UNIX - WHAT IS SHELL?

The 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.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, 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 command prompt, is issued by the shell. While the prompt is displayed, you can type a command.

The 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 **date** command which displays current date and time:

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

JKDirectory!

SHELL RESPONSIBILITIES:

The shell analyzes each line you type in and initiates execution of the selected program. But the shell also has other responsibilities, as outlined in Figure below:

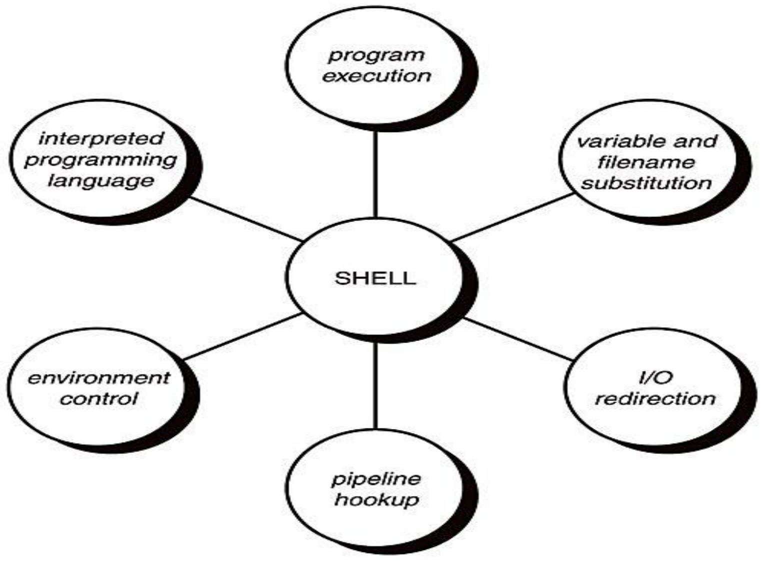

FIGURE: RESPONSIBILITIES OF SHELL

PROGRAM EXECUTION:

The shell is responsible for the execution of all programs that you request from your terminal. Each time you type in a line to the shell, the shell analyzes the line and then determines what to do. As far as the shell is concerned, each line follows the same basic format: ***Program-name arguments***

The line that is typed to the shell is known more formally as the *command line*. The shell scans this command line and determines the name of the program to be executed and what arguments to pass to the program.

The shell uses special characters to determine where the program name starts and ends, and where each argument starts and ends. These characters are collectively called whitespace characters, and are the space character, the horizontal tab character, and the end-of-line character, known more formally as the newline character.

Multiple occurrences of whitespace characters are simply ignored by the shell. When you type the command ***mv tmp/dir games*** the shell scans the command line and takes everything from the start of the line to the first whitespace character as the name of the program to execute: mv.

The set of characters up to the next whitespace character is the first argument to mv: tmp/mazewars. The set of characters up to the next whitespace character is the second argument to mv: games. After analyzing the command line, the shell then proceeds to execute the mv command, giving it the two arguments tmp/mazewars and games.

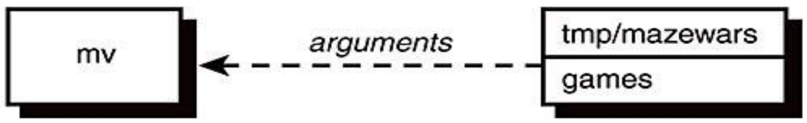

FIGURE: EXECUTION OF MV WITH TWO ARGUMENTS

As mentioned, multiple occurrences of whitespace characters are ignored by the shell. This means that when the shell processes this command line: **echo when do we eat?** It passes four arguments to the echo program: when, do, we, and eat? Because echo takes its arguments and simply displays them at the terminal, separating each by a space character, the output from the following becomes easy to understand:

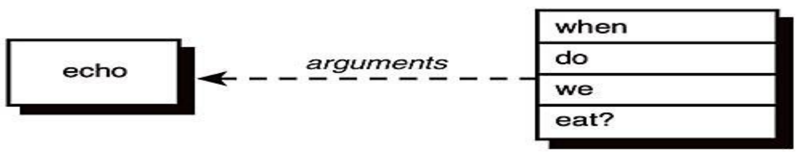

FIGURE: EXECUTION OF ECHO WITH FOUR ARGUMENTS

```
$ echo     hai how are you?
hai how are you?
```

The shell searches the disk until it finds the program you want to execute and then asks the Unix kernel to initiate its execution. This is true most of the time. However, there are some commands that the shell knows how to execute itself.

These built-in commands include cd, pwd, and echo. So before the shell goes searching the disk for a command, the shell first determines whether it's a built-in command, and if it is, the shell executes the command directly.

