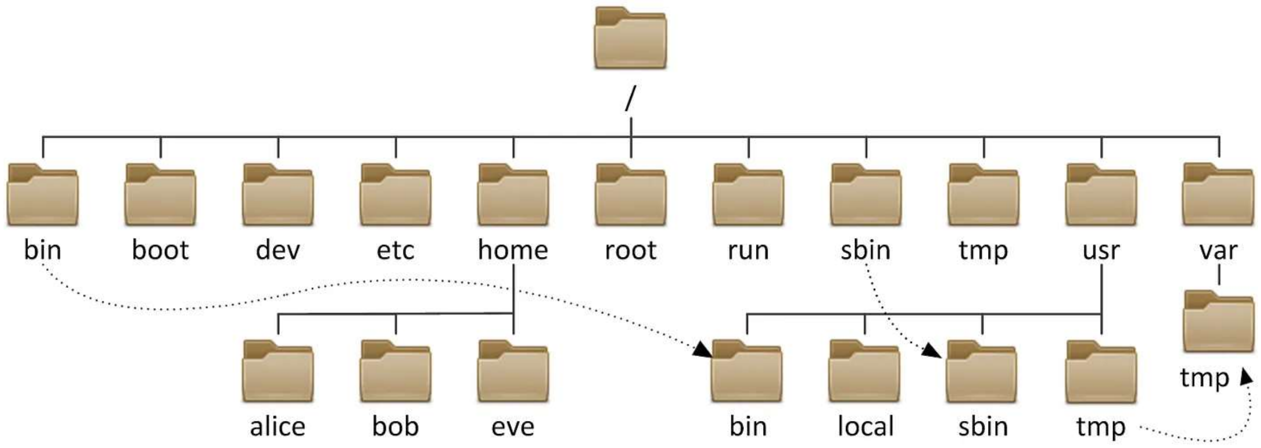
CHAPTER-3 MANAGE FILES FROM THE COMMAND LINE

Linux File System Hierarchy Concepts

The File-system Hierarchy

The Linux system stores all files on file systems, which are organized into a single inverted tree known as a file-system hierarchy. This hierarchy is an inverted tree because the tree root is at the top, and the branches of directories and subdirectories stretch below the root.



The / directory is the root directory at the top of the file-system hierarchy.

Significant Red Hat Enterprise Linux Directories

Directories	
/boot	Files to start the boot process
/dev	Special device files that the system uses to access hardware.
/etc	System-specific configuration files.
/home	Home directory of user
/root	Home directory of root
/run	Runtime data for processes that started since the last boot. This data includes process ID files and lock files. The contents of this directory are re- created on reboot. This directory consolidates the /var/run and /var/lock directories from earlier versions of Red Hat Enterprise Linux.
/tmp	A world-writable space for temporary files. Files that are not accessed, changed, or modified for 10 days are deleted from this directory automatically. The /var/tmp directory is also a temporary directory, in which files that are not accessed, changed, or modified in more than 30 days are deleted automatically
/usr	Installed software, shared libraries, including files, and read-only program data. Significant subdirectories include: • /usr/bin: User commands • /usr/sbin: System administration commands

	• /usr/local: Locally customized software
/var	System-specific variable data should persist between boots. Files that dynamically change, such as databases, cache directories, log files, printer-spooled documents, and website content, might be found under /var.

Specify Files by Name

The path of a file or directory specifies its unique file-system location. Following a file path traverses one or more named subdirectories, which are delimited by a forward slash (/), until the destination is reached. Directories, also called folders, can contain other files and other subdirectories. Directories are referenced in the same manner as files.

Absolute Paths

An absolute path is a fully qualified name that specifies the exact location of the file in the file or directory its start with /

Relative path

Relative path is defined as the path related to the present working directly (pwd). It starts at your current directory and never starts with a /

DESCRIPTIN	**COMMANDS / OPTIONS**
Displays the full path name of the current working Directory for that shell.	pwd
Home directory	tilde character (~)
List the content of file folder	**Syntax:** ls [option] [file/directory] **-l** long list **-a** hidden file **-R** Recursive **-t** Sort files and directories last modify **-r** reverse order **-S** Sort files and directories by their sizes **-i** inode **-h** file sizes in human-readable format **Example:** [user@host ~]$ ls -l↵
Changes directory	**Syntax:** cd [options][directory_name] **cd** Move to User Home driectory **..** To go inside the Parent Directory **~** To directly come inside the home directory **.** To stay in currently directory **-** change the user to the old directory

Manage Files with Command-line Tools

Creating, removing, copying, and moving files and directories are common operations for a system administrator. Without options, some commands are used to interact with files, or they can manipulate directories with the appropriate set of options.

