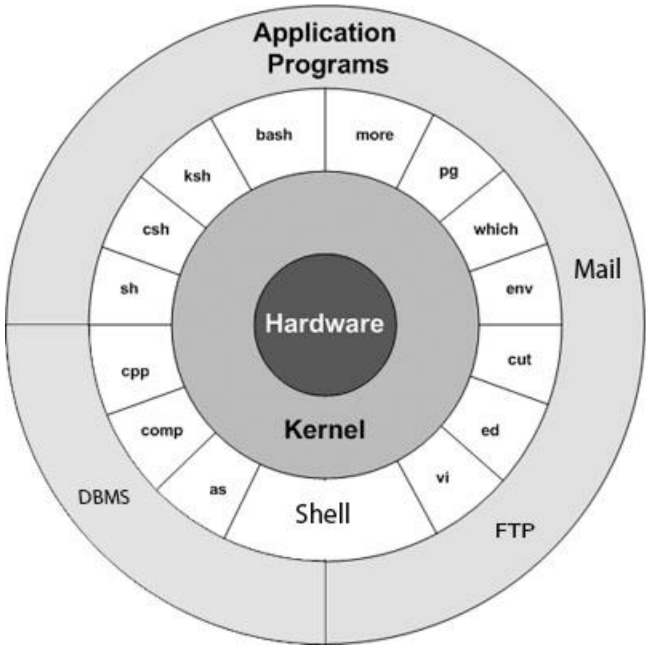
INTRODUCTION TO LINUX

What is Unix ?

- The UNIX operating system is a set of programs that act as a link between the computer and the user.
- The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the operating system or kernel.
- Users communicate with the kernel through a program known as the shell.
- The shell is a command line interpreter; it translates commands entered by the user and converts them into a language that is understood by the kernel.

- Unix was originally developed in 1969 by a group of AT&T employees at Bell Labs, including Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna.
- There are various Unix variants available in the market. Solaris Unix, AIX, HP Unix and BSD are few examples.
- Linux is also a flavor of Unix which is freely available.
- Several people can use a UNIX computer at the same time; hence UNIX is called a multiuser system.
- A user can also run multiple programs at the same time; hence UNIX is called multitasking.

Unix Architecture:



Kernel: The kernel is the heart of the operating system. It interacts with hardware and most of the tasks like memory management, tash scheduling and file management.

Shell: The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are most famous shells which are available with most of the Unix variants.

Commands and Utilities: There are various command and utilities which you would use in your day to day activities. **cp, mv, cat** and **grep** etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various optional options.

- **Files and Directories:** All data in UNIX is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

File System: Type of Files

- A simple description of the UNIX system, also applicable to Linux, is this:
  - "On a UNIX system, everything is a file; if something is not a file, it is a process.“
- This statement is true because there are special files that are more than just files (named pipes and sockets, for instance),
- but to keep things simple, saying that everything is a file is an acceptable generalization.
- A Linux system, just like UNIX, makes no difference between a file and a directory,
- since a directory is just a file containing names of other files.
- Programs, services, texts, images, and so forth, are all files.
- Input and output devices, and generally all devices, are considered to be files, according to the system.

File System: Type of Files

- Most files are just files, called *regular* files;
  - they contain normal data, for example text files, executable files or programs, input for or output from a program and so on.
- *Directories*: files that are lists of other files.
- *Special files*: the mechanism used for input and output. Most special files are in /dev.
- *Links*: a system to make a file or directory visible in multiple parts of the system's file tree..
- *(Domain) sockets*: a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.
- *Named pipes*: act more or less like sockets and form a way for processes to communicate with each other, without using network socket semantics.

- jaime:~/Documents> **ls -l**

```
total 80
-rw-rw-r-- 1 jaime jaime 31744 Feb 21 17:56 intro Linux.doc
-rw-rw-r-- 1 jaime jaime 41472 Feb 21 17:56 Linux.doc
 drwxrwxr-x 2 jaime jaime 4096 Feb 25 11:50 course
```

Symbol	**Meaning**
-	Regular file
d	Directory
l	Link
c	Special file
s	Socket
p	Named pipe
b	Block device

INODE

- In a file system, a file is represented by an *inode*,
  - a kind of serial number containing information about the actual data that makes up the file: to whom this file belongs, and where is it located on the hard disk.
- Each inode describes a data structure on the hard disk, storing the properties of a file, including the physical location of the file data.
- When a hard disk is initialized to accept data storage, usually during the initial system installation process or when adding extra disks to an existing system, a fixed number of inodes per partition is created.
- This number will be the maximum amount of files, of all types (including directories, special files, links etc.) that can exist at the same time on the partition.
- We typically count on having 1 inode per 2 to 8 kilobytes of storage.

