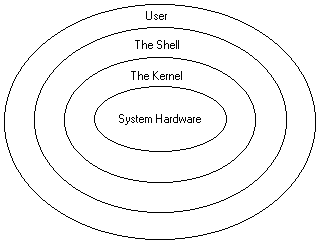
Shell:

Shell is an interface between user and the kernel. Even though there can be only one kernel; A system can have many shells running simultaneously. Whenever a user enters a command through keyboard the shell communicates with the kernel to execute it and then display the output to the user.

Shell Script: The basic concept of a shell script is a list of commands, which are executed in the listed order.

Kernel:

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



Different Types of Shells: csh, ksh, bash, Bourne.

The most commonly used and advanced shell used today is “Bash”.

1. The Bourne Shell (sh)
2. The C Shell (csh or tsch)
3. The Bourne Against Shell (bash)
4. The Korn Shell (ksh)

cat /etc/shells: Will display all the available shells in the Linux.

Default Shell:

To check the default shell type the below command

echo $SHELL

Whenever a user login into the Unix server the default shell initialization script will be executed which is present in the home directory of the login user.

Below are the default initialization scripts with their respective shells:

1. .profile − The Bourne shell ( sh) initialization script
2. .bashrc - The Bash shell ( bash) initialization script
3. .kshrc − The Korn shell ( ksh) initialization script
4. .cshrc − The C shell ( csh) initialization script
5. .rhosts − The remote shell configuration file

set –x :

Is used to debug the script and to check the execution of the script. Enabling debugging with "set -x" also shows each line of execution preceded by a plus sign (+).

Ref: Refer question 9 for example.

<http://www.livefirelabs.com/unix_tip_trick_shell_script/unix_shell_scripting/20-unix-shell-scripting-interview-questions-and-answers-part-1.htm>

shebang:

"#!" is an operator called shebang which directs the script to use the specified shell.

# 🡪 It is called hash.

! 🡪 It is called bang.

If we use #! /bin/sh the script gets directed to the Bourne-shell (sh).

If we use #! /bin/ksh the script gets directed to the Korn-shell (ksh).

Comments: Preceding with **#** will become comment in the shell script.

Shell Script Execution:

**./test.sh –** Executing from the current directory **No space** for dot (.) and slash (/).

**. /home/kiran/test/test.sh** – Executing the script from different path by specifying the relative path. **Space** is required for dot (.) and slash (/).

Debugging the script: **sh –x test.sh or sh –v test.sh both will work.**

**sh –x test.sh** (To execute the script in Bourn shell mode)

**bash –x test.sh** (To execute the script in bash shell mode)

**ksh –x test.sh** (To execute the script in korn shell mode)

**set –x** 🡪 By setting this command in the script level we can see each line of execution preceded by a plus sign (+) in the script log.

echo: is a command which is used to display on a standard output (Monitor) or file.

echo “hello this is Linux”

echo ‘Hello this Unix’

x=UNIX

echo This is $x

Backslash (\): All the Meta characters have a special meaning in UNIX. By putting the backslash (\) the shell will ignore the Meta characters.

x=10 # Assume this is USD 10.

echo “ The ticket price is x=$x”

O/P: The ticket price is x=10

echo “The ticket price is x=**\$**$x” # The backslash will bypass the Meta character $

O/P: The ticket price is x=$10.

Quotes in UNIX:

echo "Hello printing present working directory: `pwd`" # Printing command inside.

O/P: Hello printing present working directory: /home/hirwcourseuser0720

echo 'Hello printing present working directory: `pwd`' # echo with single quotes

O/P: Hello printing present working directory: `pwd`

# Internal command not executed whatever is in the single quotes it will be printed as it is.

read: is a command which takes the input from standard input like keyboard and assign it to the read variable.

read myname # will read the value from keyboard

echo $myname # This will print the value of read variable.