VARIABLE AND FILENAME SUBSTITUTION:

Like any other programming language, the shell assigns values to variables. Whenever you specify one of these variables on the command line, preceded by a dollar sign, the shell substitutes the value assigned to the variable at that point.

The shell also performs filename substitution on the command line. In fact, the shell scans the command line looking for filename substitution characters *,?, or [...] before determining the name of the program to execute and its arguments. Suppose that your current directory contains the files as shown:

```
$ ls
mrs.todd      prog1
shortcut      Sweeney
```

Now let's use filename substitution for the echo command:

```
$ echo *     List all files

mrs.todd      prog1
shortcut      sweeney
```

When the shell analyzes the line "**echo ***" it recognizes the special character * and substitutes on the command line the names of all files in the current directory.

I/O REDIRECTION:

In this lesson, we will explore a powerful feature used by many command line programs called *input/output redirection*. As we have seen, many commands such as **ls** print their output on the display. This does not have to be the case, however.

By using some special notation we can *redirect* the output of many commands to files, devices, and even to the input of other commands.

STANDARD OUTPUT:

Most command line programs that display their results do so by sending their results to a facility called *standard output*. By default, standard output directs its contents to the display. To redirect standard output to a file, the ">" character is used like this: **ls > file_list.txt**

In this example, the **ls** command is executed and the results are written in a file named file_list.txt. Since the output of **ls** was redirected to the file, no results appear on the display.

Each time the command above is repeated, file_list.txt is overwritten (from the beginning) with the output of the command **ls**. If you want the new results to be *appended* to the file instead, use ">>" like this: **ls >> file_list.txt**

When the results are appended, the new results are added to the end of the file, thus making the file longer each time the command is repeated. If the file does not exist when you attempt to append the redirected output, the file will be created.

STANDARD INPUT:

Many commands can accept input from a facility called *standard input*. By default, standard input gets its contents from the keyboard, but like standard output, it can be redirected. To redirect standard input from a file instead of the keyboard, the "<" character is used like this: **sort < file_list.txt**

In the above example we used the sort command to process the contents of file_list.txt. The results are output on the display since the standard output is not redirected in this example. We could redirect standard output to another file like this:

sort < file_list.txt > sorted_file_list.txt

As you can see, a command can have both its input and output redirected. Be aware that the order of the redirection does not matter. The only requirement is that the redirection operators (the "<" and ">") must appear after the other options and arguments in the command.

PIPES:

Just as the shell scans the command line looking for redirection characters, it also looks for the pipe character |. For each such character that it finds, it connects the standard output from the command preceding the | to the standard input of the one following the |. It then initiates execution of both programs. So when you type: who | wc -l the shell finds the pipe symbol separating the commands who and wc.

It connects the standard output of the former command to the standard input of the latter, and then initiates execution of both commands. When the who command executes, it makes a list of who's logged in and writes the results to standard output, unaware that this is not going to the terminal but to another command instead.

When the wc command executes, it recognizes that no filename was specified and counts the lines on standard input, unaware that standard input is not coming from the terminal but from the output of the who command.

ENVIRONMENT CONTROL:

The shell provides certain commands that let you customize your environment. Your environment includes your home directory, the characters that the shell displays to prompt you to type in a command, and a list of the directories to be searched whenever you request that a program be executed.

INTERPRETED PROGRAMMING LANGUAGE:

The shell has its own built-in programming language. This language is *interpreted*, meaning that the shell analyzes each statement in the language one line at a time and then executes it. This differs from programming languages such as C and FORTRAN, in which the programming statements are typically compiled into a machine-executable form before they are executed.

Programs developed in interpreted programming languages are typically easier to debug and modify than compiled ones. However, they usually take much longer to execute than their compiled equivalents.

The shell programming language provides features you'd find in most other programming languages. It has looping constructs, decision-making statements, variables, and functions, and is procedure-oriented. Modern shells based on the IEEE POSIX standard have many other features including arrays, data typing and built-in arithmetic operations.

SHELL TYPES:

In UNIX there are two major types of shells:

1. The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
2. The C shell. If you are using a C-type shell, the default prompt is the % character.

There are again various subcategories for Bourne Shell which are listed as follows:

- 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)

The original UNIX shell was written in the mid-1970s by Stephen R. Bourne while he was at AT&T Bell Labs in New Jersey.

The Bourne shell was the first shell to appear on UNIX systems, thus it is referred to as "the shell".

JKDirectory!

