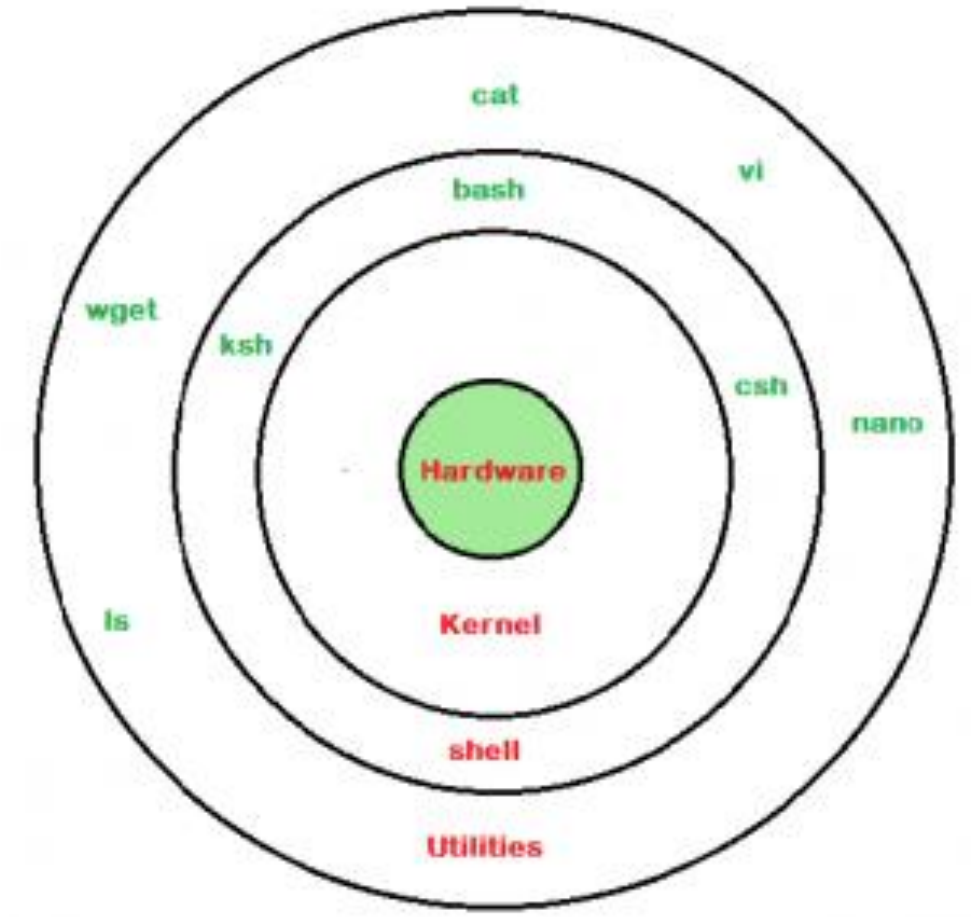
What is Shell?

A shell is a special user program that provides an interface for the user to use operating system services. Shell accepts human-readable commands from users and converts them into something which the kernel can understand. It is a command language interpreter that executes commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or starts the terminal.



Linux Shell

Shell is broadly classified into two categories –

- Command Line Shell
- Graphical shell

- ## Command Line Shell
- Shell can be accessed by users using a command line interface. A special program called Terminal in Linux/macOS, or Command Prompt in Windows OS is provided to type in the human-readable commands such as “cat”, “ls” etc. and then it is being executed.

Graphical Shells

Graphical shells provide means for manipulating programs based on the graphical user interface (GUI), by allowing for operations such as opening, closing, moving, and resizing windows, as well as switching focus between windows. Window OS or Ubuntu OS can be considered as a good example which provides GUI to the user for interacting with the program. Users do not need to type in commands for every action.

Shell Scripting

Usually, shells are interactive, which means they accept commands as input from users and execute them. However, sometimes we want to execute a bunch of commands routinely, so we have to type in all commands each time in the terminal.

As a shell can also take commands as input from file, we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called **Shell Scripts** or **Shell Programs**. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with `**.sh**` file extension e.g., **myscript.sh.**

A shell script has syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. It would be very easy to get started with it.

A shell script comprises the following elements –

- Shell Keywords – if, else, break etc.
- Shell commands – cd, ls, echo, pwd, touch etc.
- Functions
- Control flow – if..then..else, case and shell loops etc.

Why do we need shell scripts?

There are many reasons to write shell scripts:

- To avoid repetitive work and automation
- System admins use shell scripting for routine backups.
- System monitoring
- Adding new functionality to the shell etc.

Types of files in the Linux system.

1. **General Files** – It is also called ordinary files. It may be an image, video, program, or simple text file. These types of files can be in ASCII or Binary format. It is the most commonly used file in the Linux system.
2. **Directory Files** – These types of files are a warehouse for other file types. It may be a directory file within a directory (subdirectory).

