5

Using Features Within the Bash Shell

Objectives

After completing this lesson, you should be able to:

- Use shell expansion for generating shell tokens
- Use shell metacharacters for command redirection
- Use variables in the `bash` shell to store values
- Display the command history
- Customize the user's work environment

Lesson Agenda

- Using Shell Expansion for Generating Shell Tokens
- Using Shell Metacharacters for Command Redirection
- Using Variables in the `bash` Shell to Store Values
- Displaying the Command History
- Customizing the User’s Work Environment

Shell Expansions

- While working in a shell, sets or ranges of information are often repeated.
- Shell expansions help generate a large number of shell tokens by using compact syntaxes.
- Expansion is performed on the command line after the command is split into tokens.
- Some of the more common types of shell expansions are:
  - Brace expansion
  - Tilde expansion
  - Parameter expansion
  - Command substitution
  - Path name expansion/file name generation

Brace Expansion

- The brace (`{ }`) expansion is a mechanism by which arbitrary strings may be generated.
- Patterns to be brace-expanded take the form of an optional preamble, followed by either a series of comma-separated strings or a sequence expression between a pair of curly braces, followed by an optional postscript.

Brace expansion syntax: **`optional`** ***`preamble`*** **`{string1[,string2][,`*****`stringn`*****`]}optional`** ***`postscript`***

- In this syntax, the **preamble** "`a`" is prefixed to each string contained within the braces, and the **postscript** "`e`" is then appended to each resulting string, expanding left to right.

```
$ echo a{d,c,b}e
ade ace abe
```

Tilde Expansion

The tilde expansion includes:

- The tilde (~) symbol, which represents the home directory of the current user
- The tilde (~) symbol with a username, which represents the home directory of the specified user

Parameter Expansion

- In UNIX and Linux, there can be hundreds of parameters/variables.
- The parameter expansion includes:
  - The dollar sign (`$`) symbol
- The following example shows just two variables, `USER` and `HOME`:

```
$ echo $USER
oracle
$ echo $HOME
/home/oracle
```

Command Substitution

- Command substitution allows you to use the output of a command as an expression to another command (much like running a command within a command).
- The command substitution includes:
  - Dollar sign and a pair of open/close round bracket (`$( )`) symbols (`$(command)`)
  - A pair of backquotes (backticks) (`` `command ` ``)
- Use the `ls -l` to list an executable file, when you do not know which directory it is in. Start with the `which` command to locate the file and then use command substitution to complete the process.

```
$ which passwd
/usr/bin/passwd
$ ls -l $(which passwd)
-rwsr-xr-x 1 oracle oracle ... passwd
or
$ ls -l `which passwd`
-rwsr-xr-x 1 oracle oracle ... passwd
```

Path Name Expansion and File Name Generation

- The path name expansion simplifies location changes within the directory hierarchy.
- The path name expansion or file name generation includes:
  - The asterisk (`*`) symbol, which matches zero or more characters (sometimes called “globbing”)
  - The question mark (`?`) symbol, which matches zero or a single character
  - A pair of square brackets (`[ ]`), which matches a single character
  - The dash (`-`) symbol, which represents the previous working directory

Note: The asterisk, question mark, and square brackets are metacharacters also used by regular expressions.

Asterisk (*) Expansion Symbol

- The asterisk (*) expansion symbol is also a wildcard character or glob, and matches zero or more characters, except the leading period (.) of a hidden file.
- List all files and directories that start with the letter f followed by zero or more other characters.

```
$ cd
$ ls f*
feathers       file.1  file.2  file.3  file4   fruit2
feathers_6     file1   file2   file3   fruit
```

Question Mark (?) Expansion Symbol

- The question mark (?) expansion symbol is also a wildcard character and matches any single character, except the leading period (.) of a hidden file.
- List all files and directories that start with the string dir and followed by one other character.

```
$ ls dir?
dir1:
coffees fruit trees

dir2:
beans notes recipes

dir3:
cosmos moon planets space sun vegetables
...(output truncated)
```