SHELL VARIABLES:

SPECIAL VARIABLES:

Using certain non alpha numeric characters in your variable names is wrong. This is because those characters are used in the names of special Unix variables. These variables are reserved for specific functions. For example, the $ character represents the process ID number, or PID, of the current shell: **$echo $$**

Above command would write PID of the current shell: 29949

The following table shows a number of special variables that you can use in your shell scripts:

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.

Variable	Description
$@	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. Following script uses various special variables related to command line:

```
#!/bin/sh
echo "File Name: $0"
echo "First Parameter : $1"
echo "First Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Paramers : $#"
```

Here is a sample run for the above script:

```
$sh test.sh abc xyz
File Name : test.sh
First Parameter :abc
First Parameter :xyz
Quoted Values: abc xyz
Quoted Values: abc xyz
Total Number of Paramers : 2
```

SPECIAL PARAMETERS $* AND $@:

There are special parameters that allow accessing all of the command-line arguments at once. $* and $@ both will act the same unless they are enclosed in double quotes, "".

Both the parameter specifies all command-line arguments but the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script shown below to process an unknown number of command-line arguments with either the $* or $@ special parameters:

JKDirectory!

```
#!/bin/sh
for TOKEN in $*
do
  echo $TOKEN
done
```

There is one sample run for the above script:

```
$sh test.sh he is 10 Years Old
he
is
10
Years
Old
```

EXIT STATUS:

$? – is a variable represents the exit status of the previous command. Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure. Following is the example of successful command:

JKDirectory!

```
$sh test.sh hai fine
File Name :test.sh
First Parameter : hai
Second Parameter : fine
Quoted Values: hai fine
Quoted Values: hai fine
Total Number of Parameters : 2
$echo $?
0
```

CONDITIONS (DECISION MAKING):

While writing a shell script, there may be a situation when you need to adopt one path out of the given two paths. So you need to make use of conditional statements that allow your program to make correct decisions and perform right actions.

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. Here we will explain following two decision making statements:

- 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 option

Unix Shell supports following forms of if..else statement:

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
```

Here Shell expression is evaluated. If the resulting value is true, given statement(s) are executed. If expression is false then no statement would be executed. Most of the times you will use comparison operators while making decisions.

JKDirectory!

Give you attention on the spaces between braces and expression. This space is mandatory otherwise you would get syntax error. If expression is a shell command then it would be assumed true if it return 0 after its execution. If it is a boolean expression then it would be true if it returns true.

Example:

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
  echo "a is equal to b"
fi
if [ $a != $b ]
then
  echo "a is not equal to b"
fi
```

This will produce following result:

```
a is not equal to b
```

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.

Syntax:

```
if [ expression ]

then

  Statement(s) to be executed if expression is true

else

  Statement(s) to be executed if expression is not true

fi
```

Here Shell expression is evaluated. If the resulting value is true, given statement(s) are executed. If expression is false then no statement would be executed.

Example:

If we take above example then it can be written in better way using if...else statement as follows:

JKDirectory!

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
  echo "a is equal to b"
else
  echo "a is not equal to b"
fi
```

This will produce following result:

```
a is not equal to b
```

if...elif...else...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.

Syntax:

```
if [ expression1 ]

then

  Statement(s) to be executed if expression 1 is true
```

```
elif [ expression2 ]

then

  Statement(s) to be executed if expression 2 is true

elif [ expression3 ]

then

  Statement(s) to be executed if expression 3 is true

else

  Statement(s) to be executed if no expression is true

fi
```

There is nothing special about this code. It is just a series of if statements, where each if is part of the else clause of the previous statement. Here statement(s) are executed based on the true condition, if none of the condition is true then else block is executed.

Example:

JKDirectory!

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
  echo "a is equal to b"
elif [ $a -gt $b ]
then
  echo "a is greater than b"
elif [ $a -lt $b ]
then
  echo "a is less than b"
else
  echo "None of the condition met"
fi
```

This will produce following result: a is less than b

The case...esac Statement:

You can use multiple if...elif statements to perform a multi way 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. There is only one form of case...esac statement which is detailed here:

Syntax:

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.

```
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
```

Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action. There is no maximum number of patterns, but the minimum is one.

When statement(s) part executes, the command;; indicates that program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

Example:

```
#!/bin/sh
FRUIT="kiwi"
case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."
```

JKDirectory!

```
   ;;
   "banana") echo "I like banana nut bread."
   ;;
   "kiwi") echo "New Zealand is famous for kiwi."
   ;;
 *) echo “There is no option that matches with your option”
esac
```

This will produce following result:

New Zealand is famous for kiwi.

Unix Shell's case...esac is very similar to switch...case statement we have in other programming languages like C or C++ and PERL etc.

CONTROL STRUCTURES:

Loops are a powerful programming tool that enables you to execute a set of commands repeatedly. The following types of loops available to shell programmers:

The for loop:

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

JKDirectory!