INODE

- At the time a new file is created, it gets a free inode. In that inode is the following information:
  - Owner and group owner of the file.
  - File type (regular, directory, ...)
  - Permissions on the file
  - Date and time of creation, last read and change.
  - Date and time this information has been changed in the inode.
  - Number of links to this file (see later in this chapter).
  - File size
  - An address defining the actual location of the file data.
- The only information not included in an inode, is the file name and directory.
  - These are stored in the special directory files.
- By comparing file names and inode numbers, the system can make up a tree-structure that the user understands.
- Users can display inode numbers using the -i option to ls. The inodes have their own separate space on the disk.

Path: Absolute and Relative

- A path, which is the way you need to follow in the tree structure to reach a given file, can be described as starting from the trunk of the tree (the / or root directory).
- In that case, the path starts with a slash and is called an absolute path, since there can be no mistake: only one file on the system can comply.
- In the other case, the path doesn't start with a slash and confusion is possible between ~/bin/wc (in the user's home directory) and bin/wc in /usr.
- Paths that don't start with a slash are always relative.
- In relative paths we also use the . and .. indications for the current and the parent directory

The Kernel

- The kernel is the heart of the system.
- It manages the communication between the underlying hardware and the peripherals.
- The kernel also makes sure that processes and daemons (server processes) are started and stopped at the exact right times.

What is a shell?

- A shell can best be compared with a way of talking to the computer, a language.
- A shell manages the interaction between the system and its users.
- The shell is an advanced way of communicating with the system, because it allows for two-way conversation and taking initiative.
- The shell allows the user to handle a system in a very flexible way.
- An additional asset is that the shell allows for task automation.

Shell Types

- **sh** or Bourne Shell: the original shell still used on UNIX systems and in UNIX related environments. This is the basic shell, a small program with few features
- **bash** or Bourne Again SHell: the standard GNU shell, intuitive and flexible. Probably most advisable for beginning users while being at the same time a powerful tool for the advanced and professional user.
  - On Linux, **bash** is the standard shell for common users. This shell is a so-called *superset* of the Bourne shell, a set of add-ons and plug-ins. This means that the Bourne Again SHell is compatible with the Bourne shell: commands that work in **sh**, also work in **bash**.
- **csh** or C Shell: the syntax of this shell resembles that of the C programming language. Sometimes asked for by programmers.
- **tcsh** or Turbo C Shell: a superset of the common C Shell, enhancing user-friendliness and speed.
- **ksh** or the Korn shell: sometimes appreciated by people with a UNIX background. A superset of the Bourne shell; with standard configuration a nightmare for beginning users.

Unix/Linux commands: Creating directories

- A way of keeping things in place is to give certain files specific default locations by creating directories and subdirectories
- This is done with the **mkdir** command:
  - Example: richard:~> **mkdir archive**
- Creating directories and subdirectories in one step is done using the -p option:
  - **mkdir -p 2001/reports/Restaurants-Michelin/**
  - **Not right: mkdir 2001/reports/Restaurants-Michelin/**
- If you give more than one directory on the command line, mkdir creates each of the directories.
- For example: mkdir docs pub
  - Creates the directories docs and pub under the current directory.

