Technische Hochschule Rosenheim

Modul - IT Systems (IT)

Bachelor Programme AAI

06 - Lecture: Shell Programming

Prof. Dr. Marcel Tilly

Faculty of Computer Science, Cloud Computing

Vers. 21.11.2021

Agenda

Technische Hochschule Rosenheim

We will look at the following in this lecture:

1. introduction shells
2. file operations
3. scripting
4. parameters, variables, ...
5. user interaction
6. control structures
7. java VM recap

Learning Objectives

Technische Hochschule Rosenheim

Students ...

- ... can create shell scripts
- ... can automate simple tasks via shells
- ... can interact with the OS on the command line
- ... know different terminals

Technische Hochschule Rosenheim

Classification according to operating modes

- **Batch processing (batch processing)**: Processing of a sequence of batch jobs. A batch job is compiled by the user with all the necessary programs, data and job control language (JCL) instructions. It is processed completely without user interaction.
  - Examples: *IBM OS/370, OS/390, MVS, BS 2000*.
- **Dialog mode (interactive processing)**: Constant change between actions of the user (e.g. command inputs) and those of the system (e.g. command execution). The user can influence the workflow in the dialog at any time.
  - Examples: *MS-DOS, MS Windows 95/98/ME/NT/2000/XP, UNIX/Linux*.

Process concept

Technische Hochschule Rosenheim

Definition

> *A **Program (Program)** is a static sequence of instructions in a programming language using data. It is used to code an algorithm and is generally in the form of a file.*

Definition

> *A (sequential) **process (also: task)** is a dynamic sequence of actions (changes in state) that comes about by executing a program on a processor. A process is characterized in particular by its time-varying state. It is generated in the operating system as a result of a task.*

Comandline and Shells

Technische Hochschule Rosenheim

In computing, a shell is a command-line interpreter which exposes access to an operating system's services.

Unix Shells

- Bourne shell (/bin/sh)
- bash (Bourne-again-shell)
- ash (Almquist shell)
- dash (Debian Almquist shell)
- Korn shell
- csh
- tcsh
- ...

Windows

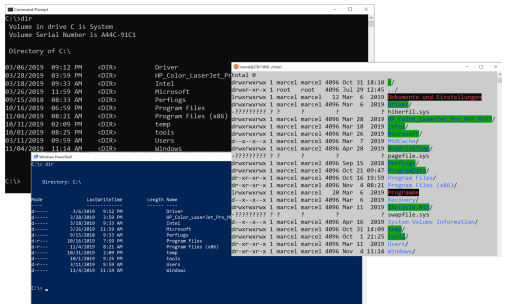
- Comandline (cmd)
- Powershell
- Cygwin
- Git bash
- Linux subsystem (WSL2, bash, ..)

Start Shell

- A shell is basically executed in a terminal window
  - Linux: Terminal
  - Windows: CMD or Powershell
  - Mac: Terminal

On Windows:

```
WIN-R
CMD or powershell
```



Linux Shell under Windows

Technische Hochschule Rosenheim

There is a possibility to run Linux shells under Windows as well.

Popular variants:

- Cygwin
- MingW
- Linux Subsystem under Windows 10 (WSL, WSL2)
- Git BASH

> Single commands (programs) can be executed on the shell!

Help for shell commands

Technische Hochschule Rosenheim

Standard procedure if you don't know what the command does anymore:

- `--help` or `--?` for quick help
- man pages (= manual pages) with detailed description
- stackoverflow if nothing else helps!

```
$ man dir

DIR(1) User Commands DIR(1)

NAME
       dir - list directory contents

SYNOPSIS
       dir [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current directory by default).  Sort entries alphabetically.

       Mandatory arguments to long options are mandatory for short options too.
...
```

File commands

Technische Hochschule Rosenheim

activity	Linux/ Mac	Windows
create empty file	touch	dir
list directory contents	ls	dir
Copy (copy)	cp	copy
Move	mv	move
Rename	mv	ren
Delete (delete)	rm	del
file output	cat, less	type, echo
change directory	cd	
create directory	mkdir	mkdir
print working directory	pwd	echo %cd%
output argument	echo	
find file	find	
find file with index	locate	

Windows commands: https://www.thomas-krenn.com/de/wiki/Cmd-Befehle_unter_Windows / Linux commands: https://docs.cs.cf.ac.uk/notes/linux-shell-commands/