DESCRIPTIN	**COMMANDS / OPTIONS**
Create Directory	**Syntax:** mkdir [options][directory_name] **-p** parent directory **Example:** [user@host ~]$ mkdir -p /Dir1/Dir2 ↵ **-v** Enables verbose mode **-m** Sets file modes or permissions **Example:** [user@host ~]$ mkdir -m a=rwx [directories] ↵
Copy file directory	**Syntax:** cp [options] source_file destination **-r** Recursive **-f** Forcefully **-i** Interactive copying with a warning **-b** Creates a backup of the destination file **Example:** [user@host ~]$ cp -b hello.txt world.txt ↵
Move or rename file Dir	**To Move File** **Syntax:** mv [options(s)] [source_file_name(s)] [Destination_file_name] **-i** (interactive) **-f** (Force) **-n** (no-clobber) **-b** (backup) **To Rename a directory** **Syntax:** mv [source_directory_name(s)] [Destination_directory_name]
To remove files	**Syntax:** rm [options] FileName **-r** Recursive **-f** Force **-i** (interactive)
Remove empty dir	rmdir [options] dirName

Make Links between Files

You can create multiple file names that point to the same file. These file names are called links. You can create two types of links: a hard link, or a symbolic link (sometimes called a soft link). Each way has its advantages and disadvantages.

Create Hard Links

Every file starts with a single hard link, from its initial name to the data on the file system. When you create a hard link to a file, you create another name that points to that same data. The new hard link acts exactly like the original file name. After the link is created, you cannot tell the difference between the new hard link and the original name of the file.

Hard links have some limitations.

First, you can use hard links only with regular files. You cannot use the ln command to create a hard link to a directory or special file.

Second, you can use hard links only if both files are on the same file system. The file-system hierarchy can be composed of multiple storage devices. Depending on the configuration of your system, when you change into a new directory, that directory and its contents might be stored on a different file system.

DESCRIPTIN	COMMANDS / OPTIONS
To create Links	**To create Hard Links** **Syntax:** ln [OPTION]... [-T] TARGET LINK_NAME ln [OPTION]... TARGET... DIRECTORY ln [OPTION]... -t DIRECTORY TARGET **Example:** creates a hard link called newfile-hlink2.txt for the existing newfile.txt file in the /tmp directory. [user@host ~]$ **ln newfile.txt /tmp/newfile-hlink2.txt** ↲

Create Symbolic Links

Which is also called a "soft link". A symbolic link is not a regular file, but a special type of file that points to an existing file or directory.
Symbolic links have some advantages over hard links:
• Symbolic links can link two files on different file systems.
• Symbolic links can point to a directory or special file, not just to a regular file.

DESCRIPTIN	**COMMANDS / OPTIONS**
To create Soft Links	ln -s [original filename] [link name] **Example:** the ln -s command creates a symbolic link for the /home/user/ newfile-link2.txt file. The name for the symbolic link is /tmp/newfile-symlink.txt. [user@host ~]$ **ln -s /home/user/newfile-link2.txt /tmp/newfile-symlink.txt**

<u>Match File Names with Shell Expansions</u>

When you type a command at the Bash shell prompt, the shell processes that command line through multiple expansions before running it. You can use these shell expansions to perform complex tasks that would otherwise be difficult or impossible.

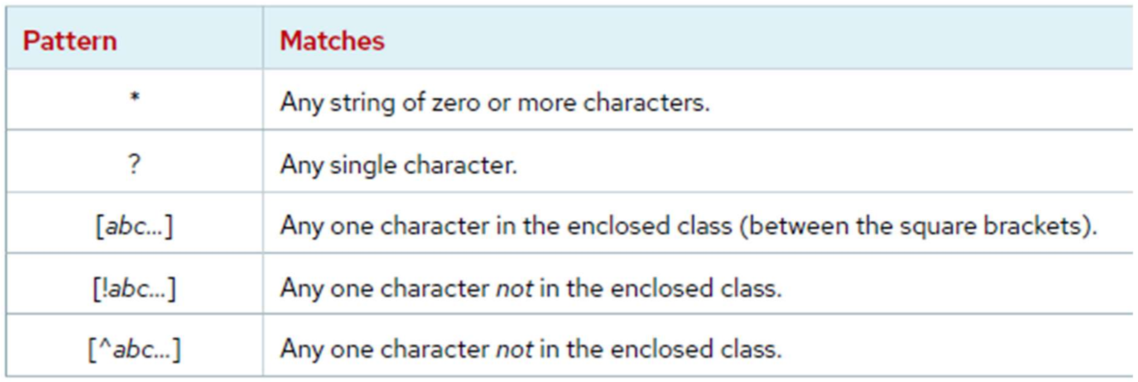
The main expansions that Bash performs are:

- brace expansion that can generate multiple strings of characters
- tilde expansion that expand to a path to a user home directory
- variable expansion that replaces text with the value stored in a shell variable
- command substitution that replaces text with the output of a command
- pathname expansion that helps select one or more files by pattern matching