Square Bracket (`[]`) Expansion Symbols

- The square bracket (`[]`) expansion symbols are used to create a *character class,* which represents a set or range of characters for a *single* character position.
  - A set of characters is any number of specific characters, for example, [`acb`].
    - The characters in a set do not necessarily have to be in any order, for example, [`abc`] is the same as [`cab`].
  - A range of characters is a series of ordered characters.
    - A range lists the first character followed by a hyphen `(-)` and then the last character, for example, [`a-z`] or [`0-9`].
    - When specifying a range, arrange the characters in the order that you want them to appear in the output, for example, use [`A-Z`] or [`a-z`] to search for any uppercase or lowercase alphabetical character, respectively.

Quiz

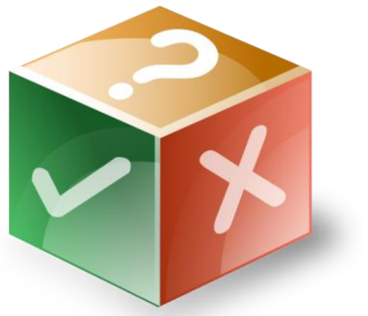
Which of the following expansion symbols equates to the absolute path name of the user’s home directory?

a. #

b. []

c. *

d. ~



Lesson Agenda

- Using Shell Expansion for Generating Shell Tokens
- Using Shell Metacharacters for Command Redirection
- Using Variables in the `bash` Shell to Store Values
- Displaying the Command History
- Customizing the User’s Work Environment

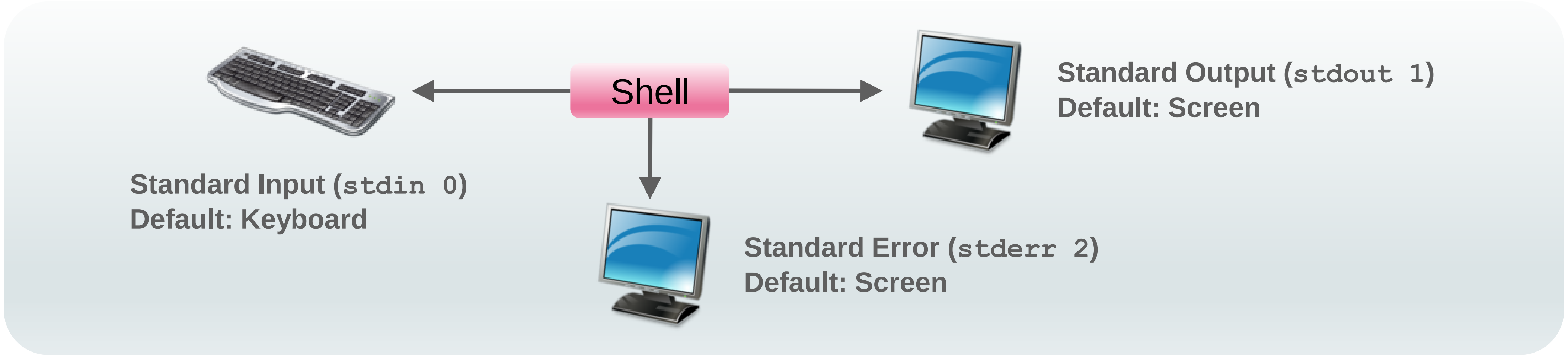
Shell Metacharacters

- Shell metacharacters are specific characters, generally symbols, that have special meaning within the shell.
- `bash` metacharacters:
  - `|` pipe, sends the output of the command on the left as input to the command on the right of the symbol.
  - `&` ampersand, background execution
  - `;` semicolon, command separator
  - `\` backslash, escapes the next metacharacter to remove its meaning.
  - `( )` round brackets (parentheses), command grouping
  - `< > >> &` angle brackets (less-than and greater-than), redirection symbols
  - `` ` ` `` `$( )` backquote (backtick), command substitution
  - `space tab newline` "whitespace" Internal Field Separator (IFS)

Note: The subsequent slides on this topic cover only the redirection symbols.

Command Communication Channels

- By default, the shell receives or reads input from the standard input—the keyboard—and displays the output and error messages to the standard output—the screen.


