A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Shell Prompt

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time –

```
$date
Thu Jun 25 08:30:19 MST 2009
```

Shell Types

In Unix, there are two major types of shells –

- **Bourne shell –** If you are using a Bourne-type shell, the **$** character is the default prompt.
- **C shell –** If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories –

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow –

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

Introduction to the Bash Shell

The Linux command line is provided by a program called the shell. Over the years, the shell program has evolved to cater to various options.

Different users can be configured to use different shells. But most users prefer to stick with the current default shell. The default shell for many Linux distros is the GNU Bourne-Again Shell (bash). Bash is succeeded by Bourne shell (sh).

When you first launch the shell, it uses a startup script located in the .bashrc or .bash_profile file which allows you to customize the behavior of the shell. When a shell is used interactively, it displays a $ when it is waiting for a command from the user. This is called the shell prompt.

```
[username@host ~]$
```

If shell is running as root, the prompt is changed to #. The superuser shell prompt looks like this:

```
[root@host ~]#
```

Bash is very powerful as it can simplify certain operations that are hard to accomplish efficiently with a GUI. Remember that most servers do not have a GUI, and it is best to learn to use the powers of a command line interface (CLI).

What is a Bash Script?

A bash script is a series of commands written in a file. These are read and executed by the bash program. The program executes line by line.

For example, you can navigate to a certain path, create a folder and spawn a process inside it using the command line.

You can do the same sequence of steps by saving the commands in a bash script and running it. You can run the script any number of times.

Advantages of Bash scripting

Bash scripting is a powerful and versatile tool for automating system administration tasks, managing system resources, and performing other routine tasks in Unix/Linux systems. Some advantages of shell scripting are:

- **Automation**: Shell scripts allow you to automate repetitive tasks and processes, saving time and reducing the risk of errors that can occur with manual execution.

- **Portability**: Shell scripts can be run on various platforms and operating systems, including Unix, Linux, macOS, and even Windows through the use of emulators or virtual machines.
- **Flexibility**: Shell scripts are highly customizable and can be easily modified to suit specific requirements. They can also be combined with other programming languages or utilities to create more powerful scripts.
- **Accessibility**: Shell scripts are easy to write and don't require any special tools or software. They can be edited using any text editor, and most operating systems have a built-in shell interpreter.
- **Integration**: Shell scripts can be integrated with other tools and applications, such as databases, web servers, and cloud services, allowing for more complex automation and system management tasks.
- **Debugging**: Shell scripts are easy to debug, and most shells have built-in debugging and error-reporting tools that can help identify and fix issues quickly.

How Do You Identify a Bash Script?

File extension of .sh.

By naming conventions, bash scripts end with a .sh. However, bash scripts can run perfectly fine without the sh extension.

Scripts start with a bash bang.

Scripts are also identified with a shebang. Shebang is a combination of bash # and bang ! followed the the bash shell path. This is the first line of the script. Shebang tells the shell to execute it via bash shell. Shebang is simply an absolute path to the bash interpreter.

Below is an example of the shebang statement.

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

The path of the bash program can vary. We will see later how to identify it.

Execution rights

Scripts have execution rights for the user executing them.

An execution right is represented by x. In the example below, my user has the rwx (read, write, execute) rights for the file test_script.sh

Create a file named hello_world.sh

```
touch hello_world.sh
```

Find the path to your bash shell.

```
which bash
```

Write the command.

We will echo "hello world" to the console.
Our script will look something like this:

```
#! usr/bin/bash
echo "Hello World"
```

Edit the file hello_world.sh using a text editor of your choice and add the above lines in it.

Provide execution rights to your user.

Modify the file permissions and allow execution of the script by using the command below:

```
chmod u+x hello_world.sh
```

chmod modifies the existing rights of a file for a particular user. We are adding +x to user u.

Run the script.

You can run the script in the following ways:

```
./hello_world.sh
bash hello_world.sh.
```

The Basic Syntax of Bash Scripting

Just like any other programming language, bash scripting follows a set of rules to create programs understandable by the computer. In this section, we will study the syntax of bash scripting.

How to define variables

We can define a variable by using the syntax variable_name=value. To get the value of the variable, add $ before the variable.

```
#!/bin/bash
# A simple variable example
greeting=Hello
```

```
name=Tux
echo $greeting $name
```

Add the following commands in your file and save it:

```
#!/bin/bash
echo "Today is " `date`

echo -e "\nenter the path to directory"
read the_path

echo -e "\n you path has the following files and folders: "
ls $the_path
```

- Line #1: The shebang (#!/bin/bash) points toward the bash shell path.
- Line #2: The echo command is displaying the current date and time on the terminal. Note that the date is in backticks.
- Line #4: We want the user to enter a valid path.

- Line #5: The read command reads the input and stores it in the variable the_path.
- line #8: The ls command takes the variable with the stored path and displays the current files and folders.

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="Ruby Jh"
```

The above example defines the variable NAME and assigns the value "Ruby Jh" 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="Ruby Jh"
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="Ruby Jh"
echo $NAME
```

The above script will produce the following value –

```
Ruby Jh
```

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="Ruby Jh"
readonly NAME
NAME="Ramesh"
```

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="Ruby Jh"
unset NAME
echo $NAME
```

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

Variable Types

When a shell is running, three main types of variables are present –

- **Local Variables –** A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.

- **Environment Variables –** An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.
- **Shell Variables –** A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

There are various operators supported by each shell. We will discuss in detail about Bourne shell (default shell) in this chapter.

We will now discuss the following operators –

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr**.

The following example shows how to add two numbers –

```
#!/bin/sh

val=`expr 2 + 2`
echo "Total value : $val"
```

The following points need to be considered while adding –

- There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- The complete expression should be enclosed between ‘ ‘, called the backtick.

Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then –

```
#!/bin/sh

a=10
```

```
b=20

val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
```

```
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

The above script will produce the following result –

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a is not equal to b
```

The following points need to be considered when using the Arithmetic Operators –

- There must be spaces between the operators and the expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- Complete expression should be enclosed between ‘ ‘, called the inverted commas.

- You should use \ on the * symbol for multiplication.
- It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[$a == $b]** is correct whereas, **[$a==$b]** is incorrect.
- All the arithmetical calculations are done using long integers.

- Bourne Shell supports the following **relational operators** that are specific to numeric values. These operators do not work for string values unless their value is numeric.
- Assume variable **a** holds 10 and variable **b** holds 20 then **–**

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.

-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if	[$a -lt $b] is true.

	yes, then the condition becomes true.	
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, **[$a <= $b]** is correct whereas, **[$a <= $b]** is incorrect.

Boolean Operators

The following Boolean operators are supported by the Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then –

Operator	Description	Example
!	This is logical negation. This inverts a true condition	[! false] is true.

	into false and vice versa.	
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true	[$a -lt 20 -a $b -gt 100] is false.

	otherwise false.	