8

Using Advanced Shell Features in Shell Scripts

Objectives

After completing this lesson, you should be able to:

- Use advanced shell features
- Write shell scripts

Agenda

- Using advanced shell features
- Writing shell scripts

Jobs in the `bash` Shell

- A job is a process, which the shell manages.
- Each job is assigned a sequential job ID. A job is a process, therefore, each job has an associated process ID (PID).
- There are three types of job statuses:
  - Foreground
  - Background
  - Stopped

Note: Other shells support job control, except the Bourne shell.

Job Control Commands

- Job control commands enable you to place jobs in the foreground or background, and to start or stop jobs.
- The following table describes the job control commands:

Option	Description
`Ctrl+Z` `(SIGTSTP 19)`	Stops the foreground job and places it in the background as a stopped job
`jobs`	Lists all jobs and their job IDs
`bg [%n]`	Places the current stopped job or the specified job ID in the **b**ack**g**round, where *n* is the job ID
`fg [%n]`	Brings the current or specified job ID from the background to the **f**ore**g**round, where *n* is the job ID
`kill %n`	Deletes the job from the background, where *n* is the job ID
`kill -19 %n`	Or, if signal 19 (SIGSTOP) is used, places the process associated with the job ID (*n*) in a stopped state

Running a Job in the Background

- To run a job in the background, you need to enter the command that you want to run, followed by an ampersand (`&`), which is a shell metacharacter, at the end of the command line.
  - For example, to run the `sleep` command in the background:

```
$ sleep 500 &
[1] 3028
```

  - The shell returns the job ID, in brackets (which it assigns to the command), and the associated PID.

Note: The `sleep` command suspends the execution of a program for *n* seconds.

Bringing a Background Job to the Foreground

- You can use the `jobs` command to list the jobs that are currently running or stopped in the background.

```
$ jobs
[1] + Running                 sleep 500 &
```

- You can use the `fg` command to bring a background job to the foreground.

```
$ fg %1
sleep 500
```

Note: The foreground job occupies the shell until the job is completed or stopped and placed into the background.

Quiz

To run a job in the background, you need to enter the command that you want to run, followed by a pipe (|) symbol at the end of the command line.

a. True

b. False

The `alias` Command

- An `alias` is a shorthand shell notation that allows you to customize and abbreviate commands.

```
$ alias aliasname="command_string"
```

- If the first word on the command line is an *`aliasname`*, the shell replaces that word with the text of the alias.
- The shell maintains a list of aliases that it searches when a command is entered.
- The following rules apply while creating an alias:
  - There can be no whitespace on either side of the equal sign.
  - The command string must be quoted if it includes any options, metacharacters, or whitespace.
  - Each command in a single alias must be separated by a semicolon.

Command Sequence

- You can group several commands under a single *aliasname*.
- Individual commands are separated by semicolons.

```
$ alias info='uname -a; id; date'
$ info
SunOS s11-server1 5.11 11.3 i86pc i386 i86pc
uid=60016(oracle) gid=100(oracle)
Fri Feb 10 15:22:47 UTC 2017
```

Predefined Aliases

- In Oracle Linux, the GNU `bash` shell contains several predefined system aliases.
- You can display these predefined aliases by using the `alias` command.

```
[oracle@ol7-server1 ~]$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

Note: The `alias` command displays both system- and user-defined aliases.

User-Defined Aliases

- User-defined aliases are defined by a user, usually to abbreviate or customize frequently used commands.
- The `history` command is aliased as `h` by using the `alias` command in the following code:

```
$ alias h=history
$ h
278     cat /etc/passwd
279     pwd
280     cp /etc/passwd /tmp
281     ls ~
282     alias h=history
283     h
```

Deactivating an Alias

- You can temporarily deactivate an alias by placing a backslash (\), the shell metacharacter *escape*, in front of the alias on the command line.
- In the following code, the backslash prevents the shell from looking in the alias list. This allows the shell to run the original `rm` command to remove the `file1` file.