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 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
```

This will produce following result: 0 1 2 3 4 5 6 7 8 9

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
```

Here Shell command is evaluated. If the resulting value is true, given statement(s) are executed. If command is false then no statement would be not executed and program would jump to the next line after done statement.

Example:

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

JKDirectory!

```
#!/bin/sh
a=0
while [ $a -lt 10 ]
do
  echo $a
  a=`expr $a + 1`
done
```

This will produce 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 then a is incremented by 1.

Until Loop:

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 Shell command is evaluated. If the resulting value is false, given statement(s) are executed. If command is true then no statement would be not executed and program would jump to the next line after 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
```

This will produce following result: 0 1 2 3 4 5 6 7 8 9

The select loop:

JKDirectory!

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

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.

For every selection a set of commands would be executed with-in the loop. This loop was introduced in ksh and has been adapted into bash. It is not available in sh.

Example: Here is a simple example to let the user select a drink of choice:

```
#!/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
```

The menu presented by the select loop looks like the following:

```
$./test.sh

1) tea

2) cofee

3) water

4) juice

5) appe

6) all

7) none

#? juice

Available at home

#? none

$
```

JKDirectory!

You would use different loops based on different situation. For example while loop would execute given commands until given condition remains true whereas until loop would execute until a given condition becomes true.

Once you have good programming practice you would start using appropriate loop based on situation.

Here while and for loops are available in most of the other programming languages like C, C++ and PERL etc.

Nesting Loops:

All the loops support nesting concept which means you can put one loop inside another similar or different loops. This nesting can go up to unlimited number of times based on your requirement. Here is an example of nesting while loop and similar way other loops can be nested based on programming requirement:

Nesting while Loops:

It is possible to use a while loop as part of the body of another while loop.

Syntax:

```
while command1 ; # this is loop1, the outer loop
do
  Statement(s) to be executed if command1 is true
  while command2 ; # this is loop2, the inner loop
  do
    Statement(s) to be executed if command2 is true
  done
  Statement(s) to be executed if command1 is true
done
```

JKDirectory!

Example:

Here is a simple example of loop nesting, let's add another countdown loop inside the loop that you used to count to nine:

```
#!/bin/sh
a=0
while [ "$a" -lt 10 ]    # this is loop1
do
  b="$a"
  while [ "$b" -ge 0 ]  # this is loop2
  do
    echo -n "$b "
    b=`expr $b - 1`
  done
  echo
  a=`expr $a + 1`
done
```

This will produce following result. It is important to note how echo -n works here. Here -n option let echo to avoid printing a new line character.

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

LOOP CONTROL:

JKDirectory!

So far you have looked at creating loops and working with loops to accomplish different tasks. Sometimes you need to stop a loop or skip iterations of the loop.

In this tutorial you will learn following two statements used to control shell loops:

1. The break statement
2. The continue statement

The infinite Loop:

All the loops have a limited life and they come out once the condition is false or true depending on the loop.

A loop may continue forever due to required condition is not met. A loop that executes forever without terminating executes an infinite number of times. For this reason, such loops are called infinite loops.

Example:

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

```
#!/bin/sh
a=10
while [ $a -ge 10 ]
do
```

```
  echo $a
  a=`expr $a + 1`
done
```

This loop would continue forever because a is always greater than 10 and it would never become less than 10. So this true example of infinite loop

The break 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.

Syntax:

The following break statement would be used to come out of a loop:

```
break
```

The break command can also be used to exit from a nested loop using this format:

```
break n
```

Here n specifies the nth enclosing loop to exit from.

JKDirectory!

Example: Here is a simple example which shows that loop would terminate as soon as it becomes 5:

```
#!/bin/sh
a=0
while [ $a -lt 10 ]
do
  echo $a
  if [ $a -eq 5 ]
  then
    break
  fi
  a=`expr $a + 1`
done
```

This will produce following result:

```
0
1
2
3
4
5
```

Here is a simple example of nested for loop. This script breaks out of both loops if var1 equals 2 and var2 equals 0:

```
#!/bin/sh
for var1 in 1 2 3
do
  for var2 in 0 5
  do
    if [ $var1 -eq 2 -a $var2 -eq 0 ]
    then
      break 2
    else
      echo "$var1 $var2"
    fi
  done
done
```

This will produce following result. In the inner loop, you have a break command with the argument 2. This indicates that if a condition is met you should break out of outer loop and ultimately from inner loop as well.