Ex:

echo This is kiran Writing the script

read name1

echo Printing the read value of name1: $name1

read name2

echo Printing the read value of name2: $name2

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Variables\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Local Variables: The scope of the variable is up to the current working shell. The variables are dynamic because we no need to define variable data types.

Example: NAME=’KIRAN’

echo $NAME # This will print the value of Variable Name- Kiran

Now type **bash** # this will create a new bash session in the current session.

echo $NAME # This will give empty result.

Hence the scope of the Shell Variable is up to the current running session.

Environment Variables: The scope of the environmental variables is across the system and available to all the session.

This are divided into two types:

1. System owned Environment Variables (By Default): The values will be available to all the sessions’ child sessions and all the users globally.
2. User defined Environmental Variables (User Specific): Usually this variables will be defined in the shell initialization scripts like .profile, .bashrc etc… The scope of this variables will be up to that session and child session. This variables are not available to other user logins since this variables are user specific.

Ex: NAME=’KIRAN’

export=NAME

echo $NAME # Available to both parent and child shells for that login and other switch users.

printenv: Will have only the environment variables (Both system & user defined Env)

env: Will have all the variables (Local Var + System Env + User defined Env).

|  |  |  |
| --- | --- | --- |
| System Variable | Meaning | To View Variable Value Type |
| BASH\_VERSION | Holds the version of this instance of bash. | echo $BASH\_VERSION |
| HOSTNAME | The name of your computer. | echo $HOSTNAME |
| CDPATH | The search path for the cd command. | echo $CDPATH |
| HISTFILE | The name of the file in which command history is saved. | echo $HISTFILE |
| HISTFILESIZE | The maximum number of lines contained in the history file. | echo $HISTFILESIZE |
| HISTSIZE | The number of commands to remember in the command history. The default value is 500. | echo $HISTSIZE |
| HOME | The home directory of the current user. | echo $HOME |
| PATH | The search path for commands. It is a colon-separated list of directories in which the shell looks for commands. | echo $PATH |
| PS1 | Your prompt settings. | echo $PS1 |
| TMOUT | The default timeout for the read builtin command. Also in an interactive shell, the value is interpreted as the number of seconds to wait for input after issuing the command. If not input provided it will logout user. | echo $TMOUT |
| SHELL | Set path to login shell. | echo $SHELL |
| DISPLAY | Set X display name | echo $DISPLAY export DISPLAY=:0.1 |

UNIX Variables Name convention:

System defined Variable will be in UPPERCASE and User Defined Values (UDV) will be in Lower cases.

System Variable: $PATH, $SHELL, $HOME, $PATH

User Defined Variables: First\_Name, last\_name etc...

Variables are case sensitive just like Unix files.

Read Only Variable: If a variable is declared readonly then the value of the variable cannot be changed.

Ex:

NAME=”Linux”

readonly NAME

echo $NAME

NAME=”Unix” # Will throw readonly variable error.

Note: Once the variable is declared read-only its value cannot be **unset** until the running shell process terminates.

unset: Unset will delete the variables (both local & user specific env ) and its value from the list of variables that the shell is tracking.

Ex: NAME=”Kiran”

echo $NAME # This will print the value of variable as Kiran.

unset NAME # This will delete the variable NAME and it’ s value from the active shell.

Special Variables: $ represents the process id of the shell and to print the pid we can type $$.

$$ : Will print the process id of the login shell terminal. Open a duplicate current running terminal from putty and print $$ it will show the new PID.

