ESSENTIALS OF RED HAT LINUX

Session 11

Shell Scripting

Objectives

- Explain what is meant by the bash shell
- Explain how to change the shell
- List the various command line shortcuts
- Explain shell script
- Explain how to generate outputs using shell scripts
- Explain how to use the structured language constructs

bash **Shell**

A computer only understands binary language, composed of two binary numbers, 0 and 1.

- It is difficult for developers to read and write in binary language.

The concept of Shell was introduced, in which the operating systems contain a program that accepts the user's instructions or commands in a language very similar to English.

- If the commands entered are valid, these are passed on to the kernel.
- The shell interprets the commands given by the user and translates them into machine code so that the kernel can understand.

Introduction to the `bash` Shell [1-3]

- An intermediary program that interprets the user-typed commands at the terminal.
- Converts user typed commands to a form that the kernel understands.
- Eliminates the need for direct communication between the programmer and the kernel.
- Several shells can run on the Linux platform, but the default shell of Linux is `bash`.
- Interprets the commands and executes the requested program.

Linux is:

- A multitasking operating system: It is possible to execute more than one program at a time.
- A multi-user OS: It can have more than one shell running at the same time.

Introduction to the `bash` Shell [2-3]

- In a multi-user system, each user gets a copy of the shell after logging in.
- The `bash` is responsible for:

- Command line editing
- Job control
- Stream manipulation
- Wildcard expansion
- Aliases
- File completion
- Command history
- Variables
- Control structures
- Sub shells

Introduction to the `bash` Shell [3-3]

- Starting the shell:

When the shell starts depends on whether the user uses a graphical or text-mode login.

If you are logging on in text-mode, the shell is immediately started after entering the password.

If you are using a graphical login manager, log on to the Linux system, and the **Terminal** option from the **Accessories** menu.

`bash` **Shell Scripting**

- All shell statements in Linux can be entered at the command line.
- The shell can read the files and execute the commands within the files, called scripts.
- Shell scripts run mostly during the installation of the operating system.

The three tasks that need to be performed to convert an ordinary file to a script file are:

- Should be executable
- Should be in text format and contain executable statements
- Should have the `#!` comment appearing at the first line indicating the interpreter to be used

- To make a file executable, the `chmod` command is used which changes the access mode of a file.
- Only the file owner, or the root user, may change the access mode.

Identifying the Shell and Changing the Shell [1-3]

When the user logs into a Linux machine, a series of messages, followed by a `$` prompt appears.

- The `` `$' `` character preceding the cursor, tells you that the system is ready and waiting for input.

- `Bash` Shell: The `sh` command located in `/bin` directory.
- The shell for any user can be switched temporarily, or changed semi-permanently.
- Although `bash` is highly versatile, each shell has its own characteristics.
- The current default shell for a user can be determined using the `grep` command.
- Following code snippet can be used to find the default shell for a user named `student`:

```
[student@localhost ~]$ grep student /etc/passwd
```

Identifying the Shell and Changing the Shell [2-3]

- The `grep` returns a line such as, `student:513:513::/home/student:/bin/bash` – the shell used is `bash`.
- Following code snippet uses the `cat` command to read the ***/etc/shells*** file, lists all the shells installed on the system:

```
[student@localhost ~]$ cat /etc/shells
```

  - To switch to a different shell, type in the shell name at the command prompt, press `ENTER`.

- Following code snippet displays how to change from the current shell to `sh` shell:

```
[student@localhost ~]$ sh
```

  - Upon changing shells, a different command prompt is shown, depending on the new shell.

Identifying the Shell and Changing the Shell [3-3]

- To return to the original shell, type the name of the original shell at the command prompt followed by pressing the `ENTER` key.

- Following code snippet returns to the `bash` shell from the `sh` shell:

```
[student@localhost ~]# bash
```

- The method to permanently change the shell may vary according to the system.

The first step is to determine the absolute path, to the shell.