Examples

Technische Hochschule Rosenheim

```
$ mkdir test
$ cd test

~/test $ touch test.txt$
~/test $ cp test.txt test2.txt

~/test $ ls
test2.txt test.txt

~/test $ mv test2.txt test.md
~/test $ ls
test.md test.txt

~/test $ rm *.txt
~/test $ ls
test.md
```

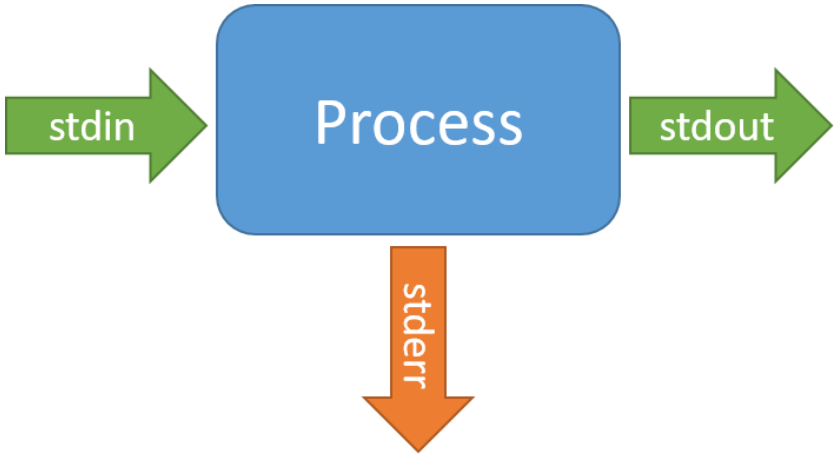
Input and Output

Technische Hochschule Rosenheim

- 0 stdin standard input (default: keyboard)
- 1 stdout standard output (default: terminal)
- 2 stderr standard error (default: terminal)

Redirection

I/O	Pipe	alt
stdin	<	
stdout	> (1>)	
(append)	1>	>&1
stderr	2>	
stderr+	2>>	>&2



Examples

Technische Hochschule Rosenheim

```
$ murks
murks: command not found
$ murks 2> err
$ cat err
murks: command not found
```

Example: `out.sh`

```
#!/bin/bash
echo "Output to stdout." >&1
echo "Output to stderr." >&2
```

What is happening here?

Useful commands

Technische Hochschule Rosenheim

Activity	Linux	Windows
counts lines, words, characters	wc	find /c
page by page output	more, less	more
programmable filter	grep	???
stream editor	sed	???
show process information	ps	tasklist
process information as tree	pstree	???

Examples

Technische Hochschule Rosenheim

- Word count

```
$ wc script.md ✔ 2239
874 2926 20940 script.md
```

- Show process information

```
$ ps 2241
  PID TTY TIME CMD
16940 pts/5 00:00:00 ps
24390 pts/5 00:00:01 zsh
```

> Try: `pstree` and `top`

Shell scripting

Technische Hochschule Rosenheim

- The shell is to communicate with the user
  - in addition, it knows most of the constructs of a programming language.
- Instructions can be stored in a text file, which can then be called like any other UNIX command.
- Such files are called *shell-script* (or *batch-script* under Windows).
- A shell script can be called in two ways:
  - via a sub-shell: `sh <filename>`.
  - via the name `./do.sh` (if Excecute permission is set - chmod u+x filename)
- Of course, other shell scripts can also be called within a script.
- *Parameters* can be passed to the shell script, so it can be used more universally.

Testing shell scripts

Technische Hochschule Rosenheim

For testing there are some possibilities:

- Insert echo commands instead of the intended commands. One can control in such a way the replacements of the Shell with the call.
- Call via subshell with options:
  - `-v` (verbose): The shell prints all processed commands.
  - `-x` (execute): The shell prints the substitutions.
  - `-n` (execute): The shell issues the commands, but they are not executed.
  - Typical command call: `sh -vx do.sh`.
- Only *critical* commands (e.g. rm) should be *disarmed* by preceding them with echo commands.
- A colon before a line acts like a comment (command will not be executed), but parameter substitution works.

Comments in shell scripts

Technische Hochschule Rosenheim

- Comments are introduced by the '#' character.
- Everything after the '#' in a line is considered a comment.
- Space and tab characters can normally be used in any number.

In many scripts you will find a special form of the comment line at the beginning, which looks like this for example:

```
#!/bin/sh
```

Sha-bang

Technische Hochschule Rosenheim

- The sha-bang (#!) at the head of a script tells your system that this file is a set of commands to be fed to the command interpreter indicated.
- The #! is actually a two-byte magic number, a special marker that designates a file type, or in this case an executable shell script (type man magic for more details on this fascinating topic).
- Immediately following the sha-bang is a path name. This is the path to the program that interprets the commands in the script, whether it be a shell, a programming language, or a utility.