The following special variables has special meaning and functionality when used in the shell scripts.

|  |  |
| --- | --- |
| **Variable** | **Description** |
| **$0** | The filename of the current execution script. |
| **$n & ${n}** | Positional parameter with which we can access the parameters that is been passed to script. # (First argument $1 , 2nd Argument $2 …$9). If we want to access from 10 th argument we have to use ${10} |
| **$#** | The number of arguments supplied to a script excluding $0.  ./Test.sh **car bus train** # Total Parameters = 3.  ./Test.sh **“**car bus train**”** # Total Parameters = 1. With in quotes it will take it as 1 Parameter. |
| **$\*** | Contains all of the arguments in a single string, with one space separating them. #It will display the script arguments |
| **$@** | Similar to $\*, but if used in quotes, it effectively quotes each argument and keeps them separate. If any argument contains whitespace, the distinction is important. |
| **$?** | The exit status of the last command executed. #exit **status=0** if successful, exit **status=1** unsuccessful. |
| **$$** | The process number of the current shell. # For shell scripts, this is the process ID under which they are executing. |
| **$!** | The process id of the last background command. |

Difference between $\* and $#:

If the argument list is: a1 a2 "a3 which contains spaces" a4  
then: $1=a1, $2=a2, $3=a3 which contains spaces, $4=a4  
and: $\*=a1 a2 a3 which contains spaces a4  
and: "$@"="a1" "a2" "a3 which contains spaces" "a4"  
  
Only using the form "$@" preserves quoted arguments. If the arguments are being passed from the script directly to some other program, it may make a big difference to the meaning.

Snippet code: <https://www.dartmouth.edu/~rc/classes/ksh/ex7.txt>

Arrays

UNIX doesn’t have data types declaration unlike in other programming Lang’s.

Variable: A shell variable is capable enough to hold a single value. These variables are called scalar variables.

Array Variable: This can hold multiple values at the same time. Arrays provide a method of grouping a set of variables.

Array Declaration:

Name[0]=”Unix”

Name[1]=”Linux”

Name[2]=100

Name[3]=”Science”

Name[4]=10.5

Name[5]=’K’

If we notice the array it has all the data types and which is valid in Unix.

Accessing Array:

${Name[\*]} # First Method

${Name[@]} # Second Method

echo “Printing the values of Array:${Name[\*]}”

Operators in UNIX

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

Below are the various operators available in UNIX.

1. Arithmetic Operators
2. Relational Operators
3. Boolean Operators / Logical Operators
4. String Operators
5. File Test Operators

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

So that is the reason why we have to use **expr** in arithmetic operators. Space is required between **operators** and **operand** and should be quoted with inverted comma (` `) as shown below.

val=`**expr** 2 + 2`

Arithmetic Operators: All the arithmetical calculations are done using long integers.

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

|  |  |  |
| --- | --- | --- |
| **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) | Assigns 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. |

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

Observe that the multiplication operand (\*) has to be escaped (\) to prevent the shell from interpreting it as the filename Meta characters.

Meta characters: have a special meaning to the shell. They can be used as wildcards to specify the name of a file without having to type out the file's full name. Some of the most commonly used met characters are **"\*", "?", "[]", and "-".**

Relational Operators:

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

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

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

Logical / Boolean Operators:

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

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

String Operators

Assume variable **a** holds "abc" and variable **b** holds "efg"

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **=** | Checks if the value of two operands are equal or not; if yes, then the 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 the 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 nonzero length, then it returns true. | [ -n $a ] is not false. |
| **str** | Checks if **str** is not the empty string; if it is empty, then it returns false. | [ $a ] is not false |

File Test Operators

Assume a variable file holds an existing file name "test" the size of which is 100 bytes and has read, write and execute permission on

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| -d file | Checks for a Directory or not. If yes, then the condition becomes true. | [ -d $file ] is not true. |
| -f file | Checks for a File or Not; if yes, then the condition becomes true. | [ -f $file ] is true. |
| -e file | Checks if file exists; is true even if file is a directory but exists. | [ -e $file ] is true. |
| -s file | Checks if file has size greater than 0; if yes, then condition becomes true. | [ -s $file ] is true. |
| **-r file** | Checks if file is **readable**; if yes, then the condition becomes true. | [ -r $file ] is true. |
| **-w file** | Checks if file is **writable**; if yes, then the condition becomes true. | [ -w $file ] is true. |
| **-x file** | Checks if file is **executable**; if yes, then the condition becomes true. | [ -x $file ] is true. |
| **-u file** | Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true. | [ -u $file ] is false. |
| **-b file** | Checks if file is a block special file; if yes, then the condition becomes true. | [ -b $file ] is false. |
| **-c file** | Checks if file is a character special file; if yes, then the condition becomes true. | [ -c $file ] is false. |