Shell
Standard Output (`stdout 1`)
Default: Screen
Standard Input (`stdin 0`)
Default: Keyboard
Standard Error (`stderr 2`)
Default: Screen

- Input redirection forces a command to read the input from a file instead of from the keyboard.
- Output redirection sends the output from a command into a file instead of sending the output to the screen.

File Descriptors

- Each process works with three file descriptors.
- File descriptors determine where the input to the command originates and where the output and error messages are directed to.
- The table explains the file descriptors.

File Descriptor Number	File Description Abbreviation	Definition
0	`stdin`	Standard command input
1	`stdout`	Standard command output
2	`stderr`	Standard command error

Redirection Metacharacters

Command redirection is enabled by the following shell metacharacters:

- Redirection of standard input (`<`)
- Redirection of standard output (`>`)
- Redirection of standard output (`>>`) append
- Redirection of standard error (`2>`)
- Redirection of both standard error and standard output (`2>&1`) to the same file
- The pipe symbol (`|`)

Redirecting Standard Input (stdin)

- The less-than (<) metacharacter processes a file as the standard input instead of reading the input from the keyboard.

```
command < filename
or
command 0< filename
```

- Use the dante file as the input for the mailx command.

```
$ mailx oracle < ~/lab/dante
```

Redirecting Standard Output (`stdout`)

- The greater-than (>) metacharacter directs the standard output to a file instead of printing the output to the screen.

```
command > filename
or
command 1> filename
and
command >> filename    # appends
```

- If the file does not exist, the shell **creates** it. If the file exists, the redirection **overwrites** the content of the file, and the >> **appends** the output to the end of the file.
- Redirect the list of files and subdirectories of your current home directory into a `directory_list` file.

```
$ cd
$ pwd
/home/oracle
$ ls -l > directory_list
```

Redirecting Standard Error (`stderr`)

- A command using the file descriptor number (`2`) and the greater-than (`>`) sign redirects any standard error messages to the `/dev/null` file (delete them).

```
command 2>/dev/null
```

- The following example shows the standard output and the standard error redirected to the `dat` file.

```
$ ls /var /test 1> dat 2>&1
$ less dat
ls: cannot access /test: No such file or directory (stderr)
/var: (stdout)
adm (stdout)
...(output truncated)
```

Note: The syntax `2>&1` instructs the shell to redirect `stderr (2)` to the same file that receives `stdout (1)`.

Pipe Symbol

- The pipe (`|`) metacharacter redirects the standard output from one command to the standard input of another command.
- The first command writes the output to standard output and the second command reads standard output from the previous command as standard input.

```
command1 | command2
```

- Use the standard output from the `who` command as the standard input for the `wc -l` command.

```
$ who | wc -l
35
```

Note: You can use pipes to connect several commands.

Using the Pipe Symbol

- To view a list of all the subdirectories located in the `/etc` directory, enter the following command.

```
$ ls -F /etc | grep "/"
X11/
acct/
apache/
apache2/
apoc/
...(output truncated)
```

- Use the output of the `head` command as the input for the `tail` command and print (`lp` - line printer) the results.

```
$ head -10 dante | tail -3 | lp
request id is printerA-177 (Standard input)
```

Redirecting Standard Output (`stdout`) by Using the `tee` Command

- As you saw earlier, the greater-than (>) metacharacter directs the standard output to a file instead of printing the output to the screen.

```
command > filename
```

- Hypothetically, if you wanted to see the output from command1 before it is redirected to a file name, you would use the `tee` command.

```
command1 | tee [-a] filename
```

- When using the `tee` command, if the file does not exist, the shell **creates** it. If the file exists, the `tee` redirection **overwrites** the contents of the file, and if the [–`a`] option (**append**) is used, the redirected output is appended to the end of the file.