Change directory: cd

- This UNIX command is used to change the current directory name to another directory name
- Directories are separated by a forward slash ("**/**").
- For instance, the directory name "**documents/work/accounting**" means "the directory named **accounting**, which is in the directory named **work**, which is in the directory named **documents** which is in the current directory.“
- To change into this directory, and make it our working directory, we would use the command:
  - cd documents/work/accounting
  - cd ..: to change back to parent directory

Moving Files

- using the **mv** command:
  - richard:~/archive> **mv ../report[1-4].doc reports/Restaurants-Michelin/**
- This command is also applicable when renaming files:
  - richard:~> **mv To_Do done**

Copying Files

- Copying files and directories is done with the **cp** command.
- cp fromfile tofile

Removing files

- Use the **rm** command to remove single files
- **rmdir** to remove empty directories.
- The **rm** command also has options for removing non-empty directories with all their subdirectories
- To protect the beginning user from this malice, the interactive behavior of the **rm**, **cp** and **mv** commands can be activated using the -i option.
    - In that case the system won't immediately act upon request. Instead it will ask for confirmation, so it takes an additional click on the **Enter** key to inflict the damage

Cat command

- **cat** stands for "catenate."
- It reads data from files, and outputs their contents.
- It is the simplest way to display the contents of a file at the command line.
- **cat** is one of the most commonly-used commands in Linux. It can be used to:
  - Display text files
  - Copy text files into a new document
  - Append the contents of a text file to the end of another text file, combining them

Cat: Displaying Text Files

- The simplest way to use **cat** is to simply give it the name of a text file. It will display the contents of the text file on the screen.
- For instance: cat mytext.txt
- If **mytext.txt** is very long, they will zoom past and you will only see the last screen's worth of your document.
- If you want to view the document page-by-page or scroll back and forth through the document, you can use a pager or viewer such as **pg**, **more**, or **less**.
- If you specify more than one file name, **cat** will display those files one after the other, *catenating* their contents to standard output. So this command:
  - cat mytext.txt mytext2.txt
  - Will print the contents of those two text files as if they were a single file.

Cat: **File Creation**

- It is accomplished by typing *cat* followed by the output redirection operator and the name of the file to be created, then pressing ENTER and finally simultaneously pressing the CONTROL(Ctrl) and *d* keys.
- For example, a new file named *file1* can be created by typing cat > file1
- then pressing the ENTER key and finally simultaneously pressing the CONTROL(Ctrl) and *d* keys.

Cat: Copy A Text File

- Normally you would copy a file with the **cp** command. You can use **cat** to make copies of text files in much the same way.
- **cat** sends its output to stdout (standard output), which is usually the terminal screen. However, you can redirect this output to a file using the shell redirection symbol "**>**".
- For instance: cat mytext.txt > newfile.txt
- Similarly, you can catenate several files into your destination file.
- For instance: cat mytext.txt mytext2.txt > newfile.txt

Cat: Append A Text File's Contents To Another Text File

- Instead of overwriting another file, you can also append a source text file to another using the redirection operator "**>>**".
- For instance: cat mytext.txt >> another-text-file.txt
  - will read the contents of **mytext.txt**, and write them at the end of **another-text-file.txt**.
  - If **another-text-file.txt** does not already exist, it will be created and the contents of **mytext.txt** will be written to the new file.
- This works for multiple text files as well:
- cat mytext.txt mytext2.txt >> another-text-file.txt
  - will write the combined contents of **mytext.txt** and **mytext2.txt** to the end of **another-text-file.txt**.

Ls command

- Lists the contents of a directory.
- **Examples**
- ls -l
  - Lists the total files in the directory and subdirectories, the names of the files in the current directory, their permissions, the number of subdirectories in directories listed, the size of the file, and the date of last modification.
- ls -laxo
  - Lists files with permissions, shows hidden files, displays them in a column format, and suppresses group information.
- ls ~
  - List the contents of your home directory by adding a tilde after the **ls** command.
- ls /
  - List the contents of your root directory.
- ls ../
  - List the contents of the parent directory.

Ls command

- ls */
  - List the contents of all subdirectories.
- ls -d */
  - Display a list of directories in the current directory.
- ls -ltr
  - List files sorted by the time they were last modified in reverse order (most recently modified files last).

File security: Access Rights

- On a Linux system, every file is owned by a user and a group user.
- There is also a third category of users, those that are not the user owner and don't belong to the group owning the file.
- For each category of users, read, write and execute permissions can be granted or denied.
- Ls command displays file permissions for these three user categories;
  - they are indicated by the nine characters that follow the first character, which is the file type indicator at the beginning of the file properties line.

