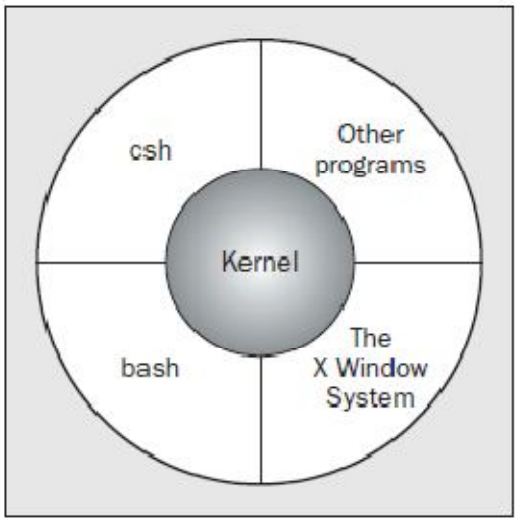
What is a command shell?

A *shell* is a program that acts as the interface between you and the Linux system, enabling you to enter commands for the operating system to execute. In that respect, it resembles the Windows command prompt, but as mentioned earlier, Linux shells are much more powerful.



On Linux, the standard shell that is always installed as /bin/sh is called *bash* (the GNU Bourne-Again Shell), from the GNU suite of tools.
You can check the version of bash you have with the following command:
$ **/bin/bash –version**
Many other shells are available, either free or commercially. The following table offers a brief summary of some of the more common shells available:

Shell Name	A Bit of History
`sh` (Bourne)	The original shell from early versions of UNIX
`csh, tcsh, zsh`	The C shell, and its derivatives, originally created by Bill Joy of Berkeley UNIX fame. The C shell is probably the third most popular type of shell after bash and the Korn shell.
`ksh, pdksh`	The Korn shell and its public domain cousin. Written by David Korn, this is the default shell on many commercial UNIX versions.
`bash`	The Linux staple shell from the GNU project. bash, or Bourne Again SHell, has the advantage that the source code is freely available, and even if it's not currently running on your UNIX system, it has probably been ported to it. bash has many similarities to the Korn shell.

Creating a shell script:
Using any text editor, you need to create a file containing the commands; create a file called first.sh that looks like this:

```
myvar=”Hi there”
echo $myvar
echo “$myvar”
echo ‘$myvar’
echo \$myvar
echo Enter some text
read myvar
```

echo ‘$myvar’ now equals $myvar
exit 0

Making a Script Executable

Now that you have your script file, you can run it in two ways. The simpler way is to invoke the shell with the name of the script file as a parameter:

$ **/bin/sh first**

Or

$ chmod +x first

Then $ first

Environment Variables

Environment Variable	Description
`$HOME`	The home directory of the current user
`$PATH`	A colon-separated list of directories to search for commands
`$PS1`	A command prompt, frequently `$`, but in bash you can use some more complex values; for example, the string `[\u@\h \W]$` is a popular default that tells you the user, machine name, and current directory, as well as providing a $ prompt.
`$PS2`	A secondary prompt, used when prompting for additional input; usually `>`.
`$IFS`	An input field separator. This is a list of characters that are used to separate words when the shell is reading input, usually space, tab, and newline characters.
`$0`	The name of the shell script
`$#`	The number of parameters passed
`$$`	The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example `/tmp/tmpfile_$$`

A sample programme to manipulate the environment variable: test2.sh

```
salutation=”Hello”
echo $salutation
echo “The program $0 is now running”
echo “The second parameter was $2”
echo “The first parameter was $1”
echo “The parameter list was $*“
echo “The user’s home directory is $HOME”
echo “Please enter a new greeting”
read salutation
echo $salutation
echo “The script is now complete”
exit 0
```

If you run this script, you get the following output:

$ **./test2_var foo bar baz**

Hello

The program ./test is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The user's home directory is /home/userid
Please enter a new greeting
Sire
Sire
The script is now complete
$

Conditions

You can also write it like this:
if [condition]
then
...
Fi

String Comparison	Result
`string1 = string2`	True if the strings are equal
`string1 != string2`	True if the strings are not equal
`-n string`	True if the string is not null
`-z string`	True if the string is null (an empty string)
Arithmetic Comparison	**Result**
`expression1 -eq expression2`	True if the expressions are equal
`expression1 -ne expression2`	True if the expressions are not equal
`expression1 -gt expression2`	True if `expression1` is greater than `expression2`
`expression1 -ge expression2`	True if `expression1` is greater than or equal to `expression2`
`expression1 -lt expression2`	True if `expression1` is less than `expression2`
`expression1 -le expression2`	True if `expression1` is less than or equal to `expression2`
`! expression`	True if the expression is false, and vice versa
File Conditional	**Result**
`-d file`	True if the file is a directory
`-e file`	True if the file exists. Note that historically the `-e` option has not been portable, so `-f` is usually used.
`-f file`	True if the file is a regular file
`-g file`	True if `set-group-id` is set on `file`
`-r file`	True if the file is readable
`-s file`	True if the file has nonzero size
`-u file`	True if `set-user-id` is set on `file`
`-w file`	True if the file is writable
`-x file`	True if the file is executable

Control Structures

if

The if statement is very simple: It tests the result of a command and then conditionally executes
a
group of statements:

if condition
then
statements
else
statements
fi

Sample programme: test3.sh

```
echo “Is it morning? Please answer yes or no”
read timeofday
if [ $timeofday = “yes” ]; then
echo “Good morning”
else
echo “Good afternoon”
fi
exit 0
```

elif

Unfortunately, there are several problems with this very simple script. For one thing, it will take any answer except yes as meaning no. You can prevent this by using the elif construct, which allows you to add a second condition to be checked when the else portion of the if is executed.

You can modify the previous script so that it reports an error message if the user types in anything other than yes or no. Do this by replacing the else with elif and then adding another condition:

```
#!/bin/sh
echo “Is it morning? Please answer yes or no”
read timeofday
if [ $timeofday = “yes” ]
then
echo “Good morning”
elif [ $timeofday = “no” ]; then
echo “Good afternoon”
else
echo “Sorry, $timeofday not recognized. Enter yes or no”
exit 1
fi
exit 0
```

Assignment9:

Write a shell script to print the following number pattern (using nested for loop).

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

Assignment10:

Write a shell script to print the prime numbers between n & m. [n & m are user input] (using nested while loop).

Reference:

Beginning Linux Programming by NEIL MATTHEW published by SPD