Quoting Symbols

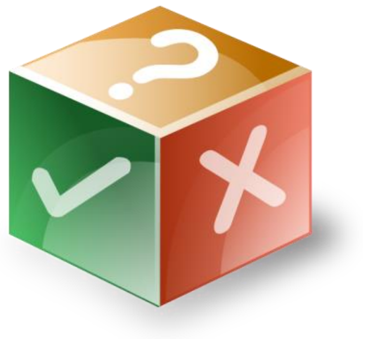
- Quoting is a process that instructs the shell to mask or ignore the special meaning of shell metacharacters.
- The quoting symbols are:
  - Apostrophe or single forward quotation marks (`'  '`): Instructs the shell to ignore all enclosed metacharacters
  - Double quotation marks (`"  "`): Instructs the shell to ignore all enclosed metacharacters and white space, except for the following three symbols:
    - Backslash (`\`) "escape symbol": Prevents the shell from interpreting the next symbol after the (`\`) as a metacharacter
    - Single backward quotation marks (`` `  ` ``) backquote or backtick: Instructs the shell to execute and display the output for a command enclosed within the backward quotation marks
    - Dollar sign and parentheses `$`(command): Instruct the shell to execute and display the output of the command enclosed within the parentheses

Quiz

The `ls -l 2> directory_list` command lists the content of your current directory and redirects that list into a file called `directory_list`.

a. True

b. False



Practice 5: Overview

This practice covers only the highlighted topics:

- 5-1: Using Shell Metacharacters
- 5-2: Using Command Redirection
- 5-3: Using Variables in the `bash` shell
- 5-4: Displaying Command History
- 5-5: Customizing the User's Work Environment

Lesson Agenda

- Using Shell Expansion for Generating Shell Tokens
- Using Shell Metacharacters for Command Redirection
- Using Variables in the **`bash`** Shell to Store Values
- Displaying the Command History
- Customizing the User's Work Environment

Variables: Introduction

- A variable/parameter is a temporary storage area in memory, which is either set by the user, shell, system, or any program that loads another program.
- There are two categories of variables:
  - Local shell variables apply only to the current instance of the shell and are used to set short-term working conditions.
  - Global environment shell variables are local shell variables that have been `export`(ed). The `export`(ed) variables are a subset of the total shell variables and are valid for the duration of any `fork`(ed) or `spawn`(ed) subordinate session.

Displaying Local Shell Variables

- The `echo` command displays the value stored inside a local shell variable using parameter expansion.

```
$ echo $SHELL
/bin/bash
```

- The `set` command lists all local shell variables and their values.

```
$ set
DISPLAY=:0.0
EDITOR=/bin/vim
ERRNO=13
FCEDIT=/bin/vim
HELPPATH=/usr/openwin/lib/locale:/usr/openwin/lib/help
HOME=/home/oracle
HZ=100
...(output truncated)
```

Displaying Global Environment Shell Variables

- The `echo` command displays the value stored inside an environment shell variable.

```
$ echo $SHELL
/usr/bin/bash
```

- The `env` command lists all global environment shell variables and their values.

```
$ env
SHELL=/usr/bin/bash
UID=1000
HOME=/home/oracle
USERNAME=oracle
...(output truncated)
```

ORACLE

Setting and Unsetting Shell Variables

- To create a `bash` local shell variable.

```
$ history=50
$ echo $history
```

- To unset a local shell variable.

```
$ history=
$ echo $history
```

- To create a `bash` environment shell variable, use the `export` command.

```
$ export history=75
$ env | grep history
history=75
$ echo $history
```

ORACLE

Default Bash Shell Variables

Variable	Meaning
`EDITOR`	Defines the default editor for the shell
`FCEDIT`	Defines the editor for the `fc` command. Used with the history mechanism for editing previously executed commands.
`HOME`	Sets the directory to which the `cd` command changes when no argument is supplied on the command line
`LOGNAME`	Sets the login name of the user
`PATH`	Specifies a colon-separated list of directories to be searched when the shell needs to find a command to be executed
`PS1`	Specifies the primary `bash` shell prompt: `$`
`PS2`	Specifies the secondary `bash` command prompt, normally: `>`
`SHELL`	Specifies the name of the shell (that is, `/usr/bin/bash`)