```
1 0

1 5
```

JKDirectory!

The continue statement:

The continue statement is similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop.

This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

Syntax:

```
Continue
```

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

```
continue n
```

Here n specifies the nth enclosing loop to continue from.

Example:

The following loop makes use of continue statement which returns from the continue statement and start processing next statement:

```
#!/bin/sh
NUMS="1 2 3 4 5 6 7"
for NUM in $NUMS
do
  Q=`expr $NUM % 2`
  if [ $Q -eq 0 ]
  then
    echo "Number is an even number!!"
    continue
  fi
  echo "Found odd number"
done
```

This will produce following result:

```
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
```

JKDirectory!

FUNCTIONS:

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.

Using functions to perform repetitive tasks is an excellent way to create code reuse. Code reuse is an important part of modern object-oriented programming principles.

Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions:

To declare a function, simply use the following syntax:

```
function_name () {

  list of commands

}
```

The name of your function is function_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.

Example:

Following is the simple example of using function:

```
#!/bin/sh

# Define your function here

Hello () {

  echo "Hello World"

}

# Invoke your function

Hello
```

When you would execute above script it would produce following result:

```
$./test.sh

Hello World

$
```

JKDirectory!

PASS PARAMETERS TO A FUNCTION:

You can define a function which would accept parameters while calling that function. These parameters would be represented by $1, $2 and so on.

Following is an example where we pass two parameters Zara and Ali and then we capture and print these parameters in the function.

```
#!/bin/sh
# Define your function here
Hello () {
  echo "Hello World $1 $2"
}
# Invoke your function
Hello Zara Ali
```

This would produce following result:

```
$./test.sh

Hello World Zara Ali

$
```

RETURNING VALUES FROM FUNCTIONS: If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the return command whose syntax is as follows:

```
return code
```

Here code can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Example:

Following function returns a value 1:

```
#!/bin/sh
# Define your function here
Hello () {
  echo "Hello World $1 $2"
  return 10
}
# Invoke your function
Hello Zara Ali

# Capture value returned by last command

ret=$?

echo "Return value is $ret"
```

This would produce following result:

```
$./test.sh

Hello World Zara Ali

Return value is 10

$
```

Nested Functions:

One of the more interesting features of functions is that they can call themselves as well as call other functions. A function that calls itself is known as a recursive function.

Following simple example demonstrates a nesting of two functions:

JKDirectory!

```
#!/bin/sh
# Calling one function from another
number_one () {
  echo "This is the first function speaking..."
  number_two
}
number_two () {
  echo "This is now the second function speaking..."
}
# Calling function one.
number_one
```

This would produce following result:

```
This is the first function speaking...

This is now the second function speaking...
```

Function Call from Prompt:

You can put definitions for commonly used functions inside your .profile so that they'll be available whenever you log in and you can use them at command prompt.

Alternatively, you can group the definitions in a file, say test.sh, and then execute the file in the current shell by typing:

```
$. test.sh
```

This has the effect of causing any functions defined inside test.sh to be read in and defined to the current shell as follows:

```
$ number_one

This is the first function speaking...

This is now the second function speaking...

$
```

Command execution:

There are 4 modes of command execution:

- Sequence Commands
- Grouped commands
- Chained commands
- Conditional commands

JKDirectory!

SEQUENCE COMMANDS:

Eg: $cat file;ls;who; wc file

GROUPED COMMANDS:

Eg: $(cat file;ls;who;wc file)>file name

CHAINED COMMANDS:

Eg: $cat file|cp filename

CONDITIONAL COMMANDS:

Eg: $cat file && cp filename && echo “file copied” ||echo “file not copied

SHELL SCRIPTS:

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.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions. Shell scripts and functions are both interpreted. This means they are not compiled.

JKDirectory!

We are going to write a many scripts in the next several tutorials. This would be a simple text file in which we would put our all the commands and several other required constructs that tell the shell environment what to do and when to do it.

Example Script:

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.

To create a script containing these commands, you put the shebang line first and then add the commands:

```
#!/bin/bash
pwd
ls
```

SHELL COMMENTS:

You can put your comments in your script as follows:

```
#!/bin/bash
pwd
ls
```

Now you save the above content and make this script executable as follows:

```
$chmod +x test.sh
```

Now you have your shell script ready to be executed as follows:

```
$./test.sh
```

This would produce following result:

```
/home/1271          /* output of pwd */

index.htm     unix-basic_utilities.htm     unix-directories.htm

test.sh       unix-communication.htm       unix-environment.htm
```

JKDirectory!

Note: To execute your any program available in current directory using sh progname.sh

EXTENDED SHELL SCRIPTS:

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, however, it is still just a list of commands executed sequentially.

Following script use the read command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

```
#!/bin/sh
# Author : RAJU
# sample script follows here:
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