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/python3
#!/bin/sed -f
```

Comments in shell scripts

Technische Hochschule Rosenheim

```
#!/bin/sh
```

- This comment specifies which program is used to run the script.
- It is mainly used for scripts that should be available to all users.
- Since different shells may be used, syntax errors may occur when executing the script depending on the shell (sh, csh, ksh,...).
- By specifying the executing shell it is ensured that only the "correct" shell is used.
- Besides shells, other programs can be used to execute the script; python and perl scripts are common.

Shell variable $

Technische Hochschule Rosenheim

- Variables are freely selectable identifiers (names) that can hold arbitrary strings.
- If the strings consist only of digits, they are interpreted as integer numbers.
- When dealing with variables, three basic forms can be distinguished:
  - Variable declaration
  - Value assignment
  - Value referencing
- If the value, i.e. the content of a variable is meant, a `$` sign is placed in front of the name.
- A variable is valid within the shell.
- Shell scripts may in turn call shell scripts or programs. Thereby the inheritance of a variable must be explicitly defined (-> export by the `export` command).

Shell variable

Technische Hochschule Rosenheim

Examples:

```
# declaration & assignment
$ foo=bar

# referencing
$ echo $foo
bar

$ foo=$(pwd)
$ echo "What comes out here? $foo"
```

What is output here? - Try it out!

Predefined variables

Technische Hochschule Rosenheim

At system startup and when calling the files

/etc/profile (system presets) and

.profile (user-defined presets), some variables are already defined.

All currently defined variables can be listed by the command `set`.

```
'!'=0
'#'=0
0=/usr/bin/zsh
'?'=0
COLORTERM=truecolor
COLUMNS=244
CPUTYPE=x86_64
CURRENT_FG=black
DEFAULT_COLOR=black
...
```

Predefined variables

Technische Hochschule Rosenheim

Some predefined variables are beside others:

Variable	Meaning
HOME	home directory (absolute path)
PATH	Search path for commands and scripts
MANPATH	search path for manual pages
MAIL	mail directory
SHELL	name of the shell
LOGNAME	Login name of the user
PS1	System prompt ($ or #)
PS2	prompt for requesting further input (>)
IFS	(internal field separator, usually CR, space and tab)

Special variables

Technische Hochschule Rosenheim

The following special variables are defined:

Variable	Meaning	Example
$-	set shell options	set -xv
$$	PID (process number) of the shell	kill -9 $$
$!	PID of the last background process	kill -9 $!
$?	exit status of the last command	cat /etc/passwd ; echo $?

Examples

```
$ echo $$
```

Parameter access

Technische Hochschule Rosenheim

- Shell scripts can be called by passing *parameters* which can be accessed by their position number.
- The parameters can also be assigned with predefined values.

parameter command	meaning
$#	Number of arguments
$0	name of the command
1...9	1st - 9th argument
$@	all arguments
$*	all arguments concatenated

Parameter access

Technische Hochschule Rosenheim

Example (foo.sh):

```
#!/bin/sh
echo "My name is $0"
echo "I have been passed $# parameters"
echo "1st parameter = $1"
echo "2nd parameter = $2"
echo "3rd parameter = $3"
echo "All parameters together: $*"
echo "My process number PID = $$"
```

What comes out when calling:

```
$ ./foo.sh one two three four
```

Interactive input

Technische Hochschule Rosenheim

Shell scripts with interactive input can also be written by using the `read` command.

```
read variable [variable ...]
```

- `read` reads a line from standard input and assigns the individual fields to the specified variables.
- If more variables than input fields are defined, the excess fields are filled with empty strings.
- Conversely, the last variable takes the rest of the line.
- If input is read from a file in the shell script with `<`, `read` processes the file line by line.

Example

```
$ read -p "Input: " VALUE
$ echo $VALUE
```

Interactive input

Technische Hochschule Rosenheim

> Note: Since the shell script runs in a sub-shell, IFS can be redefined in the script without having to restore it afterwards.

For example, the script `show.sh` contains the following commands:

```
#!/bin/sh
IFS=','
echo "Please enter three parameters, separated by commas:"
read A B C
echo Input was: $A $B $C
```

Call:

```
$ ./show.sh
Please enter three parameters, separated by comma:
one,two,three
Input was: one two three
```

Linking commands

Technische Hochschule Rosenheim

- The commands are written one after the other separated by semicolon **;**:

```
command1; command2; command3
```

- **AND &&**: The second command is executed only if the first one was successful.