File security: Access Rights

- marise:~> **ls -l To_Do**

```
        -rw-rw-r-- 1 marise users 5 Jan 15 12:39 To_Do
marise:~> ls -l /bin/ls
        -rwxr-xr-x 1 root root 45948 Aug 9 15:01 /bin/ls*
```

- As seen in the example,
  - the first three characters in this series of nine display access rights for the actual user that owns the file.
  - The next three are for the group owner of the file,
  - the last three for other users.
- The permissions are always in the same order: read, write, execute for the user, the group and the others

File Access Modes

Code	Meaning
0 or -	The access right that is supposed to be on this place is not granted.
4 or r	read access is granted to the user category defined in this place
2 or w	write permission is granted to the user category defined in this place
1 or x	execute permission is granted to the user category defined in this place

Code	Meaning
u	user permissions
g	group permissions
o	permissions for other

chmod command

- A normal consequence of applying strict file permissions, and sometimes a nuisance, is that access rights will need to be changed for all kinds of reasons.
- We use the **chmod** command to do this,
  - and eventually *to chmod* has become an almost acceptable English verb, meaning the changing of the access mode of a file.
  - The **chmod** command can be used with alphanumeric or numeric options, whatever you like best.

```
asim:~> cat hello #!/bin/bash echo "Hello, World"
asim:~> ls -l hello
-rw-rw-r-- 1 asim asim 32 Jan 15 16:29 hello
asim:~> chmod u+x hello
```

- The **+** and **-** operators are used to grant or deny a given right to a given group. Combinations separated by commas are allowed
  - **chmod u+rwx,go-rwx hello**

chmod command

- When using **chmod** with numeric arguments, the values for each granted access right have to be counted together per group.
- Thus we get a 3-digit number, which is the symbolic value for the settings **chmod** has to make.
-

Meaning of the digit values

Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

Chmod command

- The following table lists the most common combinations:

Command	Meaning
chmod 400 file	To protect a file against accidental overwriting.
chmod 500 directory	To protect yourself from accidentally removing, renaming or moving files from this directory.
chmod 600 file	A private file only changeable by the user who entered this command.
chmod 644 file	A publicly readable file that can only be changed by the issuing user.
chmod 660 file	Users belonging to your group can change this file, others don't have any access to it at all.
chmod 700 file	Protects a file against any access from other users, while the issuing user still has full access.
chmod 755 directory	For files that should be readable and executable by others, but only changeable by the issuing user.
chmod 775 file	Standard file sharing mode for a group.
chmod 777 file	Everybody can do everything to this file.

grep Command:

- searches a file or files for lines that have a certain pattern.
  - The syntax is: grep pattern file(s)
- The name "grep" derives from the ed (a UNIX line editor) command g/re/p which means "globally search for a regular expression and print all lines containing it."
- A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.
- The simplest use of grep is to look for a pattern consisting of a single word.
- It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output.

Grep command options

Options of grep	Description
-v	Print all lines that do not match pattern.
-n	Print the matched line and its line number.
-l	Print only the names of files with matching lines (letter "l")
-c	Print only the count of matching lines.
-i	Match either upper- or lowercase.

Vi(m) editor

- Vim stands for "Vi IMproved".
- It used to be "Vi IMitation", but there are so many improvements that a name change was appropriate.
- Vim is a text editor which includes almost all the commands from the UNIX program **vi**
- Commands in the **vi** editor are entered using only the keyboard, which has the advantage that you can keep your fingers on the keyboard and your eyes on the screen, rather than moving your arm repeatedly to the mouse.
- From now onward we will refer vim editor as vi editor

Using the Vi editor

- What makes **vi** confusing to the beginner is that it can operate in two modes: command mode and insert mode.
- The editor always starts in command mode.
  - Commands move you through the text, search, replace, mark blocks and perform other editing tasks,
- and some of them switch the editor to insert mode.
- This means that each key has not one, but likely two meanings:
  - it can either represent a command for the editor when in command mode,
  - or a character that you want in a text when in insert mode.

Basic Vi commands: **Moving through the text**

- Moving through the text is usually possible with the arrow keys. If not, try:
  - **h** to move the cursor to the left
  - **l** to move it to the right
  - **k** to move up
  - **j** to move down

Basic Vi commands: **Basic operations**

- **n dd** will delete n lines starting from the current cursor position.
- **n dw** will delete n words at the right side of the cursor.
- **x** will delete the character on which the cursor is positioned
- **:n** moves to line n of the file.
- **:w** will save (write) the file
- **:q** will exit the editor.
- **:q!** forces the exit when you want to quit a file containing unsaved changes.