Once the absolute path has been determined, change the default shell using the `chsh` command.

The `chsh` command prompts to enter the password followed by the absolute path of the desired shell.

Command Line Shortcuts and Expansions [1-7]

- The `bash` shell provides many powerful command line shortcuts and tools.
- **File Globbing**: The wildcards, or metacharacters allow one pattern to be spread out to multiple filenames, is globbing.
  - Following code snippets demonstrates how to delete all the files that have the extension .mp3:

```
[student@localhost ~]$ rm *.mp3
```

```
[student@localhost ~]$ rm birdsing.mp3 song.mp3
```

- **Auto-completing on the command line with the `TAB` key**: Pressing the `TAB` key helps to auto-complete a command name or a file name on the command line.
  - For example, typing the command as shown in the following code snippet and pressing the `TAB` key completes the command.

```
[student@localhost ~]$ cd /u
```

Command Line Shortcuts and Expansions [2-7]

- **Storing history of commands with the `bash` history**: To view the last command typed in, the up arrow key needs to be pressed at the `bash` prompt.
  - To view all the stored commands, the history command can be used as shown in the following code snippet:

```
[student@localhost ~]$ history
```

  - The shortcuts for retrieving the commands from history are listed in the table:

Command Line Shortcut	Description
`!!`	Repeats the last command
`!c`	Repeats the last command that started with the letter 'c'
`!n`	Repeats a command by its number in the history output
`!?abc`	Repeats the last command that contains `abc`
`!-n`	Repeats a command entered `n` commands earlier

History Shortcuts

Command Line Shortcuts and Expansions [3-7]

- Following code snippet retrieves the eighth command that had been entered by the user:

```
[student@localhost ~]$ !8
```

- Following code snippet retrieves the last command that started with the letter V:

```
[student@localhost ~]$ !V
```

- **Referring to home directory with tilde (~)**: Used as a shortcut for referring to the current user's or another user's home directory.

  Syntax:

  ```
  [student@localhost ~]$ ~username
  ```

- **Explaining the variable and string declaration in** `bash` **shell**: Variables in the shell are defined by words on the command line, prefixed by a `$` sign.
  - The shell substitutes the string with the value of the variable before the command is executed.

Command Line Shortcuts and Expansions [4-7]

- The code in the following code snippet expands the variable `$HOME` to its appropriate value, and executes the `cd` command.

```
[student@localhost ~]$ cd /home/username/index.html
```

```
[student@localhost ~]$ cd $HOME/index.html
```

- Commas are used to separate the various string patterns; the following code snippet demonstrates the use of commas:

```
[student@localhost ~]$ echo {a,b}
a b
[student@localhost ~]$ echo x{a,b}
xa xb
```

- **Command substitution with back quotes**: Termed as command substitution, allows the output of a command to replace the command itself.

  Syntax:

```
[student@localhost ~]$ `(command)`
```

Command Line Shortcuts and Expansions [5-7]

- Following code snippet executes '`ls -l`' part and substitutes the output after the string "`The contents of this directory are`":

```
[student@localhost ~]$ echo "The contents of this directory are
" `ls -l` > dir.txt
```

- **Arithmetic expansion**: The arithmetic expression is evaluated and the result is substituted.

  Syntax:

```
[student@localhost ~]$(( expression ))
```

- Following code snippet uses the backslash before the asterisk to ensure that each element is a separate shell word:

```
[student@localhost ~]$ echo Area : 'expr $X \* $Y'
```

- Following code snippet is rewritten using the $[] syntax as shown here:

```
[student@localhost ~]$ echo Area: $[ $X * $Y ]
```

Command Line Shortcuts and Expansions [6-7]

The escape character backslash

- A non-quoted backslash '\' is the `bash` escape character.
- Preserves the literal value of the next character that follows, except for the `newline` character.
- If a `\newline` pair appears, and the backslash itself is not quoted, the `\newline` is treated as a line continuation.