```
$ rm file1
rm: remove file1 (yes/no)? n
$ \rm file1
$ ls file1
file1: No such file or directory
```

Removing an Alias

- The `unalias` command removes aliases from the alias list.

```
$ unalias aliasname
```

- The `h` alias that was created earlier is removed by using the `unalias` command.

```
$ unalias h
$ h
bash: h: not found
```

Quiz

Which of the following rules does not apply while creating an alias?

a. There can be no space on either side of the equal sign.

b. The backslash (\) is always placed in front of the alias.

c. The command string in an alias must be quoted if it includes any options, metacharacters, or whitespace.

d. Each command in a single alias must be separated with a semicolon.

Shell Functions

- Functions, which is a powerful feature of shell programming, is a group of commands organized by common functionality.
- There can be hundreds of predefined shell functions.
- These easy-to-manage units, a function when called returns a single value with no additional output.
- Using a function involves two steps:
  - Defining the function
  - Invoking the function

Defining a Function

- A function is defined by using the following general syntax:

```
function functionname [()] { compound-command [redirections]; }
```

- To define a function called `num` that displays the total number of users currently logged in to the system:

```
$ function num { who | wc -l; }
```

  - The `num` function runs the `who` command, whose output is redirected to the `wc` command.

- To remove a function, use the `unset -f` command:

```
$ unset -f num
```

Invoking a Function

You can invoke a function by merely entering the function name on the command line or within the shell script.

```
$ num
```

Shell Options

- Options are switches that control the behavior of the shell. In the `bash` shell, there are about 27 different options.
- Options are of the `Boolean` data type, which means that they can be either `on` or `off`.
- To show the current option settings, enter:

```
$ set -o
```

- To turn on an option, enter:

```
$ set -o option-name
```

- To turn off an option, enter:

```
$ set +o option-name
```

Activating the `noclobber` Shell Option

- Redirecting standard output (`stdout`) to an existing file overwrites the previous file's content that results in data loss.
- This process of overwriting existing data is known as *clobbering*.
- To prevent an overwrite from occurring, the shell supports a `noclobber` option.
- When the `noclobber` option is set, the shell refuses to redirect standard output to the existing file and displays an error message on the screen.
- The `noclobber` option is activated in the shell by using the `set -o` command.

Deactivating the noclobber Shell Option

- To deactivate the noclobber option, enter the following commands:

```
$ set +o noclobber
$ set -o | grep noclobber
noclobber off
```

- To temporarily deactivate the noclobber option, use the >| deactivation syntax on the command line:

```
$ ls -l >| file_new
```

Note: There is no space between the > and | on the command line. The noclobber option is ignored for this command line only, and the contents of the file are overwritten.

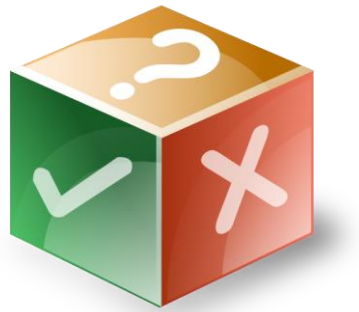
Quiz

Which of the following syntaxes can be used to turn off an option?

a. `$ set +o option-name`

b. `$ set -o e`

c. `$ set -o option-name`



Agenda

- Using advanced shell features
- Writing shell scripts

Shell Scripts

- A shell script is a text file that contains a sequence of commands and comments for a UNIX-like OS, and is designed to be run by the UNIX **shell**, a command-line interpreter.
- Shell scripts are often used to automate repeating command sequences, such as services that start or stop on system startup or shutdown.
- There can be many shell scripting languages, for example, Perl, PHP, Tcl, and so on. However, for this lesson, we will focus on the default shell `bash`.
- Users with little or no programming experience can create and run shell scripts.
- You can run the shell script by simply entering the name of the shell script on the command line.

Determining the Shell to Interpret and Execute a Shell Script

- Oracle Solaris and Oracle Linux support various shell scripting languages, such as GNU Bash, Bourne, Korn, C, Perl, PHP, and others.
- The first line of a shell script identifies the shell program that interprets and executes the commands in the script.
- The first line should always begin with the symbols `#!` (called a `shebang`) followed immediately by the absolute pathname of the shell program used to interpret the script.