Basic Vi commands: **Basic operations**

- **:wq** will save and exit
- **:w newfile** will save the text to newfile.
- **:wq!** overrides read-only permission (if you have the permission to override permissions, for instance when you are using the *root* account.
- **:1, $s/word/anotherword/g** will replace word with anotherword throughout the file.
- **yy** will copy a block of text.
- **n p** will paste it n times.

Commands that switch the editor to insert mode

- **a** will append: it moves the cursor one position to the right before switching to insert mode
- **i** will insert
- **o** will insert a blank line under the current cursor position and move the cursor to that line.
- Pressing the **Esc** key switches back to command mode.
- If you're not sure what mode you're in because you use a really old version of **vi** that doesn't display an "INSERT" message, type **Esc** and you'll be sure to return to command mode.
- It is possible that the system gives a little alert when you are already in command mode when hitting **Esc**, by beeping or giving a visual bell (a flash on the screen). This is normal behavior.

Some commands in insert mode

i	insert text before cursor, until <Esc> hit
I	insert text at beginning of current line, until <Esc> hit
a	append text after cursor, until <Esc> hit
A	append text to end of current line, until <Esc> hit
o	open and put text in a new line below current line, until <Esc> hit
O	open and put text in a new line above current line, until <Esc> hit

Commands:Copy and Paste Commands

- Yy: copies the current line
- Yw: copies the current word from the character, the lowercase w cursor is on until the end of the word
- p: puts the copied text after the cursor
- P: puts the yanked text before the cursor

Deleting text

Command	Description
x	Deletes the character under the cursor location.
X	Deletes the character before the cursor location.
dw	Deletes from the current cursor location to the next word.
d^	Deletes from current cursor position to the beginning of the line.
d$	Deletes from current cursor position to the end of the line.
D	Deletes from the cursor position to the end of the current line.
dd	Deletes the line the cursor is on.

Cursor Movement

- h move left (backspace)
- j move down
- k move up
- l move right (spacebar)
- b move to the beginning of the previous word or punctuation mark
- B move to the beginning of the previous word, ignores punctuation
- e end of next word or punctuation mark
- E end of next word, ignoring punctuation
- H move cursor to the top of the screen
- M move cursor to the middle of the screen
- L move cursor to the bottom of the screen

Shell scripting or programming

- The basic concept of a shell script is a list of commands, which are listed in the order of execution.
- A good shell script will have comments, preceded by a pound sign, #, describing the steps.
- Assume we create a test.sh script. Note all the scripts would have **.sh** extension.
- Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct.
- For example: #!/bin/sh
- This tells the system that the commands that follow are to be executed by the Bourne shell.
- *It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.*

Variables in shell

- To process our data/information, data must be kept in computers RAM memory.
- RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data.
- Programmer can give a unique name to this memory location/address called memory variable or variable (Its a named storage location that may take different values, but only one at a time).

Special Variables

Variable	Description
$0	The filename of the current script.
$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
$#	The number of arguments supplied to a script.
$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.
$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	The process number of the last background command.

Command-line arguments

- The command-line arguments $1, $2, $3,...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.
- #!/bin/sh
- echo "File Name: $0"
- echo "First Parameter : $1"
- echo "First Parameter : $2"
- echo "Quoted Values: $@"
- echo "Quoted Values: $*"
- echo "Total Number of Parameters : $#"

How to define variables

- To define a variable, use following syntax
  Syntax:
  variable name=value
- '**value**' is assigned to given '**variable name**' and Value must be on right side = sign.

