Introduction - shell

⇒ Computers understand the language of 0's and 1's called binary language.
⇒ In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in O/S there is special program called Shell.
⇒ Shell accepts your instruction or commands in English (mostly) and if it's a valid command, it is passed to kernel.
⇒ Shell is a user program or its environment provided for user interaction.
⇒ Shell is a command language interpreter that executes commands read from the standard input device (keyboard) or from a file.
⇒ Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.
⇒ Shell is used from a terminal (in GUI), will issue a prompt before reading a command. By default the prompt is **"$"**.
⇒ Several shell available with Linux/UNIX including:

Shell Name	Developed by	Where	Remark
BASH (Bourne-Again SHell)	Brian Fox and Chet Ramey	Free Software Foundation	Most common shell in Linux. It's Freeware shell.
CSH (C SHell)	Bill Joy	University of California (For BSD)	The C shell's syntax and usage are very similar to the C programming language.
KSH (KornSHell)	David Korn	AT & T Bell Labs	--
TCSH	See the man page. Type $ man tcsh	--	TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH).

Tip: To find all available shells in your system type following command:

```
$ cat /etc/shells ⟵
```

⇒ **Note** that each shell does the same job, but each understand different command syntax and provides different built-in functions.
⇒ In MS-DOS, Shell name is COMMAND.COM which is also used for same purpose, but it's not as powerful as our Linux Shells are!
⇒ Any of the above shell reads command from user (via Keyboard or Mouse) and tells Linux/UNIX OS what users want.

⇒ If we are giving commands from keyboard it is called command line interface (Usually in-front of $ prompt, This prompt is depend upon your shell and Environment that you set or by your System Administrator, therefore you may get different prompt).

Tip: To find your current shell type following command

```
$ echo $SHELL ↵
```

Unix Command Line Structure

A **command** is a program that tells the Unix system to do something. It has the form:

command [**options**] [**arguments**]

where an **argument** indicates on what the command is to perform its action, usually a file or series of files. An option modifies the command, changing the way it performs.

Commands are case sensitive. **command** and **Command** are not the same.

Options are generally preceded by a hyphen (**-**), and for most commands, more than one option can be strung together, in the form:

command -[option][option][option]

e.g.:

ls -alR

will perform a long list on all files in the current directory and recursively perform the list through all sub-directories.

For most commands you can separate the options, preceding each with a hyphen, e.g.:

command -option1 -option2 -option3

as in:

ls -a -l -R

Some commands have options that require parameters. Options requiring parameters are usually specified separately, e.g.:

lpr -Pprinter3 -# 2 file

will send 2 copies of file to printer3.

These are the standard conventions for commands. However, not all Unix commands will follow the standard. Some don't require the hyphen before options and some won't let you group options together, i.e. they may require that each option be preceded by a hyphen and separated by whitespace from other options and arguments.

Options and syntax for a command are listed in the **man page** for the command.

Shell Metacharacters

Linux for Programmers and Users, Section 5.5.

As was discussed in Structure of a Command, the command options, option arguments and command arguments are separated by the space character. However, we can also use special characters called **metacharacters** in a Unix command that the shell interprets rather than passing to the command.

The Shell Metacharacters are listed here for reference. Many of the metacharacters are described elsewhere in the study guide.

Symbol	**Meaning**
>	Output redirection, (see File Redirection)
>>	Output redirection (append)
<	Input redirection
*	File substitution wildcard; zero or more characters
?	File substitution wildcard; one character
[ ]	File substitution wildcard; any character between brackets
\`cmd\`	Command Substitution
$(cmd)	Command Substitution
|	The Pipe (|)
;	Command sequence, Sequences of Commands
||	OR conditional execution

Symbol	Meaning
&&	AND conditional execution
( )	Group commands, Sequences of Commands
&	Run command in the background, Background Processes
#	Comment
$	Expand the value of a variable
\	Prevent or escape interpretation of the next character
<<	Input redirection (see Here Documents)

4.3.1. How to Avoid Shell Interpretation

Linux for Programmers and Users, Section 5.16.

Sometimes we need to pass metacharacters to the command being run and do not want the shell to interpret them. There are three options to avoid shell interpretation of metacharacters.