- Single and Double quotes: Enclosing characters in single quotes (' ') preserves the literal value of each character within the quotes.

Command Line Shortcuts and Expansions [7-7]

- The steps in the process of expanding a command line are:

Step 1	The command line is split into the shell words, delimited by spaces, tabs, new lines, and some other characters.
Step 2	Functions are expanded.
Step 3	Curly brace ({}) statements are expanded.
Step 4	Tilde (~) statements are expanded.
Step 5	Parameters and variables are substituted.
Step 6	The lines are again split into shell words.
Step 7	File globs (*, ?, [abc]) are expanded.
Step 8	The command is executed.

Creating Shell Scripts [1-3]

- A script file is created by using the vi editor.
- Following code snippet creates and edits a text file by using the `vi` command:

```
[student@localhost ~]$ vi
```

  - Press `i` to insert text at the current cursor position.
- Finally, type the script as shown in the following code snippet:
-
```
#!/bin/bash
#My first script
echo "Hello World"
```

- After creating the script file, save the file with the name as ‘`my_script`’.
- This is done by pressing the `:wq` command followed by a space bar.

Creating Shell Scripts [2-3]

- Anything following the '#' symbol is considered as a comment and is ignored by the interpreter.
- Adding comments to the shell script makes the script readable and helps the person to understand what the script does.

The first line in a shell script should contain '`magic`', which is generally referred to as the `shebang`.

- This line informs the operating system about the interpreter being used to execute the shell script.

After creating and saving the shell script, its file permissions need to be changed to make the script executable.

- The `chmod` command is used to set the permissions on the file.

Creating Shell Scripts [3-3]

- The permission groups are represented as:

Owner	• Represented by 'u'
Group	• Represented by 'g'
World	• Represented by 'o'
All of the above	• Represented by 'a'

Syntax:

```
[student@localhost ~]$ chmod u+x scriptfilename
```

- The script can be executed by either:
  - Placing the script file in a directory in the executable path, or
  - By specifying the absolute or relative path to the script file on the command line.
- The '`u`' specifies the user who owns the file, and '`x`' specifies the permission to execute the file.

Generating Outputs with `echo` and `printf` Commands [1-3]

- To display some text or the value of a variable, the `echo` command is used.

  Syntax:

```
[student@localhost ~]$ echo [options] [string, variables...]
```

- The options available with the `echo` command are listed in the following table:

Option	Description
-n	Does not display the trailing new line
-e	Enables interpretation of the backslash escaped characters in the strings

Options of the `echo` Command

- Following code snippet shows the use of the `echo` command:

```
[student@localhost ~]$ echo "Welcome to Red Hat Enterprise
Linux"
[student@localhost ~]$ echo -n "Please enter the correct login
name"
```

Generating Outputs with `echo` and `printf` Commands [2-3]

To display the formatted output, the `printf` command is used.

- The `printf` command does not provide a newline, so multiple `printf` statements can be applied to one line for complex formatting.
- The newline character, `\n`, is used if a newline is desired.

Syntax:

```
[student@localhost ~]$ printf Format [ Argument]
```

- Converts, formats, and writes its `Argument` parameters to the standard output.
- The `Format` parameter can consist of literal characters, format control strings, and additional options.
- Format control strings are of the form `%width.precision` followed by a conversion character.

Generating Outputs with `echo` and `printf` Commands [3-3]

Following code snippet shows an example:

- The result is formatted as 05.2f, which means the result is a floating point number with a total width of 5 digits.

```
#!/bin/sh
$Result=6.789
printf "The result is %05.2f\n" $Result
```

- The output after executing code snippet is:

```
The result is 06.79
```

Reading Input with the `read` Command [1-2]

Reads one line of data from the standard input and separates it into individual words.

- Separated words are assigned to the variables sequentially.
- If the number of words exceeds the number of variables, the remaining words are assigned to the last variable.

Syntax:

```
[student@localhost ~]$ read [Options] [filename]
```

- The common options available with `read` command are listed in the table given below.