Decision Making

If Statement:

Unix Shell supports following forms of **if…else** statement −

* [if...fi statement](https://www.tutorialspoint.com/unix/if-fi-statement.htm) # if block should always end with fi.
* [if...else...fi statement](https://www.tutorialspoint.com/unix/if-else-statement.htm)
* [if...elif...else...fi statement](https://www.tutorialspoint.com/unix/if-elif-statement.htm) #elif is nothing but else if.

If ….fi # Snippet code:

***If*** *is followed by* ***then*** *in the next line or in the same line separated with ;*

* **#!/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**

if…else….fi # Snippet Code.

* **#!/bin/sh**
* **a=10**
* **b=20**
* **if [ $a == $b ]**
* **then # Then should be in the new line or if and then should have ; in between.**
* **echo "a is equal to b"**
* **else**
* **echo "a is not equal to b"**
* **fi**

if...elif...else...fi Snippet Code:

* **#!/bin/sh**
* **a=10**
* **b=20**
* **if [ $a == $b ] # If cond met it will print and come out. If not met it will go for elif**
* **then**
* **echo "a is equal to b"**
* **elif [ $a -gt $b ] # If cond met it will print and exit.If not met it will go for next elif**
* **then**
* **echo "a is greater than b"**
* **elif [ $a -lt $b ] # If cond met it will print and exit. If not met it will go for exit**
* **then**
* **echo "a is less than b"**
* **else**
* **echo "None of the condition met"**
* **fi**

Note: whenever the condition satisfies it will come out of the conditional check block by leaving rest of the conditions.

case...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 the program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

* **#!/bin/bash**
* **# This is case statement in Unix**
* **fruit=$1**
* **case "$fruit" in**
* **"apple")**
* **echo "This is Apple"**
* **;;**
* **"Banana")**
* **echo "This is Banana"**
* **;;**
* **"orange")**
* **echo "This is orange"**
* **;;**
* **"mangos")**
* **echo "This is mangos"**
* **;;**
* **esac**

Loops in Unix:

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

*do.. done is used to indicate the start and end of the execution of the block.*

#!/bin/sh

a=0

while [ $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

For Loop: The **for** loop operates on list of items. It will Travers across the in list of all the items that is been provided.

*do.. done is used indicate the start and end of the execution of the block in Java we use {}*

#!/bin/sh

for var in 0 1 2 3 4 5 6 7 8 9

do # do..done is equal to execution block {} in java.

echo $var

done

Note: If the list of items are enclosed in quotes it will be taken as single Unit.

Syntax 2:

#!/bin/sh

for ((  i = 0 ;  i <= 5;  i++  ))  
do  
  echo "Welcome $i times"  
done

Printing all the files in Directory “/home/hirwcourseuser0720/kirandir/” with wild card character \*

#!/bin/sh

for FILE in /home/hirwcourseuser0720/kirandir/\*

do

echo $FILE

done

Shell functions: A function is a block of code which is used to perform repetitive tasks by passing parameters as functions arguments.

Syntax:

function\_name () {

list of commands

return value

}

**return** will return the value to calling function.

Note: If we use **exit** key word in the function it will exit the method as well as shell session.

Example:

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World"

}

# Invoke your function

Hello # Here there won’t be parenthesis like java

Passing Parameters to Functions:

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World $1 $2" # This parameters always refer to method but not script.

}

# Invoke your function

Hello Zara Ali # Passing parameters to method Hello.

Usage of return keyword:

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World $1 $2"

return 10

}

# Invoke your function

Hello Zara Ali

# Capture value returnd by last command

ret=$? # $? Is also used to capture the exit status of last command.

echo "Return value is $ret"

Nested functions:

#!/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