1. Escape the metacharacter with a backslash (\). (See also Escaped Characters) Escaping characters can be inconvenient to use when the command line contains several metacharacters that need to be escaped.
2. Use single quotes (' ') around a string. Single quotes protect all characters except the backslash (\).
3. Use double quotes (" "). Double quotes protect all characters except the backslash (\), dollar sign ($) and grave accent (`).

   Double quotes is often the easiest to use because we often want environment variables to be expanded.

Note

Single and double quotes protect each other. For example:

```
$ echo 'Hi "Intro to Unix" Class'
Hi "Intro to Unix" Class

$ echo "Hi 'Intro to Unix' Class"
```

Create Your Own Command in Linux

Linux operating system allows users to create commands and execute them over the command line. To **create a command in Linux**, the first step is to **create a bash script** for the command. The second step is to **make the command executable**.

This tutorial will walk you through both steps and show you how to create your own command in Linux.

Creating a Bash Script

To create a bash script, enter the following code:

```
#!/bin/bash

#on displays the actual folder name
echo "the folder is 'pwd'"

#then the rest of the files
echo "The folder which contains files are 'ls'"
```

Save this file by pressing **CTRL + O** with Nano. Give it the name of your command.

Make the Command Executable

If you try to type the name of your bash script, you will notice that it will be executed (run).
Now, you have to modify the **CHMOD** of the script that you will run by typing:

```
chmod +x yourScript
```

Now, copy your script in the path, **/usr/bin**, like below:

```
cp yourscript /usr/bi
```

Command Line Arguments in Unix Shell Script

The Unix shell is used to run commands, and it allows users to pass run time arguments to these commands.

These arguments, also known as command line parameters, that allows the users to either control the flow of the command or to specify the input data for the command.

we will understand how to work with command line parameters.

While running a command, the user can pass a variable number of parameters in the command line.

Within the command script, the passed parameters are accessible using ‘positional parameters’. These range from $0 to $9, where $0 refers to the name of the command itself, and $1 to $9 are the first through to the ninth parameter, depending on how many parameters were actually passed.

Example:

$ sh hello how to do you do

Here $0 would be assigned sh

$1 would be assigned hello

$2 would be assigned how

And so on …

We will now look at some additional commands to process these parameters

Command Line Arguments in Unix Shell

The Unix shell is used to run commands, and it allows users to pass run time arguments to these commands.

These arguments, also known as command line parameters, that allows the users to either control the flow of the command or to specify the input data for the command.

While running a command, the user can pass a variable number of parameters in the command line.

Within the command script, the passed parameters are accessible using ‘positional parameters’. These range from $0 to $9, where $0 refers to the name of the command itself, and $1 to $9 are the first through to the ninth parameter, depending on how many parameters were actually passed.

Example:

$ sh hello how to do you do

Here $0 would be assigned sh

$1 would be assigned hello

$2 would be assigned how

And so on …

We will now look at some additional commands to process these parameters.

1) **Set**

   This command can be used to set the values of the positional parameters on the command line.

   Example:

```
   $ set how do you do
$ echo $1 $2
how do
```

   Here, "how" was assigned to $1 and "do" was assigned to $2 and so on.

2) shift

This command is used to shift the position of the positional parameters. i.e. $2 will be shifted to $1 all the way to the tenth parameter being shifted to $9. Note that if in case there are more than 9 parameters, this mechanism can be used to read beyond the 9th.

Example:

$ set hello good morning how do you do welcome to Unix tutorial.

Here, 'hello' is assigned to $1, 'good' to $2 and so on to 'to' being assigned to $9. Now the shift command can be used to shift the parameters 'N' places.

Example:

```
$ shift 2
$ echo $1
```

Now $1 will be 'morning' and so on to $8 being 'unix' and $9 being 'tutorial'

Shell Variables:

we will learn how to use Shell variables in Unix. A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

Variable Names

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names –

```
_ALI
TOKEN_A
VAR_1
VAR_2
```

Following are the examples of invalid variable names –

```
2_VAR
-VARIABLE
VAR1-VAR2
VAR_A!
```

The reason you cannot use other characters such as **!**, *****, or **-** is that these characters have a special meaning for the shell.

Defining Variables

Variables are defined as follows –

```
variable_name=variable_value
```

For example –

```
NAME="Zara Ali"
```

The above example defines the variable NAME and assigns the value "Zara Ali" to it. Variables of this type are called **scalar variables**. A scalar variable can hold only one value at a time.