- *Example:*
  $ no=10# this is ok
  $ 10=no# Error, NOT Ok, Value must be on right side of = sign.
  To define variable called 'vech' having value Bus
  $ vech=Bus
  To define variable called n having value 10
  $ n=10

How to print or access value of variable

- To print or access a variable use following syntax
  Syntax:
  $variablename
- Define variable vech and n as follows:
  $ vech=Bus
  $ n=10
  To print contains of variable 'vech' type
  $ echo $vech
  It will print 'Bus',To print contains of variable 'n' type command as follows
  $ echo $n

echo command

- Use echo command to display text or value of variable.
- echo [options] [string, variables...]
  Displays text or variables value on screen.
  Options
  - -n Do not output the trailing new line.
  - -e Enable interpretation of the following backslash escaped characters in the strings:
    - \a alert (bell)
    - \b backspace
    - \c suppress trailing new line
    - \n new line
    - \r carriage return
    - \t horizontal tab
    - \\ backslash
- For e.g. **$ echo -e "An apple a day keeps away \a\t\tdoctor\n"**

More about quotes

Quotes	Name	Meaning
"	Double Quotes	"Double Quotes" - Anything enclose in double quotes removed meaning of that characters (except \ and $).
'	Single quotes	'Single quotes' - Enclosed in single quotes remains unchanged.
`	Back quote	\`Back quote\` - To execute command

Example:

$ echo "Today is date"

Can't print message with today's date.

$ echo "Today is \`date\`".

It will print today's date as, Today is Tue Jan ....,Can you see that the \`date\` statement uses back quote?

The read statement

- Use to get input (data from user) from keyboard and store (data) to variable.
- *Syntax:* read variable1, variable2,...variableN

```
$ vi sayH
#
#Script to read your name from key-board
#
echo "Your first name please:"
read fname
echo "Hello $fname, Lets be friend!"
```

Unix - Shell Basic Operators

- Arithmetic Operators.
- Relational Operators.
- Boolean Operators.
- String Operators.
- File Test Operators.
- There are following points to note down:
  - There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
  - Complete expression should be enclosed between ``, called inverted commas.
  - Use expr command for operations

Arithmetic Operators:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	`expr $a + $b` will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
*	Multiplication - Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/	Division - Divides left hand operand by right hand operand	`expr $b / $a` will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
=	Assignment - Assign right operand in left operand	a=$b would assign value of b into a
==	Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!=	Not Equality - Compares two numbers, if both are different then returns true	[$a != $b] would return true

Relational Operators:

- The following table list out relational operators which are specific to numeric values.
- These operators would not work for string values unless their value is numeric.

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

Boolean Operators:

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true then condition would be true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.	[$a -lt 20 -a $b -gt 100] is false.

String Operators:

Operator	Description	Example
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero. If it is zero length then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero. If it is non-zero length then it returns true.	[-z $a] is not false.

File Test Operators:

- to test various properties associated with a Unix file

Operator	Description	Example
-b file	Checks if file is a block special file if yes then condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file if yes then condition becomes true.	[-b $file] is false.
-d file	Check if file is a directory if yes then condition becomes true.	[-d $file] is not true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.	[-g $file] is false.

File Test Operators:

Operator	Description	Example
-k file	Checks if file has its sticky bit set if yes then condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe if yes then condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.	[-t $file] is false.
-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable if yes then condition becomes true.	[-r $file] is true.

File Test Operators:

Operator	Description	Example
-w file	Check if file is writable if yes then condition becomes true.	[-w $file] is true.
-x file	Check if file is execute if yes then condition becomes true.	[-x $file] is true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.	[-s $file] is true.
-e file	Check if file exists. Is true even if file is a directory but exists.	[-e $file] is true.

Unix - Shell Decision Making

- Unix Shell supports conditional statements which are used to perform different actions based on different conditions.
- The following two decision making statements are :
  - The **if...else** statements
  - The **case...esac** statement

The if...else statements:

- If else statements are useful decision making statements which can be used to select an option from a given set of options.
- Unix Shell supports following forms of if..else statement:
  - if...fi statement
  - if...else...fi statement
  - if...elif...else...fi statement

Unix Shell - The if...fi statement

- The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.
- **Syntax:**

```
if [ expression ]
 then
Statement(s) to be executed if expression is true
fi
```

Unix Shell - The if...else...fi statement

- The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.

```
if [ expression ]
then
Statement(s) to be executed if expression is true
else
Statement(s) to be executed if expression is not true
 fi
```

Unix Shell - The if...elif...fi statement

- The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

```
if [ expression 1 ]
then
Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
then
Statement(s) to be executed if expression 3 is true
else
Statement(s) to be executed if no expression is true
fi
```