```
command1 && command2
```

- **XOR ||**: The second command is executed only if the first was unsuccessful.

```
command1 || command2
```

Test conditions

Technische Hochschule Rosenheim

An important command is `test`:

```
test argument
```

- This command tests a condition and returns
  - *true* (0): if the condition is true
  - *false* (1): if the condition is not true
  - The error value 2 is returned if the argument is syntactically incorrect (usually caused by substitution).
- Files, strings and integer numbers (16 bit, 32 bit for Linux) can be checked.
- The argument of Test consists of a test option and an operand, which can be a file name or a shell variable (string or number).

test - command

Technische Hochschule Rosenheim

Examples

```
test -w ./err
```

with the command concatenation you can already make logical decisions, e.g.:

```
test -w ./err && echo "Error!"
```

Normally, instead of `test` the argument can also be put in square brackets.

> **Watch out for spaces after and before the brackets**:

```
[ -w ./err ]
```

`test` operations

Technische Hochschule Rosenheim

The following operations can be used with **test** or **[...]**.

expression	meaning
-e < file >	file exists
-r	file exists and read permission
-w	file exists and write permission
-x	file exists and execute right
-f	file exists and is simple file
-d	file exists and is directory
-h	file exists and is symbolic link

Logical operations

Technische Hochschule Rosenheim

- Comparison of strings

expression	meaning
-n	true if string is not empty
-z	true if string is empty
=	true if string is equal
!=	true if strings are different

Example

```
$ test -z "" && echo "empty"
empty
$ [ -z "" ] && echo "empty"
empty
```

Logical operations

Technische Hochschule Rosenheim

- Algebraic comparison of integers

operator	meaning
-eq	equal - equal
-ne	not equal - unequal
-ge	greater than or equal - greater than equal
-gt	greater than - greater
-le	less than or equal - less than equal
-lt	less than - less

Example

```
$ test 13 -eq 13 && echo "Equal"
Equal
```

Logical operations

Technische Hochschule Rosenheim

- Logical operation of two arguments

operator	meaning
AND	-a
OR	-o
brackets	( )
negation	!

Example

```
$ test 13 -eq 13 -a 2 -eq 2 && echo "Equal"
Equal
```

if ... then

Technische Hochschule Rosenheim

- Not only the *test* command, but any sequence of commands can be used as a condition.
- Each command returns an *error code*, which is equal to
  - on successful execution equal to *null* (**true**)
  - in case of an error or abort *is not equal to zero* (**false**).
- The `if` statement is used to test a condition.
- Each statement must either be on a separate line or separated from the other statements by a semicolon.

```
if commandlist
then
      commands
else
      commands
fi
```

if ... then

Technische Hochschule Rosenheim

Examples:

A message should be output if more than 5 users are logged in:

```
if [ $(who | wc -l) -gt 5 ] ; then
  echo "More than 5 users on the device".
fi
```

What does this query do?

```
if test $# -eq 0
then
  echo "usage: sort filename" >&2
else
  sort +1 -2 $1 | lp
fi
```

`case` statement

Technische Hochschule Rosenheim

- This statement allows multiple selections.
- It is also popular because it allows patterns with wildcards and multiple patterns for a selection

```
case selector in
     pattern-1) command sequence 1 ;;
     pattern-2) command sequence 2 ;;
              ....

     pattern-n) command sequence n ;;
esac
```

- The variable `selector` (string) is compared in sequence with the patterns "pattern-1" to "pattern-n".
- If they are equal, the following command sequence is executed.

`case` statement

Technische Hochschule Rosenheim

- Meta characters (*, ?, []) are allowed in the patterns, but not in the selector.
- The pattern * coincides with each selector (default output); must be the last pattern in the case construction.
- Several patterns, separated by |, can be placed in front of the parenthesis.
- The character | forms an or-condition:

```
case selector in
        pattern1) command sequence1 ;;
        pattern2 | pattern3) command sequence2 ;;
        *) command sequence3 ;;
esac
```

`case` example

Technische Hochschule Rosenheim