Here is sample run of the script:

```
$sh test1.sh
What is your name?
raju
Hello,raju
```

TEXT PROCESSING UTILITIES:

All Unix-like operating systems rely heavily on text files for several types of data storage. So it makes sense that there are many tools for manipulating text. The following are some of the list of text processing utilities:

- cat – Concatenate files and print on the standard output
- sort – Sort lines of text files
- uniq – Report or omit repeated lines
- cut – Remove sections from each line of files
- paste – Merge lines of files

APPLICATIONS OF TEXT:

DOCUMENTS:

Many people write documents using plain text formats. While it is easy to see how a small text file could be useful for keeping simple notes, it is also possible to write large documents in text format, as well. One popular approach is to write a large document in a text format and then use a *markup language* to describe the formatting of the finished document.

Many scientific papers are written using this method, as Unix-based text processing systems were among the first systems that supported the advanced typographical layout needed by writers in technical disciplines.

WEB PAGES:

The world's most popular type of electronic document is probably the web page. Web pages are text documents that use either *HTML (Hypertext Markup Language)* or *XML (Extensible Markup Language)* as markup languages to describe the document's visual format.

EMAIL:

Email is an intrinsically text-based medium. Even non-text attachments are converted into a text representation for transmission. We can see this for ourselves by downloading an email message and then viewing it in less.

We will see that the message begins with a *header* that describes the source of the message and the processing it received during its journey, followed by the *body* of the message with its content.

PRINTER OUTPUT:

On Unix-like systems, output destined for a printer is sent as plain text or, if the page contains graphics, is converted into a text format *page description language* known as *PostScript*, which is then sent to a program that generates the graphic dots to be printed.

JKDirectory!

PROGRAM SOURCE CODE:

Many of the command line programs found on Unix-like systems were created to support system administration and software development, and text processing programs are no exception. Many of them are designed to solve software development problems. The reason text processing is important to software developers is that all software starts out as text. *Source code*, the part of the program the programmer actually writes, is always in text format.

CAT COMMAND:

The cat program has a number of interesting options. Many of them are used to help better visualize text content. One example is the -A option, which is used to display nonprinting characters in the text. There are times when we want to know if control characters are embedded in our otherwise visible text.

The most common of these are tab characters (as opposed to spaces) and carriage returns, often present as end-of-line characters in MS-DOS-style text files. Another common situation is a file containing lines of text with trailing spaces. Let's create a test file using cat as a primitive word processor.

To do this, we'll just enter the command cat (along with specifying a file for redirected output) and type our text, followed by Enter to properly end the line, then Ctrl-d, to indicate to cat that we have reached end-of-file.

JKDirectory!

Example:

```
$ cat > foo.txt

The quick brown fox jumped over the lazy dog.

$
```

We will use cat with the -A option to display the text:

```
$ cat -A foo.txt

^IThe quick brown fox jumped over the lazy dog. $

$
```

The command cat also has options that are used to modify text. The two most prominent are -n, which numbers lines, and -s, which suppresses the output of multiple blank lines. We can demonstrate thusly:

```
$ cat > foo.txt

The quick brown fox

jumped over the lazy dog.
```

```
$ cat -ns foo.txt

1 The quick brown fox

2

3 jumped over the lazy dog.

$
```

SORT COMMAND:

The sort program sorts the contents of standard input, or one or more files specified on the command line, and sends the results to standard output. Using the same technique that we used with cat, we can demonstrate processing of standard input directly from the keyboard:

```
$ sort > foo.txt

c

b

a

$ cat foo.txt

a

b

c
```

JKDirectory!

After entering the command, we type the letters “c”, “b”, and “a”, followed once again by Ctrl-d to indicate end-of-file. We then view the resulting file and see that the lines now appear in sorted order. Since sort can accept multiple files on the command line as arguments, it is possible to *merge* multiple files into a single sorted whole.

Example: sort file1.txt file2.txt file3.txt > final_sorted_list.txt

Some Common sort Options:

Option	Long Option	Description
-f	- -ignore-case	Makes sorting case-insensitive.
-n	- -numeric-sort	Performs sorting based on the numeric evaluation of a string. Using this option allows sorting to be performed on numeric values rather than alphabetic values.
-b	- -ignore-leading-blanks	By default, sorting is performed on the entire line, starting with the first character in the line. This option causes sort to ignore leading spaces in lines and calculates sorting based on the first non-whitespace character on the line.

Option	Long Option	Description
-r	- -reverse	Sort in reverse order. Results are in descending rather than ascending order.