The case...esac Statement:

- You can use multiple if...elif statements to perform a multiway branch.
- However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.
- Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

Unix Shell - The case...esac Statement

- The basic syntax of the case...esac statement is to give an expression to evaluate and several different statements to execute based on the value of the expression.
- The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

Syntax of case

```
case word in
    pattern1)
        Statement(s) to be executed if pattern1 matches ;;
    pattern2)
        Statement(s) to be executed if pattern2 matches ;;
    pattern3)
        Statement(s) to be executed if pattern3 matches ;; esac
```

Unix - Shell Loop Types

- Loops are a powerful programming tool that enable you to execute a set of commands repeatedly.
- The following types of loops available to shell programmers:
  - The while loop
  - The for loop
  - The until loop
  - The select loop

Unix Shell - The while Loop

- 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
```

Unix Shell - The for Loop

- The for loop operate 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
```

Example

```
#!/bin/bash
for i in 1 2 3 4 5
do echo
"Welcome $i times"
done
```

For loop with ranges

```
#!/bin/bash
for i in {1..5}
do
echo "Welcome $i times"
done
```

Resolve using:

- brace expansion, *{x..y}* is performed before other expansions, so you cannot use that for variable length sequences.
- Instead, use the seq 2 $max method

```
max=10
for i in `seq 2 $max`
do
echo "$i"
done
```

Another variety of for loop

- Bash v4.0+ has inbuilt support for setting up a step value using {START**..**END**..**INCREMENT} syntax:

```
#!/bin/bash
echo "Bash version ${BASH_VERSION}..."
for i in {0..10..2}
do
 echo "Welcome $i times"
done
```

Three-expression bash for loops syntax

- This type of for loop share a common heritage with the C programming language.
- It is characterized by a three-parameter loop control expression; consisting of an initializer (EXP1), a loop-test or condition (EXP2), and a counting expression (EXP3).

```
for (( EXP1; EXP2; EXP3 ))
do
command1
command2
command3
done
```

A representative three-expression example in bash as follows:

```
#!/bin/bash
for (( c=1; c<=5; c++ ))
do
        echo "Welcome $c times..."
done
```

Looping through arrays

```
for (( i = 0 ; i < ${#names[@]} ; i++ ))
do
        echo ${names[$i]}

done
```

Unix Shell - The until Loop

- The until loop executes a set of commands until a condition is true.
- Syntax:

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

Unix Shell - The select Loop

- The *select* loop provides an easy way to create a numbered menu from which users can select options.
- It is useful when you need to ask the user to choose one or more items from a list of choices.
- Syntax:

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

Example of Select

```
#!/bin/ksh
select DRINK in tea cofee water juice appe all none
do
case $DRINK in
tea|cofee|water|all)
        echo "Go to canteen" ;;
juice|appe)
        echo "Available at home" ;;
none) break ;;
*)
echo "ERROR: Invalid selection" ;;
esac
done
```

Unix - Shell Loop Control

- Sometimes you need to stop a loop or skip iterations of the loop.
- two statements used to control shell loops:
  - The **break** statement
  - The **continue** statement
- The **break** statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement.
- It then steps down to the code following the end of the loop.

Regular Expressions

- An expression is a string of characters.
- Those characters having an interpretation above and beyond their literal meaning are called *metacharacters*.
- Regular Expressions are sets of characters and/or metacharacters that match (or specify) patterns.
- A Regular Expression contains one or more of the following:
  - *A character set*. These are the characters retaining their literal meaning. The simplest type of Regular Expression consists *only* of a character set, with no metacharacters.
  - *An anchor*. These designate (*anchor*) the position in the line of text that the RE is to match. For example, ^, and $ are anchors.
  - *Modifiers*. These expand or narrow (*modify*) the range of text the RE is to match. Modifiers include the asterisk, brackets, and the backslash.