Reading Input with the `read` Command [2-2]

Option	Description
`-a aname`	Words are assigned to sequential indices of the array variable `aname`, starting at 0th position
`-d delim`	Character specified in `delim` is used to terminate the input line, rather than newline character
`-e`	If the input comes from a terminal, `readline` is used to obtain the entire line
`-n nchars`	Stops reading after `n` number of characters
`-s`	Silent mode. If input is coming from a terminal, characters are not echoed

Options of the `read` Command

- Following code snippet prompts the user to enter three values, and reads the values entered by the user:

```
#!/bin/bash
read "Enter three values:" a b c
echo "Value of a is $a"
echo "Value of b is $b"
echo "Value of c is $c"
```

bash Shell Variables [1-2]

- The shell offers the facility to define and use the variables in the command line, known as shell variables.
- Shell variables are assigned a value using the = operator.
  - To access the value of the variable, the $ character is used as a prefix before the variable name.
  - Following code snippet demonstrates the use of shell variables:

```
[student@localhost ~]$ x=40
[student@localhost ~]$ echo $x
```

- The general form of declaring a shell variable is: variable=value.

The variables are of string type.

- Any word preceded by a $ sign is a variable.
- The variable is then replaced by the value assigned to it.
- All the shell variables are initialized to null strings by default.

`bash` Shell Variables [2-2]

- To assign multi-word strings to a variable, enclose the value within single quote as shown in the following code snippet:

```
[student@localhost ~]$ x='Welcome to Linux'
[student@localhost ~]$ echo $x
```

- Variables created within a shell are local to that shell, are not accessible to other shells.
  - To be accessible to commands outside the shell, use the export command to export it into the environment.
- When arguments are specified with a shell procedure, they are assigned to positional parameters.
- Following code snippet displays the name of the program assigned to the argument `$0`, the first argument, assigned to the argument `$1`:

```
[student@localhost ~]$ echo "The program name is $0"
[student@localhost ~]$ echo "The first argument is $1"
```

`expr` Command [1-2]

This command performs two functions:

- Performs arithmetic operations on integers.
- Manipulate strings to a limited extent.

Can perform the four basic arithmetic operations, and the modulus function.

Can handle only integers.

- Some examples of the usage of the `expr` command, and their outputs are shown in the given code snippet.

`expr` Command [2-2]

```
[student@localhost ~]$ x=3
[student@localhost ~] y=5                  # Variable assignments;
[student@localhost ~]$ expr 3+5
Output :
8
[student@localhost ~]$ expr 3 \* 5# The Asterisk is escaped here
Output:
15
[student@localhost ~]$ expr $x - $y
```

```
Output:
-2
[student@localhost ~]$ expr $y / $x #        The decimal part
after the division operation is truncated
Output:
1
[student@localhost ~]$ expr 13 % 5
Output:
3
```

Structured Language Constructs

- Structured programming: An organized approach towards programming that uses a hierarchy of modules.
- Modularization: Grouping of statements together that have some relation to each other.
- Shell programming:
  - Follows the structured programming model.
  - Supports the following three types of control structures:

Sequential	Executes the code line by line till the end of the program
Selection	Executes the appropriate code based on a logical decision
Repetition	Repeatedly executes the code based on a logical decision

`if/else` Statements [1-4]

- `if` statement: Evaluates the expression to a boolean value of `true` or `false`.
  - If the expression evaluates to `true`, the commands after the `then` statement are executed.
  - If the expression evaluates to `false`, the commands between the `then` and `if` statements are skipped and the shell resumes with the processing of the statements lying outside the `if` block.

Syntax:

```
if [ condition ]; then
    command1
    command2
…….
If
```

- The given code snippet checks if keyboard input is being accepted.

if/else Statements [2-4]

```
if tty -s; then
            echo "Enter text and end with \^D"
if
```

- The if-then statement, evaluates the code following the then statement only if the condition is true.
- The limitation of the if-then statement is that it does not consider the situation if the expression evaluates to false.
- This problem can be solved using the if/else statement.