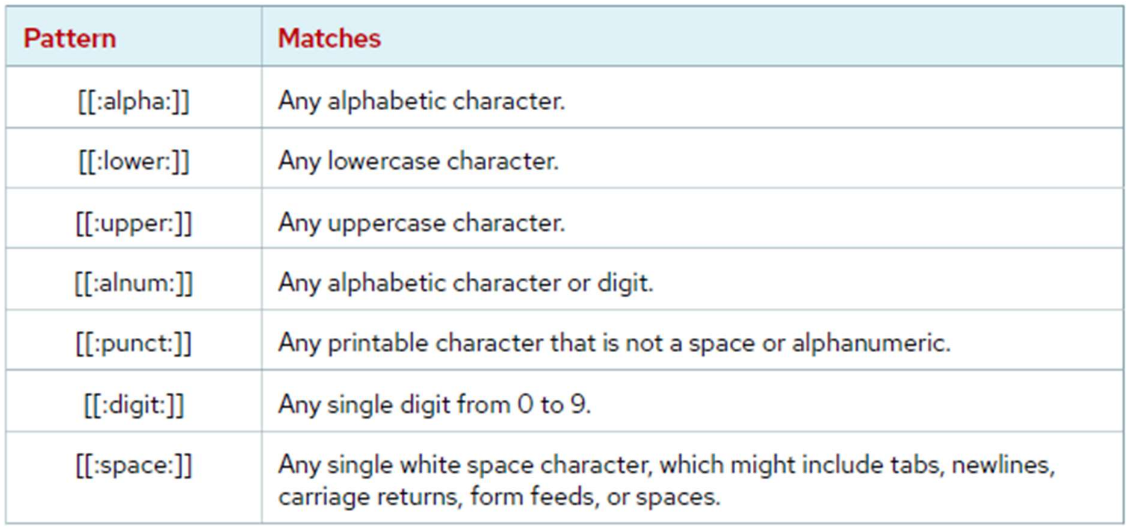
Pathname expansion, historically called globbing, is one of the most useful features of Bash By using metacharacters that "expand" to match the file and path names that you are looking for, commands can act on a focused set of files at once.

Pathname Expansion and Pattern Matching

Pathname expansion expands a pattern of special characters that represent wild cards or classes of characters into a list of file names that match the pattern.



Pattern	Matches
*	Any string of zero or more characters.
?	Any single character.
[*abc*...]	Any one character in the enclosed class (between the square brackets).
[!*abc*...]	Any one character *not* in the enclosed class.
[^*abc*...]	Any one character *not* in the enclosed class.



Pattern	Matches
[[:alpha:]]	Any alphabetic character.
[[:lower:]]	Any lowercase character.
[[:upper:]]	Any uppercase character.
[[:alnum:]]	Any alphabetic character or digit.
[[:punct:]]	Any printable character that is not a space or alphanumeric.
[[:digit:]]	Any single digit from 0 to 9.
[[:space:]]	Any single white space character, which might include tabs, newlines, carriage returns, form feeds, or spaces.

Brace Expansion

Brace expansion is used to generate discretionary strings of characters. **Braces contain a comma separated list of strings,** or a sequence expression. The result includes the text that precedes or follows the brace definition. Brace expansions might be nested, one inside another. You can also use **double-dot syntax (..), which expands to a sequence**.

Example: the {m..p} double-dot syntax inside braces expands to m n o p.
A practical use of brace expansion is to quickly create multiple files or directories.

```
[user@host glob]$ mkdir ../RHEL{7,8,9}
```

Variable Expansion

A variable acts like a named container that stores a value in memory. Variables simplify accessing and modifying the stored data either from the command line or within a shell script.

You can assign data as a value to a variable with the following syntax:

```
[user@host ~]$ VARIABLENAME=value
```

You can use variable expansion to convert the variable name to its value on the command line. If a string starts with a dollar sign ($), then the shell tries to use the rest of that string as a variable name and replace it with the variable value.

```
[user@host ~]$ USERNAME=operator
[user@host ~]$ echo $USERNAME
```

To prevent mistakes due to other shell expansions, you can put the name of the variable in curly braces, for example ${VARIABLENAME}.

```
[user@host ~]$ USERNAME=operator
[user@host ~]$ echo ${USERNAME}
Operator
```

Command Substitution

Command substitution allows the output of a command to replace the command itself on the command line. Command substitution occurs when you enclose a command in parentheses and precede it by a dollar sign ($). The $(command) form can nest multiple command expansions inside each other.

```
[user@host glob]$ echo Today is $(date +%A).
Today is Wednesday.
```

```
[user@host glob]$ echo The time is $(date +%M) minutes past $(date +%l%p).
The time is 26 minutes past 11AM.
```

Protecting Arguments from Expansion

Many characters have a special meaning in the Bash shell. To prevent shell expansions on parts of your command line, you can quote and escape characters and strings.
The backslash (\) is an escape character in the Bash shell. It protects the following character from expansion.

```
[user@host glob]$ echo The value of $HOME is your home directory.
The value of /home/user is your home directory.
```

```
[user@host glob]$ echo The value of \$HOME is your home directory.
The value of $HOME is your home directory.
```

To protect longer character strings, you can use single quotation marks (') or double quotation marks (") to enclose strings. They have slightly different effects. Single quotation marks stop all shell expansion. Double quotation marks stop most shell expansion.