Although most of the options above are pretty self-explanatory, some are not. First, let's look at the -n option, used for numeric sorting. With this option, it is possible to sort values based on numeric values. We can demonstrate this by sorting the results of the ***du*** command to determine the largest users of disk space. Normally, the du command lists the results of a summary in pathname order:

```
$ du -s /usr/share/* | head

252          /usr/share/aclocal
96           /usr/share/acpi-support
8            /usr/share/adduser
196          /usr/share/alacarte
344          /usr/share/alsa
8            /usr/share/alsa-base
12488        /usr/share/anthy
8            /usr/share/apmd
21440        /usr/share/app-install
48           /usr/share/application-registry
```

JKDirectory!

UNIQ COMMAND:

Compared to sort, the uniq program is a lightweight. uniq performs a seemingly trivial task. When given a sorted file (including standard input), it removes any duplicate lines and sends the results to standard output. It is often used in conjunction with sort to clean the output of duplicates.

```
$ cat > foo.txt

a

b

c

a

b

c
```

Remember to type Ctrl-d to terminate standard input. Now, if we run uniq on our text file:

```
$ uniq foo.txt
```

```
a
b
c
a
b
c
```

the results are no different from our original file; the duplicates were not removed. For uniq to actually do its job, the input must be sorted first:

$ **sort foo.txt | uniq**

```
a
b
c
```

This is because uniq only removes duplicate lines which are adjacent to each other.

JKDirectory!

Common uniq Options:

Option	Description
`-c`	Output a list of duplicate lines preceded by the number of times the line occurs.
`-d`	Only output repeated lines, rather than unique lines.
`-f` *n*	Ignore *n* leading fields in each line. Fields are separated by whitespace as they are in `sort`; however, unlike `sort`, `uniq` has no option for setting an alternate field separator.
`-i`	Ignore case during the line comparisons.
`-s` *n*	Skip (ignore) the leading *n* characters of each line.
`-u`	Only output unique lines. This is the default.

CUT COMMAND:

The cut program is used to extract a section of text from a line and output the extracted section to standard output. It can accept multiple file arguments or input from standard input. Specifying the section of the line to be extracted is somewhat awkward and is specified using the following options:

cut Selection Options:

Option	Description
`-c` *`char_list`*	Extract the portion of the line defined by *char_list*. The list may consist of one or more comma-separated numerical ranges.
`-f` *`field_list`*	Extract one or more fields from the line as defined by *field_list*. The list may contain one or more fields or field ranges separated by commas.
`-d` *`delim_char`*	When `-f` is specified, use *delim_char* as the field delimiting character. By default, fields must be separated by a single tab character.
`--complement`	Extract the entire line of text, except for those portions specified by `-c` and/or `-f`.

Example: Consider the following text file

```
$ cat -A distros.txt
```

```
SUSE^I10.2^I12/07/2006$
Fedora^I10^I11/25/2008$
SUSE^I11.0^I06/19/2008$
Ubuntu^I8.04^I04/24/2008$
Fedora^I8^I11/08/2007$
SUSE^I10.3^I10/04/2007$
Ubuntu^I6.10^I10/26/2006$
Fedora^I7^I05/31/2007$
Ubuntu^I7.10^I10/18/2007$
Ubuntu^I7.04^I04/19/2007$
SUSE^I10.1^I05/11/2006$
Fedora^I6^I10/24/2006$
Fedora^I9^I05/13/2008$
Ubuntu^I6.06^I06/01/2006$
Ubuntu^I8.10^I10/30/2008$
Fedora^I5^I03/20/2006$
```

JKDirectory!

It looks good. No embedded spaces, just single tab characters between the fields. Since the file uses tabs rather than spaces, we'll use the -f option to extract a field:

```
$ cut -f 3 distros.txt
```

```
12/07/2006
11/25/2008
06/19/2008
04/24/2008
11/08/2007
10/04/2007
10/26/2006
```

```
05/31/2007
10/18/2007
04/19/2007
05/11/2006
10/24/2006
05/13/2008
06/01/2006
10/30/2008
03/20/2006
```

Example:

$ **cut -f 3 distros.txt | cut -c 7-10**

```
2006
2008
2008
2008
2007
2007
2006
2007
2007
2007
2006
```

JKDirectory!

PASTE COMMAND:

The paste command does the opposite of cut. Rather than extracting a column of text from a file, it adds one or more columns of text to a file. It does this by reading multiple files and combining the fields found in each file into a single stream on standard output.

Like cut, paste accepts multiple file arguments and/or standard input. To demonstrate how paste operates, we will perform some surgery on our distros.txt file to produce a chronological list of releases.