Syntax:

```
if condition
then
            do something
else
            do something else
if
```

if/else Statements [3-4]

- Following code snippet searches for a pattern 'director' in the file emp.lst:

```
#!/bin/sh
if grep "director" emp.lst
then
   echo "Pattern found"
else
echo "Pattern not found"
if
```

- Here, the grep command is executed first.
- The if/ elif/ else statement is used when one of the conditions can evaluate to true.
- In the given code snippet, the value stored in the variable, person, is matched against three constant values, Steve, Todd, and Markus.

if/else Statements [4-4]

```
if [ $person = Steve ]
   then
       print $person is on the sixth floor.
elif [ $person = Todd ]
   then
       print $person is on the fifth floor.
      elif [ $person = Markus ]
   then
       print $person is on the fourth floor.
Else
       print "Who are you talking about?"
fi
```

`case` Statement [1-2]

The `case` selection structure compares the value in the variable against every pattern until a match is found.

- When a match is found, the statements following the matching pattern are executed.
- If no match is found, the `case` statement exits without performing any action.
- It provides a default section that is used if none of the patterns match.

- Each `case` statement is terminated by `esac` statement.

Syntax:

```
case variable in
              pattern1 )
                       statements ;;
              pattern2 )
                       statements ;;
 .
 .
 esac
```

`case` Statement [2-2]

- The statements corresponding to the first pattern matching the expression are executed, after which the case statement terminates.
- The patterns can be plain strings or expressions using *, ?, !, [], and so on.
- In the following code snippet, the case statement matches the value of the variable `$person` for the strings Steve, Todd, and Markus:

```
case $person in
           steve)
           print “He’s on the sixth floor.” ;;
        todd)
           print “He’s on the fifth floor.” ;;
        markus)
           print “He’s on the fourth floor.” ;;
         *)
           print I do not know $person. ;;
        esac
```

`for-loop` **Repetition Statement [1-2]**

Can be used to iterate over all items in a list and execute commands on each of these items.

Is terminated by the done statement.

Syntax:

```
for variable in list-of-values
    do
          commands
    done
```

The `for` statement continues iterating over the list until the list-of-values is exhausted.

for-loop Repetition Statement [2-2]

- Following code snippet demonstrates the use of `for` statement.
- Consists of a list of series of `character` strings (`a, b, c, d,` and `e`) assigned to the variable '`alphabet`'.

```
$alphabet="a b c d e"                # Initialize a string
count=0                              # Initialize a counter
for letter in $alphabet
Do
       count=`expr $count + 1`   # Increment the counter
       echo "Letter $count is [$letter]" # Display the result
done
```

while-loop Repetition Statement [1-2]

- Uses conditions the same way the `if` statements do.
- Is executed as long as the condition remains true.
- The code to be executed is written within `do` and `done` statements.

  Syntax:

```
while condition
     do
             commands..


done
```

- The given code snippet just replaces the for loop set-up in the previous code snippet with its equivalent `while` syntax.
- The code uses the `bc` command, which represents a precision calculator.

while-loop Repetition Statement [2-2]

```
alphabet="a b c d e"                     # Initialize a string
count=0                                  # Initialize a counter
while [ $count -lt 5 ]                   # Set up a loop control
do   count=`expr $count + 1`              # Increment the counter
  position=`bc $count + $count - 1`      # Position of next letter
  letter=`echo "$alphabet" | cut -c$position-$position` # Get next
letter
  echo "Letter $count is [$letter]"        # Display the result
done
```

`continue`, `break`, and `exit` Statements [1-3]

- Used to interrupt the execution of the loop.

`Break` command	`Continue` command	`Exit` Command
• Stops the execution of the current iteration of the loop, and jumps out to the nearest enclosing loop.	• Stops the execution of the current iteration of the loop, and forces the loop to jump to its next iteration.	• In the middle of a loop, or somewhere in a script if it may become necessary to exit the execution, which can be achieved using the `exit` command.