3. **Device Files –** In a Windows-like operating system, devices like CD-ROM, and hard drives are represented as drive letters like F: G: H whereas in the Linux system devices are represented as files. As for example, /dev/sda1, /dev/sda2, and so on.

4. These are the common top-level directories associated with the root directory:

Directories	Description
/bin	binary or executable programs.
/etc	system configuration files.
/home	home directory. It is the default current directory.
/opt	optional or third-party software.
/tmp	temporary space, typically cleared on reboot.
/usr	User related programs.
/var	log files.

5. Some other directories in the Linux system:

Directories	Description
/boot	It contains all the boot-related information files and folders such as conf, grub, etc.
/dev	It is the location of the device files such as dev/sda1, dev/sda2, etc.
/lib	It contains kernel modules and a shared library.
/lost+found	It is used to find recovered bits of corrupted files.

Directories	Description
/media	It contains subdirectories where removal media devices are inserted.
/mnt	It contains temporary mount directories for mounting the file system.
/proc	It is a virtual and pseudo-file system to contains info about the running processes with a specific process ID or PID.
/run	It stores volatile runtime data.
/sbin	binary executable programs for an administrator.
/srv	It contains server-specific and server-related files.
/sys	It is a virtual file system for modern Linux distributions to store and allows modification of the devices connected to the system.

Variables

1) Accessing variable

Variable data could be accessed by appending the variable name with '$' as follows:

```
#!/bin/bash

    VAR_1="Devil"
    VAR_2="OWL"

    echo "$VAR_1$VAR_2"
```

Output:

DevilOWL

2) Unsetting Variables

The unset command directs a shell to delete a variable and its stored data from list of variables. It can be used as follows:

```
#!/bin/bash

    var1="Devil"
    var2=23
```

```
echo $var1 $var2

unset var1

echo $var1 $var2
```

Output:

```
DEVIL 23
    23
```

Variable Types

We can discuss three main types of variables:

1) Local Variable:

Variables which are specific to the current instance of shell. They are basically used within the shell, but not available for the program or other shells that are started from within the current shell.

For example:

`name=Jayesh`

In this case the local variable is (name) with the value of Jayesh. Local variables is temporary storage of data within a shell script.

2) Environment Variable:

These variables are commonly used to configure the behavior script and programs that are run by shell. Environment variables are only created once, after which they can be used by any user.

For example:

`export PATH=/usr/local/bin:$PATH` would add `/usr/local/bin` to the beginning of the shell's search path for executable programs.

3) Shell Variables:

Variables that are set by shell itself and help shell to work with functions correctly. It contains both, which means it has both, some variables are Environment variable, and some are Local Variables.

For example:

`$PWD` = Stores working directory

`$HOME` = Stores user's home directory

`$SHELL` = Stores the path to the shell program that is being used.

1. **Special Variables**:

- $0: The name of the script.
- $1, $2, ...: Positional parameters (arguments passed to the script).
- $#: The number of positional parameters.
- $@: All positional parameters as separate words.
- $*: All positional parameters as a single word.
- $?: The exit status of the last command.
- $$: The process ID of the current shell.
- $_: The last argument of the previous command.

2. The following are some basic regular expressions:

Sr. no.	Symbol	Description
1.	.	It is called a wild card character, It matches any one character other than the new line.
2.	^	It matches the start of the string.
3.	$	It matches the end of the string.
4.	*	It matches up to zero or more occurrences i.e. any number of times of the character of the string.
5.	\	It is used for escape following character.
6.	()	It is used to match or search for a set of regular expressions.
7.	?	It matches exactly one character in the string or stream.

Exit Status in Shell Scripting

The variable $? serves as a representation of the exit status of the preceding command.

Exit status is a numerical value returned by each command upon completion. Typically, a successful command returns an exit status of 0, while an unsuccessful one returns 1. Some commands may yield different exit statuses to convey specific types of failures.

Special Characters

```
#!/bin/bash

echo "Number of argument passed: $#"
echo "Script name is $0"
echo "The 2nd argument passed is: $2"
echo "Arguments passed to script are: $*"
echo "Exit status of last command that executed:$?" #This is the previous command for $_
echo "Last argument provide to previous command:$_"
echo "PID of current shell is: $$"
echo "Flags are set in the shell: $-"
```

break

The break statement is used to terminate the loop and can be used within a while, for, until, and select loops.

Syntax

break [N]

// N is the number of nested loops.

// This parameter is optional.

// By default the value of N is 1.

continue

Continue is a command which is used to skip the remaining command inside the loop for the current iteration in for, while, and until loop.

Syntax:

continue [N]

// the optional parameter N specifies the nth enclosing loop to continue from.

// This parameter is optional.

// By default the value of N is 1.

Demo topics

Calender,who.mkdir,cd rmdir,$,home,