Customizing `bash` Shell Variables: PS1

- The shell prompt string is stored in the shell variable `PS1`, and you can customize it according to your preference.

```
$ PS1='[\u@\h \W]\$'
[oracle@ol7-server1 ~]$
```

  – In this Oracle Linux example, the prompt displays the user's login name "`\u`", the system's host name "`\h`" , and the current working directory "`\W`".
  – This shell prompt displays the correct information even when the user logs in to different hosts.
  – In Oracle Solaris the `PS1` variable is slightly different, observe the colon "`:`" between "`\h`" and "`\W`".

```
$ PS1='[\u@\h:\W]\$'
[oracle@s11-server1:~]$
```

Customizing Shell Variables: `PATH`

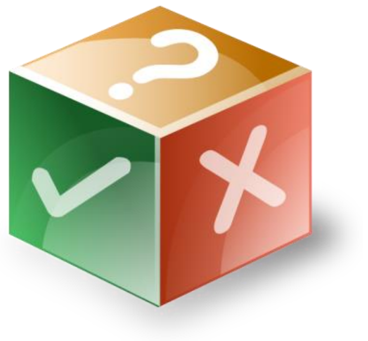
- The `PATH` variable contains a list of directory path names, separated by colons.
- When executing a command on the command line, the shell searches the directories listed in the `PATH` variable from left to right, in sequence to locate that command.
- If the shell does not find the command in the list of directories, it displays a “not found” error message.
- To ensure that commands operate smoothly, you must include the respective directory in the `PATH` variable.
- The example in the notes pages illustrates the inclusion of the `/home/oracle/lab` directory into the `PATH` variable and the use of `which`, a bash shell built-in.

Quiz

The `set` command lists all local shell variables and their values.

a. True

b. False



Lesson Agenda

- Using Shell Expansion for Generating Shell Tokens
- Using Shell Metacharacters for Command Redirection
- Using Variables in the `bash` Shell to Store Values
- Displaying the Command History
- Customizing the user's Work Environment

ORACLE

Introducing Command History

- The shell keeps a history of previously entered commands.
- There are two global shell variables `HISTFILESIZE` and `HISTSIZE` that control the number of history entries.
- This history mechanism enables you to view, repeat, or modify previously executed commands.
- By default, the `history` command displays all history entries to standard output.

```
$ history
...
109 date
110 cd /etc
111 touch dat1 dat2
112 ps -ef
113 history
```

Note: The output may vary based on the commands recorded in the `~/.bash_history` file when you `exit` a terminal session.

Displaying Previously Executed Commands

To display the last four commands.

```
$ history 4
111 touch dat1 dat2
112 ps -ef
113 history
114 history 4
```

Use the ! Command to Re-execute a Command Line from History

- The exclamation symbol (`!`) command, also called “bang”, is an alias built-in to the `bash` shell, which enables you to repeat a command.
- The output from the `history` command shows a line number in front of the command line. Use “`!###`” to re-execute any command or use a relative location number, for example, “`!-n`”.

```
$ history
...(output truncated)
109 date
110 cd /etc
111 touch dat1 dat2
112 ps -ef
113 history
```

- Re-execute `112 ps -ef` by using the `!` command or by using relative positioning.

```
$ !112
or
$ !-2
```

Use the `!!` Command to Repeat the Last Command

- The `!!` command is an alias built in to the `bash` shell, which enables you to repeat the last command.
- Repeat/re-execute the `cal` command by using “`!!`” or simply recall the last command by pressing the `up arrow` key and then press `Return/Enter` to execute.

```
$ cal
 March 2017
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
...(output truncated)
$ !!
cal
 March 2017
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
...(output truncated)
```

Searching the History Entries

- Pressing the Ctrl + R keys together initiates a search and prompts for the search string.

```
$ Ctrl+r
(reverse-i-search)`':
```

- When entering the search string, `bash` returns the first found match from the bottom of the current working set of commands combined with the `~/.bash_history` file.
  - If you want to search for the last occurrence of the `clear` command, entering “`cl`” returns “`clear`”. Press `Return/Enter` to execute, or press Ctrl + C to cancel.

```
$ Ctrl+r
(reverse-i-search)`cl': clear
```

- If that is not the command you are looking for, pressing Ctrl + R again continues the reverse search.