```
while : # infinite loop (see later)
do
tput clear # clear screen and output menu text
   echo " +---------------------------------+"
   echo " | 0 → end |"
   echo " | 1 → date and time |"
   echo " | 2 → current directory |"
   echo " | 3 → table of contents |"
   echo "+---------------------------------+"
   echo "input: \c" # no line feed
   read ANTW
   case $ANTW in
    0) kill -9 0 ;; # and tschuess
    1) date ;;
    2) pwd ;;
    3) ls -CF ;;
    *) echo "Wrong input!" ;;
   esac
done
```

`for` statement

Technische Hochschule Rosenheim

This loop statement ìterates through a list of elements to be processed.

for loop with list:

```
for selector in list
  do
  Command sequence
done
```

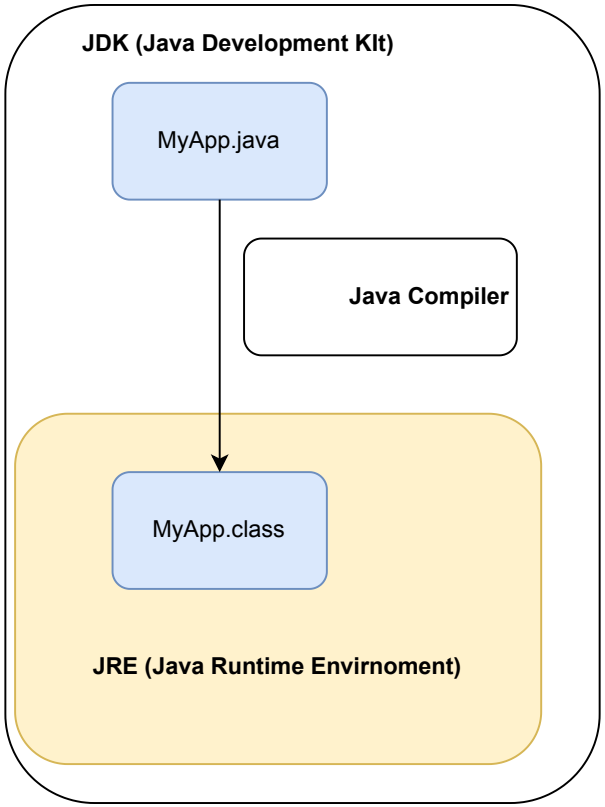
Examples:

```
for VAR in 1 2 3 4
  do
  echo $VAR
done
```

Java compiler

- Java source code must be translated into bytecode by the compiler.
- For this purpose the Java source code `myClass.java` is transferred into a class file `MyClass.class`.
- The Java compiler is called `javac` and is located where?

```
$ javac myClass.java
```



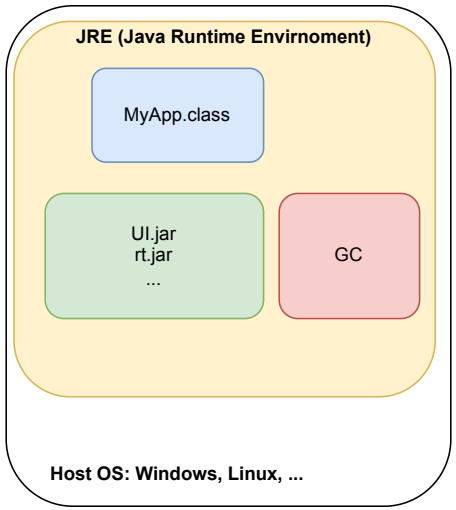
Java VM

Technische Hochschule Rosenheim

Java programs (class files) are executed as bytecode in a Java Virtual Machine (JVM) - the JVM is available for different operating systems.

- The JVM makes the Java code portable and executable on different systems.
- The JVM is started by the `java` program.
- The class file (without extension) is passed to the `java` program at startup.
- At startup the method `main(String[] args)` is searched and executed.

```
$ java myClass
```



Java Classpath

Technische Hochschule Rosenheim

- The *Classpath* is the path where the JVM searches for classes or resources.
- The *Classpath* can be set by the parameter both for the Java compiler and for the JVM.
- Default classpath is `.` (current directory)!
- The parameter is either `-classpath` or `-cp`.

```
C:> java -classpath .;/./classes/main/java;./libs/junit.jar;./libs/log4j.zip
```

-or-

```
$ javac -cp .:/./classes/main/java:./libs/junit.jar:./libs/log4j.zip
```

-or-

```
C:> set CLASSPATH=classpath1;classpath2...
```

Summary

Technische Hochschule Rosenheim

This is what you should know after the lecture:

- File management commands
- Basic shell scripting
  - parameters
  - For statement
  - IF...THEN
  - echo and read
  - test
- What does `java`, `javac`

and especially

- **What is the Java classpath?**.