s" (Windows), "ls -R" (Unix)
    - ... list also entries in subdirectories
  - "dir a* /s" (Windows), "ls -R a*" (Unix)
    - ... list entries starting with "a"
  - "sort /R" (Windows) "sort -r" (Unix)
    - ... sort in reverse order

Cf. rexxref.pdf (1.16.2. Commands)

Commands, 2

- Commands always have a numeric **r**eturn **c**ode (RC)
  - Upon return of a command Rexx sets a Rexx variable named RC to the command's return code
    - It is therefore very easy to learn about return codes in Rexx
  - If the return code is 0, then the command executed without errors
    - Any other value for the return code and its meaning is usually documented with the command
- Attention! Commands with file and path names may contain blanks!
  - The command line interpreter splits commands at blanks, unless double quotes get used!
  - Therefore always enquote file/path names with double quotes!
    - Windows example: dir /s ***"c:\abc def\this is a file with blanks.txt"***
    - Unix example: ls -al ***"/abc def/this is a file with blanks.txt"***

Cf. rexxref.pdf (1.16.2. Commands)

Commands, 3

- Rexx can run commands directly from strings, e.g.

```
"dir /s"                    -- execute command
```

  - Rexx will submit the string as the command to the system for execution
  - Upon return the Rexx RC variable refers to the command's return code

- Rexx can run commands referred to by variables, e.g.

```
command="dir /s"            -- assign to variable command
command                     -- execute command
```

- Use the ***ADDRESS*** keyword statement explicitly, e.g.

```
ADDRESS SYSTEM "dir /s" -- execute command
-- or
command="dir /s"            -- assign to variable command
ADDRESS SYSTEM command  -- execute command
```


Cf. rexxref.pdf (1.16.2. Commands)

Commands, 4

- One can assign the command to a Rexx variable and use that

```
 -- get current operating system
parse source op_sys +1  -- "W"...Windows, everything else is regarded to be Unix
if op_sys="W" then command="dir /s" -- list directory and subdirectories
              else command="ls -R"  -- list directory and subdirectories
command           -- execute command via the operating system (command line)
say "Rexx, RC="rc
```

Output (Windows), maybe:

```
Directory of e:\tmp
10.11.2020  13:38    <DIR>          .
10.11.2020  13:38    <DIR>          ..
10.11.2020  13:40    <DIR>          .idea
10.11.2020  13:16               546 example1.rex
               2 File(s)          1 287 bytes

Directory of e:\tmp\.idea
…cut…
Rexx, RC=0
```


Cf. rexxref.pdf (1.16.2. Commands)

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 a single or multiple threads)

"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)

"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
```

"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

"Redirecting Standard Files", 4

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

Excursus

"Redirection", Example, 1

```
/* myprogram.rex: write input lines (stdin) to output (stdout) */
do until value=""     -- if empty input, leave
   parse pull value   -- get value from stdin
   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  -- get value from stdin
   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           -- get value from stdin
   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  -- get value from stdin
   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           -- get value from stdin
   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, 4a (PARSE PULL)

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

"Redirection", Example, 4b (PARSE LINEIN)

```
/* myprogram2.rex: write input lines (stdin) to output (stdout) */
signal on notready       -- raised, if no input data anymore
loop                     -- loop eternally, same as: do forever
  parse linein value     -- get line from stdin, raises notready if no data anymore
  say value              -- write to stdout
end
notready:                -- label to signal, if no input anymore
```

```
/* myfilter2.rex: if value contains 'r' then write it to stdout, else to stderr */
signal on notready       -- raised, if no input data anymore
do forever               -- loop eternally, same as: loop
   parse linein value    -- get line from stdin, raises notready if no data anymore
   if pos('r',value)>0 then .output~say(value)  -- write value to stdout
                       else .error ~say(value)  -- write value to stderr
end
notready:                -- label to signal, if no input anymore
```

Command (Windows):

```
rexx myprogram2.rex < myinput.txt | rexx myfilter2.rex 2>nul | sort
```

Command (Unix):

```
rexx myprogram2.rex < myinput.txt | rexx myfilter2.rex 2>/dev/null | sort -f
```

myinput.txt:

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

Output:

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

Commands, 5

- Commands may include all desired redirections and pipes!
- Possible to redirect the command's standard files to and from ooRexx!
- Use the ***ADDRESS ... WITH*** statement for redirections
  - INPUT option: command's *stdin* provided by an ooRexx collection object
  - OUTPUT option: command's *stdout* gets appended to an ooRexx collection object
  - ERROR option: command's *stderr* gets appended to an ooRexx collection object
- Example
  - Use two ooRexx arrays to receive *stdout* and *stderr* output from command

```
outArr=.array~new
errArr=.array~new
ADDRESS SYSTEM "some command" WITH OUTPUT USING (outArr) ERROR USING (errArr)
```