Using the `!` Command to Search for and Execute History Entries

You can search for ***and*** execute using "`!`" combined with a search string, for example "`!cl`".

```
$ !cl
```

Caution: Please ensure the string you entered is not a string in a destructive command; you may not get the results you are expecting.

Editing Commands on the Command Line

- You can edit commands by using a shell inline editor.
- The default command-line editing mode in `bash` is `emacs`.
- You can, however, switch to `vim (vi)` mode as well.
- The `set -o` command switches between the two modes.

```
$ set -o vi
$ set -o emacs
```

- You can also set the editing mode by using the `EDITOR` or `VISUAL` shell variables.

```
$ export EDITOR=/bin/vim
or
$ export VISUAL=/bin/vim
```

ORACLE

Invoking File Name Completion

- File name completion is a feature that allows you to enter the first part of a file name or directory name and press a key to fill out or complete the file name/directory name.
- To invoke file name completion, enter the desired command followed by one or more characters of a file name and then press the Esc/Tab keys or just press the Tab key.
- Expand a file name beginning with the letters `sb` in the `/usr` directory:

```
$ cd /usr
$ ls sb “Press the Tab key”
```

- The shell completes the remainder of the file name by displaying, `ls sbin/`.

File Name Completion with More Than a Single Solution

- You can request the shell to present all possible alternatives of a partial file name from which you can select.
- This request can be invoked by pressing the Escape (Esc) and the equal (=) sign keys in sequence or by pressing Tab twice.

Lesson Agenda

- Using Shell Expansion for Generating Shell Tokens
- Using Shell Metacharacters for Command Redirection
- Using Variables in the `bash` Shell to Store Values
- Displaying the Command History
- Customizing the User’s Work Environment

User Initialization Files

- Other than having a home directory to create and store files, users need an environment that gives them access to the tools and resources.
- When a user logs in to a system, the user’s work environment is determined by the initialization files.
- These initialization files are defined by the user’s startup shell, which can vary depending on the update or release.
- The default initialization files in your home directory enable you to customize your working environment.

Default User Initialization Files for the `bash` Shell

- When `bash` is invoked, it first reads and executes commands from the `/etc/profile` file, if the file exists.
- `bash` then reads and executes commands from the `~/.bash_profile`, `~/.bash_login` and `~/.bashrc`, which executes `/etc/bashrc`, if it exists.
- In the absence of the aforementioned files, the `~/.profile` file is executed.
- When a login shell exits, `bash` reads and executes commands from the `~/.bash_logout` file, if it exists.

Configuring the `~/.bashrc` File

- The `~/.bashrc` file is a personal initialization file for configuring the user environment.
- The file is defined in your home directory and can be used for the following:
  - Modifying your working environment by setting custom shell environment variables and terminal settings
  - Instructing the system to initiate applications
- However, before the changes can be instantiated the `~/.bashrc` file has to be reread.

Rereading the `~/.bashrc` File

- There are two ways to reread the `~/.bashrc`:
  - `exit` the current terminal session and restart a new terminal session.
  - Use a `bash` shell built-in called `source` also aliased as (`.`) a period to reread the `~/.bashrc` file in the current shell without `fork`(ing) or `spawn`(ing) a subordinate shell.

```
$ source ~/.bashrc
or
$ . ~/.bashrc
```

Quiz

Which of the following is the default command-line editing mode in `bash`?

a. `vi`

b. `ed`

c. `emacs`

d. `vim`

Summary

In this lesson, you should have learned how to:

- Use shell expansion for generating shell tokens
- Use shell metacharacters for command redirection
- Use variables in the `bash` shell to store values
- Display the command history
- Customize the user’s work environment

Practice 5: Overview

This practice covers only the highlighted topics:

- 5-1: Using Shell Metacharacters
- 5-2: Using Command Redirection
- 5-3: Using Variables in the `bash` shell
- 5-4: Displaying Command History
- 5-5: Customizing the User's Work Environment

ORACLE