From our earlier work with sort, we will first produce a list of distros sorted by date and store the result in a file called distros-by-date.txt:

$ **sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt > distros-by-date.txt**

Next, we will use cut to extract the first two fields from the file (the distro name and version), and store that result in a file named distro-versions.txt:

$ **cut -f 1,2 distros-by-date.txt > distros-versions.txt**

$ **head distros-versions.txt**

```
Fedora      10
Ubuntu      8.10
SUSE        11.0
Fedora      9
Ubuntu      8.04
Fedora      8
Ubuntu      7.10
SUSE        10.3
Fedora      7
Ubuntu      7.04
```

The final piece of preparation is to extract the release dates and store them a file named distro-dates.txt:

```
$ cut -f 3 distros-by-date.txt > distros-dates.txt
$ head distros-dates.txt
```

```
11/25/2008
10/30/2008
06/19/2008
05/13/2008
04/24/2008
11/08/2007
10/18/2007
10/04/2007
05/31/2007
04/19/2007
```

JK Directory!

We now have the parts we need. To complete the process, use paste to put the column of dates ahead of the distro names and versions, thus creating a chronological list. This is done simply by using paste and ordering its arguments in the desired arrangement:

```
$ paste distros-dates.txt distros-versions.txt
```

```
11/25/2008 Fedora   10
10/30/2008 Ubuntu   8.10
06/19/2008 SUSE     11.0
05/13/2008 Fedora   9
04/24/2008 Ubuntu   8.04
11/08/2007 Fedora   8
10/18/2007 Ubuntu   7.10
10/04/2007 SUSE     10.3
05/31/2007 Fedora   7
04/19/2007 Ubuntu   7.04
12/07/2006 SUSE     10.2
10/26/2006 Ubuntu   6.10
```

```
10/24/2006 Fedora   6
06/01/2006 Ubuntu   6.06
05/11/2006 SUSE     10.1
03/20/2006 Fedora   5
```

BACKUP UTILITIES:

One of the primary tasks of a computer system's administrator is keeping the system's data secure. One way this is done is by performing timely backups of the system's files. Even if you're not a system administrator, it is often useful to make copies of things and to move large collections of files from place to place and from device to device.

A command named dd is used to backup the Linux system. dd is a powerful UNIX utility, which is used by the Linux kernel makefiles to make boot images. It can also be used to copy data. Only super user can execute dd command.

Example 1: Backup Entire Hard disk

To backup an entire copy of a hard disk to another hard disk connected to the same system, execute the dd command as shown below. In this dd command example, the UNIX device name of the source hard disk is /dev/hda, and device name of the target hard disk is /dev/hdb.

JKDirectory!

# dd if=/dev/sda of=/dev/sdb

- "if" represents inputfile, and "of" represents output file. So the exact copy of /dev/sda will be available in /dev/sdb.
- If there are any errors, the above command will fail. If you give the parameter "conv=noerror" then it will continue to copy if there are read errors.
- Input file and output file should be mentioned very carefully, if you mention source device in the target and vice versa, you might loss all your data.

In the copy of hard drive to hard drive using dd command given below, sync option allows you to copy everything using synchronized I/O.

# dd if=/dev/sda of=/dev/sdb conv=noerror, sync

Example 2: Create an Image of a Hard Disk

Instead of taking a backup of the hard disk, you can create an image file of the hard disk and save it in other storage devices. There are many advantages to backing up your data to a disk image, one being the ease of use. This method is typically faster than other types of backups, enabling you to quickly restore data following an unexpected catastrophe.

# dd if=/dev/hda of=~/hdadisk.img

The above creates the image of a hard disk /dev/hda.

Example 3:. Restore using Hard Disk Image

To restore a hard disk with the image file of another hard disk, use the following dd command example.

```
# dd if=hdadisk.img of=/dev/hdb
```

The image file hdadisk.img file, is the image of a /dev/hda, so the above command will restore the image of /dev/hda to /dev/hdb.

Example 4: Backup a Partition

You can use the device name of a partition in the input file, and in the output either you can specify your target path or image file as shown in the dd command example below.

```
# dd if=/dev/hda1 of=~/partition1.img
```

Example 5: CDROM Backup

dd command allows you to create an iso file from a source file. So we can insert the CD and enter dd command to create an iso file of a CD content.

```
# dd if=/dev/cdrom of=tgsservice.iso bs=2048
```

JKDirectory!

dd command reads one block of input and process it and writes it into an output file. You can specify the block size for input and output file. In the above dd command example, the parameter “bs” specifies the block size for the both the input and output file. So dd uses 2048bytes as a block size in the above command.

Note: If CD is auto mounted, before creating an iso image using dd command, it’s always good if you unmount the CD device to avoid any unnecessary access to the CD ROM.