Cf. rexxref.pdf (1.16.2. Commands)

Commands, 6

- Redirecting command's *stdout* to an ooRexx array

```
-- get current operating system
parse source op_sys +1  -- "W"...Windows, everything else is regarded to be Unix
if op_sys="W" then command="dir /s" -- list directory and subdirectories
              else command="ls -R"  -- list directory and subdirectories

say "Rexx, command:" quote(command)
say "Rexx, current directory:" quote(directory())

   -- redirect standard output to the 'arrOut' array
arrOut=.array~new -- create Rexx array for fetching standard output
   -- execute command, redirect output to Rexx array
ADDRESS SYSTEM command WITH OUTPUT USING (arrOut)
say "Rexx, RC="rc
say "Rexx, arrOut~items:" arrOut~items "(lines)"

::routine quote   -- enquote argument in double quotes
  return '"' || arg(1) || '"'
```

Output (Windows), maybe:

```
Rexx, command: "dir /s"
Rexx, current directory: "e:\tmp"
Rexx, RC=0
Rexx, arrOut~items: 26 (lines)
```

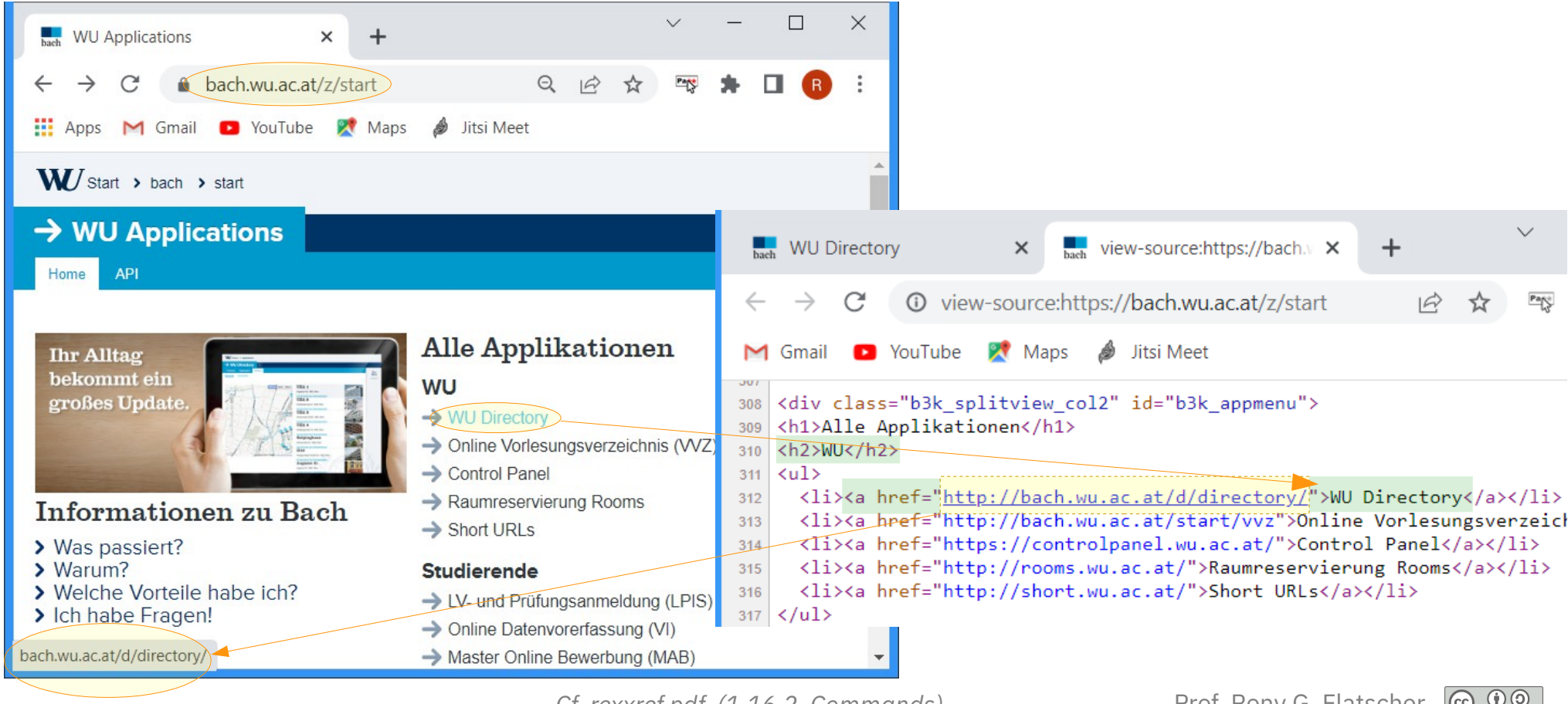

Cf. rexxref.pdf (1.16.2. Commands)