```
#!/full-pathname-of-shell
```

- The first line for a `bash` shell script is as follows:

```
#!/usr/bin/bash
```

Creating a Shell Script

- To create a shell script, you need a text editor.
- A text editor is a program that reads and writes text files.
- The following code is a simple shell script:

```
#!/usr/bin/bash
# This is my first shell script.
echo "Hello World!"
```

  - The first line of the script indicates the shell program that interprets the commands in the script. In this case, it is `/usr/bin/bash`.
  - The second line is a comment. Everything that appears after a hash (`#`) symbol is ignored by `bash`.
  - The last line is the `echo` command, which prints what is displayed.

Executing a Shell Script

- After the shell script is created, you can run it.
- To run a shell script, the user must have execute permissions.
  - To grant read and execute permissions to the user so that you can execute the `mycmd` shell script, use the `chmod` command:

```
$ chmod u+rx mycmd
```

- A shell script is executed by just calling out the script name on the command line.
  - To run the `mycmd` script in the current directory, enter `./mycmd` on the command line.

Comments in a Shell Script

- A comment is a textual description of the script and the lines within the script file.
- Comments are always preceded by a hash (#) symbol.

```
# This is a comment inside a shell script
ls -l # lists the files in a directory
```

- Whenever a shell encounters a hash (#) symbol in a script file, the line following it is ignored by the shell.
- The addition of comments in a shell script file does not affect the execution of the script unless a syntactical error is introduced when the comments are added.

Positional Parameters in a Shell Script

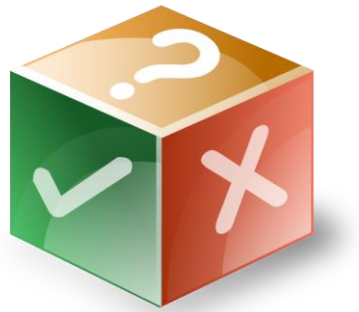
- You can pass command-line arguments to a shell script while it is running.
- As you pass these arguments on the command line, the shell stores the first parameter after the script name into variable `$1`, the second parameter into variable `$2`, and so on.
- These variables are called *positional* parameters.

Quiz

To run a shell script, the user must have write permissions.

a. True

b. False



Checking the Exit Status

- Exit status is a numeric value that indicates the success or failure of a command.
  - A value of zero indicates success.
  - A nonzero value indicates failure.
    - This nonzero value can be any integer in the range of `1–255`.
- All commands in the UNIX and Linux environments return an exit status, which is held in the read-only shell variable `?`.
- A developer can use exit status values to indicate different error situations.
  - The exit status value can be set inside a shell script with the `exit=##` command.
- To check the value of exit status, use the `echo` command and the variable expansion dollar sign (`$`):