Uses

- The main uses for Regular Expressions (*RE*s) are text searches and string manipulation. An RE *matches* a single character or a set of characters -- a string or a part of a string.
- The asterisk -- * -- matches any number of repeats of the character string or RE preceding it, including *zero* instances.
    - "1133*" matches *11 + one or more 3's*: *113*, *1133*, *1133333*, and so forth.
- The *dot* -- . -- matches any one character, except a newline. [2]
    - "13." matches *13 + at least one of any character (including a space)*: *1133*, *11333*, but not *13* (additional character missing).
- The caret -- ^ -- matches the beginning of a line, but sometimes, depending on context, negates the meaning of a set of characters in an RE.
- The dollar sign -- $ -- at the end of an RE matches the end of a line.
    - "XXX$" matches XXX at the end of a line.
    - "^$" matches blank lines.

Uses

- Brackets -- [...] -- enclose a set of characters to match in a single RE.
  - "[xyz]" matches any one of the characters *x*, *y*, or *z*.
  - "[c-n]" matches any one of the characters in the range *c* to *n*.
  - "[B-Pk-y]" matches any one of the characters in the ranges *B* to *P* and *k* to *y*.
  - "[a-z0-9]" matches any single lowercase letter or any digit.
  - "[^b-d]" matches any character *except* those in the range *b* to *d*. This is an instance of ^ negating or inverting the meaning of the following RE (taking on a role similar to ! in a different context).
  - Combined sequences of bracketed characters match common word patterns. "[Yy][Ee][Ss]" matches *yes*, *Yes*, *YES*, *yEs*, and so forth. "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" matches any Social Security number.

Uses

- The backslash -- \ -- escapes a special character, which means that character gets interpreted literally (and is therefore no longer *special*).
  - A "\$" reverts back to its literal meaning of "$", rather than its RE meaning of end-of-line. Likewise a "\\" has the literal meaning of "\".
- Escaped "angle brackets" -- \<...\> -- mark word boundaries.
  - The angle brackets must be escaped, since otherwise they have only their literal character meaning.
  - "\<the\>" matches the word "the," but not the words "them," "there," "other," etc.

Additional Metacharacters

- A regular expression may be followed by one of several repetition operators:
- The period (.) matches any single character.
- ? means that the preceding item is optional, and if found, will be matched at the most, once.
- * means that the preceding item will be matched zero or more times.
- + means the preceding item will be matched one or more times.
- {n} means the preceding item is matched exactly *n* times, while {n,} means the item is matched *n* or more times. {n,m} means that the preceding item is matched at least *n* times, but not more than *m* times. {,m} means that the preceding item is matched, at the most, *m* times.

Character Classes

- **[:alnum:]** matches alphabetic or numeric characters. This is equivalent to **A-Za-z0-9**.
- **[:alpha:]** matches alphabetic characters. This is equivalent to **A-Za-z**.
- **[:blank:]** matches a space or a tab.
- **[:cntrl:]** matches control characters.
- **[:digit:]** matches (decimal) digits. This is equivalent to **0-9**.
- **[:graph:]** (graphic printable characters). Matches characters in the range of ASCII 33 - 126. This is the same as **[:print:]**, below, but excluding the space character.
- **[:lower:]** matches lowercase alphabetic characters. This is equivalent to **a-z**.
- **[:print:]** (printable characters). Matches characters in the range of ASCII 32 - 126. This is the same as **[:graph:]**, above, but adding the space character.
- **[:space:]** matches whitespace characters (space and horizontal tab).
- **[:upper:]** matches uppercase alphabetic characters. This is equivalent to **A-Z**.
- **[:xdigit:]** matches hexadecimal digits. This is equivalent to **0-9A-Fa-f**.

- The ^ anchor specifies that the pattern following it should be at the start of the line.
- The $ anchor specifies that the pattern before it should be at the end of the line.

Character Class Abbreviations

\d	Match any character in the range 0 - 9 (equivalent of POSIX [:digit:])
\D	Match any character NOT in the range 0 - 9 (equivalent of POSIX [^[:digit:]])
\s	Match any whitespace characters (space, tab etc.). (equivalent of POSIX [:space:] EXCEPT VT is not recognized)
\S	Match any character NOT whitespace (space, tab). (equivalent of POSIX [^[:space:]])
\w	Match any character in the range 0 - 9, A - Z and a - z (equivalent of POSIX [:alnum:])
\W	Match any character NOT the range 0 - 9, A - Z and a - z (equivalent of POSIX [^[:alnum:]])