"curl" Command

- "curl"
  - Command "Client URL (cURL)" to allow accessing web resources
    - A Utility originating from Unix available for Windows 10, Linux and MacOS as well
    - Very powerful, very popular, quite easy to use
    - Uses standard files
      - *stdin* e.g. for data to upload
      - *stdout* for the fetched data from the server
      - *stderr* for information about the operation and errors, if any
    - Links for more information on "curl"
      - Wikipedia: <https://en.wikipedia.org/wiki/CURL>
      - Tutorials:
        - <https://curl.se/docs/httpscripting.html>
        - <https://curl.se/docs/manual.html>
        - <https://www.baeldung.com/curl-rest>

Cf. rexxref.pdf (1.16.2. Commands)

"curl" Example 1, Extracting Link from HTML Text, 1



Cf. rexxref.pdf (1.16.2. Commands)

"curl" Example 1, Extracting Link from HTML Text, 2

- Extracting WU's Directory URL (as of 2022-05-24)
  - Get WU BACH's home page from the URL: https://bach.wu.ac.at/z/start
  - Parse the received data for the WU directory URL and display it

```
command="curl https://bach.wu.ac.at/z/start"
outArr=.array~new           -- array for stdout
ADDRESS SYSTEM command WITH OUTPUT USING (outArr)
source=outArr~makeString    -- turn array into string
parse var source '<h2>WU</h2>' . '<a href="' url '">WU Directory<'
say "Current URL of BACH-WU Directory:" url
```

Output:

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 15583  100 15583    0     0   423k      0 --:--:-- --:--:-- --:--:--  434k
Current URL of BACH-WU Directory: http://bach.wu.ac.at/d/directory/
```

from curl's *stderr*

Cf. rexxref.pdf (1.16.2. Commands)

"curl" Example 1, Extracting Link from HTML Text, 3

- Extracting WU's Directory URL (as of 2022-04-03)
  - Version that intercepts *stderr* output from *curl* to inhibit it to be displayed

```
command="curl https://bach.wu.ac.at/z/start"
outArr=.array~new           -- array for stdout
errArr=.array~new           -- array for stderr: curl's information goes here
ADDRESS SYSTEM command WITH OUTPUT USING (outArr) ERROR USING (errArr)
source=outArr~makeString    -- turn array into string
parse var source '<h2>WU</h2>' . '<a href="' url '">WU Directory<'
say "Current URL of BACH-WU Directory:" url
```

Output:

```
Current URL of BACH-WU Directory: http://bach.wu.ac.at/d/directory/
```


Cf. rexxref.pdf (1.16.2. Commands)

"curl" Example 2 – Using a Weather API (wttr.in)

- Most websites have APIs (**A**pplication **P**rogramming **I**nterface**s**), which allow to specify the response cURL gets using **URL query strings**.

```
curl https://wttr.in/Vienna?format="%l:+%t+%w"
```

- Websites describe what responses can be delivered:
  - <https://github.com/chubin/wttr.in#one-line-output>

```
command='curl https://wttr.in/Vienna?format="%l:+%t+%w"'
outArr=.array~new           -- array for stdout
errArr=.array~new           -- array for stderr: curl's information goes here
ADDRESS SYSTEM command WITH OUTPUT USING (outArr) ERROR USING (errArr)
SAY outArr
```

Output:

```
Vienna: +15°C →33km/h
```

%l:	+%t	+%w

Cf. rexxref.pdf (1.16.2. Commands)

"curl" Example 3 – Using a Translation API (Deepl.com)

- Many websites require an authorization key, which is often free and available under certain conditions.
  - Get the Deepl.com authorization key (2022-04-19): <https://www.deepl.com/pro#developer>
  - A list of websites for inspiration (2022-04-19): <https://github.com/public-apis/public-apis>

```
url = "https://api-free.deepl.com/v2/translate"
key = "yourAuthorizationKey"      -- you need to obtain a valid key from deepl.com!
original_text = "ooRexx die Sprache der unbegrenzten Möglichkeiten."
translate_to = "EN"
command="curl" url "-d auth_key="key "-d text="quote(original_text) "-d target_lang="translate_to
outArr=.array~new           -- array for stdout
ADDRESS SYSTEM command WITH OUTPUT USING (outArr) ERROR USING (.array~new)
parse var outArr . ':"' translated_from '","text":"' translated_text '"' -- parse turns array into string
say "Translated by Deepl.com from" translated_from "to" translate_to
say "Translated text:" translated_text

::routine quote -- encloses text in quotes, such that it may contain blanks
  return '"' || arg(1) || '"'
```

Output (if authorization key is correct):

```
Translated by Deepl.com from DE to EN
Translated text: ooRexx the language of unlimited possibilities.
```


Cf. rexxref.pdf (1.16.2. Commands)