CE350
Lecture5

PROGRAMMING THE BASH SHELL
PART I

by Kaya Oğuz and İlker Korkmaz

Programming the bash

- As pointed in syllabus document, there will be 5 lecture weeks to cover the programming with bash scripts.
- This lecture is the first part on scripting in bash.

References

- The contents of this lecture are prepared with the help of and based on:
  - the textbook, UNIX Shells by Example
  - the tutorial, Advanced Bash-Scripting Guide at http://tldp.org/LDP/abs/html/

Contents of Lecture5

- The interactive bash shell's :
  - Variables
  - Quoting
  - Arrays
  - Standard I/O redirection
  - Pipes
  - Built-in commands

A note:

- From now on, for all related LAB works of the lectures on scripting in bash shell, you shall make script files to put your commands and script codes in.

Variables

- A variable is a tag/label/identifier/name in the program that refers to a corresponding memory location.
- Some variables are defined by the user and some others are special shell variables.
- There are two types of variables in bash:
  - Local variables: known only to the shell in which they were created.
  - Environment variables: available to the new spawned (forked) processes as well.

Naming conventions

- Variable naming conventions, or identifier rules, are similar as used in programming languages.
- The simplest format for defining a local variable is to use initialization.
    - *variableName=value*
    - decimalVariable=18
- If the name, or identifier, of a variable is **variableName**, the reference to its value is **$variableName**
- **$variableName** is actually a simplified form of **${variableName}**
- To set a variable to “null” (means no assigned value and does not mean “zero”), the equal sign is to be followed with a newline
    - variableName=

Variable substitution

- To illustrate some script examples, which introduces the variables, you may dissect Example 4-1 at
  - http://tldp.org/LDP/abs/html/varsubn.html

The “declare” built-in

- “declare” is also used to declare a variable.
  - *declare variableName=value*
  - *declare decimalVariable=18*
- “declare” is a built-in command.
- Table 13.13 of the textbook explains the argument options of the “declare” command.
  - -r option makes variables read-only (can not be unset, but can be reassigned).
  - -i option makes variables integer types.
  - -x option exports variable names to subshells.

Local variables

- Local variables of parent shell can not be accessed from the child shell.
- Try to understand the “scope” concept.
- an example:
  - $ round=world # a local variable in the shell
  - $ echo $round # prints the value as “world”
  - $ bash # a new shell starts
  - $ echo $round # **nothing to be printed !!**
  - $ exit # exits and returns to parent
  - $ echo $round # prints “world”

Environment variables

- Table 13.14 lists the bash environment variables
- Environment variables of parent shell can be inherited from the child shell.
  - use “export” or “declare” with -x option
    - ***export variableName=value***
    - OR
    - ***variableName=value; export variableName***
    - OR
    - ***declare -x variableName=value***

An example

- $ NAMES=“Kaya and Ilker”
- $ export NAMES
- $ echo $NAMES # prints the value
- $ bash # a new subshell starts
- $ echo $NAMES # prints the value
- $ declare -x NAMES=“Ilker and Kaya”
- $ echo $NAMES # **What gets printed?**
- $ exit # returns to parent shell
- $ echo $NAMES # **What gets printed?**

Printing the values of the variables

- To print the values of the variables, use
  - The built-in **echo** command.
  - OR
  - The **printf** command, as same with its use in C programming language, can be used to format the printed output.

Bash variables are untyped

- You shall quickly read the subject at
  - http://tldp.org/LDP/abs/html/untyped.html

TO DO AT HOME

- The two sections of the textbook, “variable expansion modifiers” and “variable expansion of substrings” should be read and Table 13.18 and Table 13.19 should be examined at home.
- *A reminder: TO DO parts will not be graded.*

Quoting

- Quoting is used to protect special metacharacters from interpretation.
- Three kinds of quoting:
  - the backslash, \
  - single quotes, ' '
  - double quotes, “ “

The backslash

- \ is used to quote (or escape) a single character from interpretation.
    - $ echo CE350\? # prints CE350?
- \ protects the dollar sign ($), backquotes (` `), and the backslash (\) from interpretation if enclosed in double quotes.
    - $ echo “\$25.00” # prints $25.00
- \ is not interpreted if placed in single quotes
    - $ echo \\ # prints \
    - $ echo '\\' # prints \\
    - $ echo '\$25.00' # prints \$25.00

The single quotes

- Single quotes (' ') protect all metacharacters from interpretation.
    - $ echo hello CE350 # prints -> hello CE350
    - $ echo 'bye “CE350”' # prints -> bye “CE350”
- To print a single quote, enclose it within double quotes or escape it with a backslash.
    - $ echo isn\'t it CE350'?' # prints -> isn't it CE350?

The double quotes

- “ ” will allow variable and command substitution, and protect any other special metacharacters from being interpreted.
    - $ lectureCode=CE350
    - $ echo “Hello $lectureCode ! Time is $(date)”
        - # **What gets printed ??**

Arrays

- Newer versions of bash provides the shell with one dimensional arrays, which allow the script programmers collect a list of words into a variable name.
- To introduce the array in a script:
    - declare -a arrayName=(item1 item2 ...)
    - OR
    - myArray[4]=100 # no maximum size limit
      # here, the size of myArray is 1
- The order may be changed at initialization:
    - array=(ilker [2]=kaya [1]=korkmaz [3]=oguz)
- to extract an element of an array:
    - use ${arrayName[index]}

A little more about arrays

- $ declare -a myArray
- $ myArray=(first second third fourth)
- $ echo ${myArray[0]} # the first element in myArray
- $ echo ${#myArray[0]} # the length of the first element
- $ echo ${#myArray} # the length of the first element
- $ echo ${#myArray[*]} # the number of elements in myArray
- $ echo ${#myArray[@]} # the number of elements in myArray
- $ unset myArray # **OR** unset ${myArray[*]}

Standard I/O redirection

- The shell inherits 3 files at the starting phase:
  - stdin, stdout, stderr
- Through the use of I/O redirection
  - inputs can be read from a file
  - outputs or errors can be sent to a file
- Table 13.23 lists the redirection operators.
  - command < file # redirects input
  - command > file # redirects output
  - command >> file # redirects and appends output
  - command 2> file # redirects error
  - command 2>> file # redirects and appends error
  - command 1>&2 # redirects the output to where error is going
  - ...

Redirection using exec

- The exec command can be used to replace the current program with a new one without starting a new process.
- If you try “ exec date ” in the current shell, “date” executes in place of the current shell and the shell would not return since the user would be logged out.
- Redirection using exec is explained in Table 13.24 as follows:
    - $ exec < inputFile # standard input is to be inputFile
    - $ exec > outFile # standard output is to be outFile
    - $ exec 3< inFile # opens inFile as file descriptor 3 for reading input

Pipes

- A **pipe** takes the output from the command on the left-hand side of the pipe symbol (|) and sends it to the input of the command on the right-hand side of the symbol. A pipeline can consist of more than one pipe.
    - $ pwd | wc -l
    - the same as: $ pwd > tmp; wc -l tmp

Bash shell built-in commands

- The bash shell includes many built-in commands within its source code.
- The built-in commands are not located in disk, which makes the execution faster.
- Table 13.28 lists the shell built-in commands and their objectives.
- **help** command provides the user with the help for the features of any built-in command.