```
$ echo $?
0
```

The `test` Command

- The shell built-in `test` command is used for testing conditions.
- The `test` command can be written as a test expression or written using the `[ expression ]` special notation.
- The `test` command is also used for evaluating expressions, such as the following:
  - Variable values
  - File access permissions
  - File types
- There are several types or categories of `bash` shell test comparison operators:
  - Integer/Arithmetic test comparison operators
  - String test comparison operators
  - File test comparison operators
  - And there are other test comparison operators

Integer/Arithmetic `test` Comparison Operators

The Integer/Arithmetic `test` comparison operators use characters:

Operator	Meaning	Syntax

-eq	Equal to	`[ "$var1" -eq "$var2" ]`
		`Note1:  ^       ^   ^       ^ required whitespace (IFS)`
		`Note2:  While not required, the double-quotes (" ") provide excellent variable and value isolation and is considered a Best Practice.`

-ne	Not equal to	`[ "$var1" -ne "$var2" ]`
-le	Less than or equal to	`[ "$var1" -le "$var2" ]`
-ge	Greater than or equal to	`[ "$var1" -ge "$var2" ]`
-lt	Less than	`[ "$var1" -lt "$var2" ]`
-gt	Greater than	`[ "$var1" -gt "$var2" ]`

String `test` Comparison Operators

The String `test` comparison operators use symbols.

Operator	Meaning	Syntax
= or **==**	Equal to	**["$str1" == "$str2"]** **Note: While both = and == are usable as string comparison operators. The = is also used as an *assignment* operator. The == is considered a Best Practice.**
!=	Not equal to	**["$str1" != "$str2"]**
-z	String is *null*, zero length	**["$str1" –z "$str2"]**
-n	String is not *null*	**["$str1" –n "$str2"]**
<	Sorts before	**["$str1" < "$str2"]**
>	Sorts after	**["$str1" > "$str2"]**

File `test` Comparison Operators

The File `test` comparison operators use characters. (Not a complete listing.)

Operator	Meaning	Syntax
-e (or **-a**) ***filename***	File exists	`[ -e filename ]`
-f ***filename***	File is a regular file	`[ -f filename ]`
-d ***filename***	File is directory	`[ -d filename ]`
-c ***filename***	File is a character device	`[ -c filename ]`
-b ***filename***	File is a block device	`[ -d filename ]`
-h (or **-L**) ***filename***	File is a symbolic link	`[ -h filename ]`
-r ***filename***	File is a readable	`[ -r filename ]`
-w ***filename***	File is a writeable	`[ -w filename ]`
-x ***filename***	File is a executable	`[ -x filename ]`
-s ***filename***	File size is bigger than zero bytes (not empty)	`[ -s filename ]`

Using the `test` Command in an `if` Statement

- The `test` *`condition`* command or `[ expression ]` special notation often follows the `if` statement.
- The `test` command evaluates a condition and, if the result of the test is true, it returns an exit status of zero.
- If the result of the test is false, the `test` command returns a nonzero exit status.

```
if test condition
   then
   command
   ...
fi
or
if [ expression ]
   then
   command
   ...
fi
```

How to Test/Debug a Shell Script

There are several `bash` shell command-line options, which can also be included in your shell scripts on the line with the `shebang` after the script interpreter.

- The `-x` option described as a *debugger*
- The `-v` option described as *verbose*
- The `-n` option described as a *syntax checker*

```
$ bash -xv scriptname
or
$ cat scriptname
#!/usr/bin/bash -xv
...(output truncated)
```

Conditional Expressions

The shell provides the following special expressions that enable you to run a command based on the success or failure of the preceding command.

- The `&&` operator
- The `||` operator
- The `if` statement
- The `case` statement

The && Operator

The && (and) operator ensures that the second command is run only if the preceding command succeeds, both commands succeed.

```
$ mkdir $HOME/newdir && cd $HOME/newdir
```

The || Operator

The || (or) operator ensures that the second command is run only if the preceding command fails.

```
$ mkdir /usr/tmp/newdir || mkdir $HOME/newdir
```

The `if` Statement

- The `if` statement evaluates the exit status of a command and initiates additional actions based on the return value.

```
$ if [ command | test ];
 > then                  # on true execute
 >   command1
 >   ...
 > else                  # on false execute
 >   commandn
 >   ...
 > fi
```

- If the exit status is zero (true), any commands that follow the `then` statement are run.
- If the exit status is nonzero (false), any commands that follow the `else` statement are run.

Note: The `if` statement is often used with the `test` command.

The `if` Statement Additional Syntax

Within the `if` statement, there can be multiple nested `if` test using the `elif` syntax in place of an `else`.