- The given code snippet checks if the value of the variable index, is less than or equal to 3.
  - The `-le` operator denotes "`less than or equal to`" and `-ge` operator denotes "`greater than or equal to`".

continue, break, and exit Statements [2-3]

```
for index in 1 2 3 4 5 6 7 8 9 10
 do
 if [ $index -le 3 ]; # Checks if index is less than or equal to 3
 then
   echo "continue"
 continue # Moves to the next iteration of the for statement until
the index
         # is less than or equal to 3
 if
 echo $index
 if [ $index -ge 8 ]; # Checks if index is greater than or equal to
8
 then
   echo "break"
 break # Moves out of the for loop when the index is greater than
or equal to 8
 if
 done
```

continue, break, and exit Statements [3-3]

To specify the exit status, use the exit command followed by a non-zero number.

- If no exit status is provided, the exit command exits having the status value as zero, which indicates success.
- Following code snippet takes two positional arguments. It will exit with status 2 (error) rather than 0 (success) if it is not invoked with two parameters:

```
if [ $# -ne 2 ]
   # "$#" is number of parameters- here we test
   # whether it is not equal to two
   then
   echo "Usage $0 \<file1\> \<file2\>"        #not two parameters
   # so print message
   exit 2    # and fail ($0 is # name of command).
   if
```

Functions [1-4]

- Using functions while scripting helps make scripting easier and in easy maintenance of the code.
- Enable the program to be broken into smaller manageable entities.
  - Each of these entities perform a particular task and can return a value.
- A function always begin with a function name.
  - The commands to be executed are enclosed within the curly braces.
  - The code is reusable.
  - The shell functions must be declared in the shell scripts before being used.

Syntax:

```
function-name ( )
   {
              command1
              …
              commandN
              return
   }
```

Functions [2-4]

- The `function-name` is the name of your function, that executes a series of commands.
- A `return` statement is used to terminate a function.
- Following code snippet defines a function named ‘`SayHello`’:

```
SayHello()
{
        echo “Hello , Have a nice day”
}
```

- To execute the function, the function name is typed on the command line as shown here:

```
[student@localhost ~]$ SayHello
```

- Can receive arguments, to make more generic or versatile reusable codes.

Functions [3-4]

- Following code snippet demonstrates a function to check if the arguments are passed to it:

```
#!/bin/bash
func2 () {
    if [ -z $1 ]
    # Checks if any params.
    then
        echo “No parameters passed to function.”
        return 0
 else
 echo “Param #1 is $1.”
    f
   }
  func2
  # Called with no params
  func2 first
  # Called with one param
  func2 first second
  # Called with two params
  exit 0
```

Functions [4-4]

- `func2()` checks to see if the number of arguments passed to it is zero.

The `return` statement in the function can be used to set the value of the variable $1.

- When a value is returned by a function, it is accessible outside the function.
- Following code snippet demonstrates an example of a function that returns a value:

```
anymore() {
echo "\n$1 ? (y/n) : c" 1>&2
read ans
case "$ans" in
    y|Y) return 0 ;;
    *) return 1 ;;
esac
}
```

Summary

- The shell is an intermediary program that interprets the user-typed commands at the terminal.
- The user can store sequences of frequently used Linux commands in files, called scripts. The shell can read these files and execute the commands within these files.
- The `echo` and `printf` commands are used to display some text or the value of a variable. The `printf` command displays formatted text. The `read` command is used to read the input from the standard input.
- The `if/ elif/ else` structure is used when multiple comparison tests have to be performed.
- The case selection structure is used to select a pattern from amongst multiple patterns and execute the statements based on the selected pattern.
- A `for loop` can be used to iterate over all items in an array or list and execute the statements. The `while` loops use conditions the same way if statements do; they continue to run until the condition becomes `false`.
- The `break` and `continue` statements are used to interrupt the loop execution.