Shell enables you to store any value you want in a variable. For example –

```
VAR1="Zara Ali"
VAR2=100
```

Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign (**$**) –

For example, the following script will access the value of defined variable NAME and print it on STDOUT –

```
#!/bin/sh

NAME="Zara Ali"
echo $NAME
```

The above script will produce the following value –

```
Zara Ali
```

Read-only Variables

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME –

```
#!/bin/sh

NAME="Zara Ali"
readonly NAME
NAME="Qadiri"
```

The above script will generate the following result –

```
/bin/sh: NAME: This variable is read only.
```

Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command –

```
unset variable_name
```

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works –

```
#!/bin/sh

NAME="Zara Ali"
unset NAME
echo $NAME
```

The above example does not print anything. You cannot use the unset command to **unset** variables that are marked **readonly**.

Shell Input/Output Redirections

we will discuss in detail about the Shell input/output redirections. Most Unix system commands take input from your terminal and send the resulting output back to your terminal. A command normally reads its input from the standard input, which happens to be your terminal by default. Similarly, a command normally writes its output to standard output, which is again your terminal by default.

Output Redirection

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as output redirection.

If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal.

Check the following **who** command which redirects the complete output of the command in the users file.

```
$ who > users
```

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. You can check the users file for the complete content −

```
$ cat users
oko         tty01   Sep 12 07:30
ai          tty15   Sep 12 13:32
ruth        tty21   Sep 12 10:10
pat         tty24   Sep 12 13:07
steve       tty25   Sep 12 13:03
$
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider the following example −

```
$ echo line 1 > users
$ cat users
line 1
$
```

You can use >> operator to append the output in an existing file as follows −

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

Input Redirection

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the **greater-than character** > is used for output redirection, the **less-than character** < is used to redirect the input of a command.

The commands that normally take their input from the standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file *users* generated above, you can execute the command as follows −

```
$ wc -l users
2 users
$
```

Upon execution, you will receive the following output. You can count the number of lines in the file by redirecting the standard input of the **wc** command from the file *users* −

```
$ wc -l < users
2
$
```

Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.

In the first case, wc knows that it is reading its input from the file users. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

Loops

will discuss shell loops in Unix. A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. In this chapter, we will examine the following types of loops available to shell programmers −

- The while loop
- The for loop
- The until loop

You will use different loops based on the situation. For example, the while loop executes the given commands until the given condition remains true; the until loop executes until a given condition becomes true.

The **while** loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

Syntax

```
while command
```

```
do
   Statement(s) to be executed if command is true
done
```

Here the Shell *command* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *command* is *false* then no statement will be executed and the program will jump to the next line after the done statement.

Example

Here is a simple example that uses the **while** loop to display the numbers zero to nine −

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

Upon execution, you will receive the following result −

```
0
1
2
3
4
5
6
7
8
9
```

Each time this loop executes, the variable **a** is checked to see whether it has a value that is less than 10. If the value of **a** is less than 10, this test condition has an exit status of 0. In this case, the current value of **a** is displayed and later **a** is incremented by 1.

Unitl

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

Syntax

```
until command
do
```

```
   Statement(s) to be executed until command is true
done
```

Here the Shell *command* is evaluated. If the resulting value is *false*, given *statement(s)* are executed. If the *command* is *true* then no statement will be executed and the program jumps to the next line after the done statement.

Example

Here is a simple example that uses the until loop to display the numbers zero to nine −

```
#!/bin/sh

a=0

until [ ! $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

Upon execution, you will receive the following result −

```
0
1
2
3
4
5
6
7
8
9
```

For loop

The **for** loop operates on lists of items. It repeats a set of commands for every item in a list.

Syntax

```
for var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Example

Here is a simple example that uses the **for** loop to span through the given list of numbers −

```
#!/bin/sh

for var in 0 1 2 3 4 5 6 7 8 9
do
   echo $var
done
```

Upon execution, you will receive the following result −

```
0
1
2
3
4
5
6
7
8
9
```

Following is the example to display all the files starting with **.bash** and available in your home. We will execute this script from my root −

```
#!/bin/sh

for FILE in $HOME/.bash*
do
   echo $FILE
done
```

The above script will produce the following result −

```
/root/.bash_history
/root/.bash_logout
/root/.bash_profile
/root/.bashrc
```