```
$ if [ command1 | test1 ];
 > then                          # on true execute
 >   commandn
 >   ...
 > elif [ command2 | test2 ];    # on false run a second nested test
 >   then                        # on true from the second nested test execute
 >     commandn
 >     ...
 >   else                        # on false from the second nested test execute
 >     commandn
 >     ...
 > fi
```

The `case` Statement

The `case` command compares a single value against other values, and runs a command or group of commands when a match is found.

```
$ case value in
> pat1)
> command1
> ...
> ;;
> patn)
> commandn
> ...
> ;;
> *)         # The * is a catch-all
> echo "Usage: $0 { pat1 | patn }"
> exit 3
> ;;
> esac
```

Looping Constructs

Many programming/scripting languages provide structures to iterate through a list of objects. The `bash` shell provides the following three loop structures.

- The `for` loop statement
- The `while` (true) loop statement
- The `until` (true) loop statement

The `for` Loop Statement

- The `for` command enables you to repeat a command or group of commands in a loop.

```
$ for arg in [ command | test ]
> do
>   commandn
>   ...
> done
```

- The `for` command evaluates the exit status of the `in` operation that follows it.
  - If the exit status is zero, any instructions that follow the `do` statement are run, `command` or `test` is rerun, and the exit status rechecked.
  - If the exit status is nonzero, the loop terminates.

Shifting Positional Parameters in a Loop

- While passing command-line arguments, the Bourne (`sh`) shell accepts only a single number after the `$` sign (`$0`–`$9`).
- An attempt to access the value in the tenth argument using the notation `$10` results in the value of `$1` followed by a zero (`0`).
- Both the Korn (`ksh`) shell and Bash (`bash`) shell can access the 10th parameter directly with the value of the 10th argument `${10}`. However, that could become very cumbersome in a loop.
- The `shift` command enables you to shift your positional parameter values back by one position when processing the positional parameters in a loop.
  - The value of the `$2` parameter becomes assigned to the `$1` parameter.
  - Therefore, there is **no limit** on the number of positional parameters that can be passed to a shell script.

The `while` (True) Loop Statement

- The `while` command enables you to repeat a command or group of commands in a loop.

```
$ while [ command | test ]
> do
>   commandn
>   ...
> done
```

- The `while` command evaluates the exit status of the `command` or `test` command that follows it.
  - If the exit status is zero, any instructions that follow the `do` statement are run, `command` or `test` is rerun, and the exit status rechecked.
  - If the exit status is nonzero, the loop terminates.

The `until` (True) Loop Statement

- The `until` command enables you to repeat a command or a group of commands in a loop:

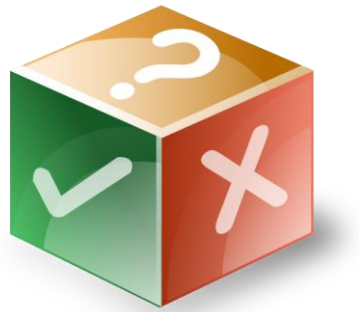
```
$ until [ command | test ]
> do
>   commandn
>   ...
> done
```

- The `until` command evaluates the exit status of the `command` or the `test` command that follows it.
  - If the exit status is nonzero, any instructions that follow the `do` statement are run, `command` or `test` is rerun, and the exit status rechecked.
  - If the exit status is zero, the loop terminates.

Quiz

Which of the following evaluate the exit status of a command and initiate additional actions based on the return values?

a. The `case` statement
b. The `test` command
c. The `if` statement
d. The `while` statement



Summary

In this lesson, you should have learned how to:

- Use advanced shell features
- Write shell scripts

Practice 8: Overview

This practice covers the following topics:

- 8-1: Using advanced Bash shell functionality
- 8-2: Using shell scripts
- 8-3: Using the `test` command and conditional expressions