Chapter 1

Shell Scripting

1.1 Understanding Shell Scripting Core Elements

To help understand the components of the shell scripts, we're going to look at a couple of them. The first one is:

```
1 #!/bin/bash
2 #
3 # This is a script that greets the world
4 # Usage : ./hello
5
6 clear
7 echo Hello, World!
8
9 exit 0
```

The first line of the script `#!/bin/bash` is called the **shebang**. The shebang defines which program is going to execute the script that we're writing, and since we're writing bash scripts here, we put the location of the bash binary executable here on the first line. This is *extremely* important in Linux since many shells such as ksh, tcsh, zsh, etc., share a somewhat common syntax with bash, and as such unexpected errors may occur if a wrong shell is used!

The next few lines (*lines 2-4*) is a comment. It's a good idea to include comments in shell scripts, since bash has a tendency to seem cryptic at first sight, and the comments make the goal of the program easier to understand, and thus the code more readable.

The next few lines are self-explanatory, where the script clears the shell and then echoes (prints) *Hello, World!* to the terminal. Finally, the program exits with an **exit code of 0**. While this line is *not* required here, it's important in certain places. Every program in Linux tells it's parent shell if the operation it tried to perform was successful while exiting via the means of an exit code. An exit code of **0 = success** while anything else means failure. This gives us the opportunity to debug our programs via custom exit codes that indicate what a problem is in the script.

The exit code of the last command can be viewed both in the script or in the shell by:

```
1 $ chmod u+x hello
2 $ ./hello
3 ...
4 Hello, world!
```

```
$ echo $?
0
```

On line 1 we have to give the script executable permissions since otherwise bash won't be able to execute it! Then, we execute our script, which executes with an exit code of 0, which we can verify using `echo $?`.

1.2 Using Variables

1.2.1 Setting and getting the value of a variable

To set a variable to a certain value, we use the syntax: `var=<valueToSet>`, while anywhere that the value of variable is required to be output, we use the variable name with a `$` in front of it, `<something>=$var`. Thus, to assign the value of `var1` to `var2`, the code is:

```
var2=$var1
```

1.2.2 if-else flow control

In bash each `if` code block eventually ends with a corresponding `fi`. In between there can be multiple `elif` checks and finally an `else`, although the last two aren't compulsory.

Conditions are checked using the `test` command. Quite often, we use a shorthand notation for the test command: put the test between square brackets, i.e., `[ <testCriteria ]`. The other way would be to write `test <testCriteria>`. This command has several options that check for different criteria. For example, `test -z` checks to see if a variable is empty, `test -d` checks to see if a folder exists in the current directory with a name same as that of the content of the variable, etc. A list of all possible tests are documented in the manpage for test, accessible with: `man 1 test`.

1.2.3 Example program

Let us consider the following program that prints the first command-line argument if present, or prompts the user for one:

```
#!/bin/bash

if [ -z $1 ]; then
echo "Enter a name:"
read NAME
else
NAME=$1
fi

echo "The name you entered is: $NAME"
```

The command line arguments are represnted by a variable corresponding to their position. So, `$1` is the first argument, `$2` the second and so on. `$0` is the name of the program/script itself!

On line 4 we check if a name has been provided in the form of the first command line argument by checking if the `$1` variable is empty or not. The `test -z` command succeeds if the variable is empty (i.e., if a name hasn't been provided). In that case, it asks for a value. Otherwise it prints the value of the command line argument. The read command stores input from `stdin` and stores it in the variable provided (here, *NAME*).

The output of the script is:

```
$ ./ex2
Enter a name:
Sam
The name you entered is: Sam
$ ./ex2 Dean
The name you entered is: Dean
```

1.3 Using Positional Parameters

Anything that's provided to a script as an argument becomes a positional parameter. For example, in the command `ls -l /etc`, the first positional parameter, i.e., `$1` is the value `-l` while the second parameter `$2` is `/etc`. One **wrong** way to use positional parameters in a script would be:

```
#!/bin/bash

echo parameter 1: $1
echo parameter 2: $2
echo parameter 3: $3
```

This script works as expected when 3 positional arguments are provided:

```
$ ./ex3 a b c
parameter 1: a
parameter 2: b
parameter 3: c
$ ./ex3 a b c d e f
parameter 1: a
parameter 2: b
parameter 3: c
```

In the second case, when more than 3 arguments are given, the ones after the 3^{rd} argument are simply ignored. But when the number of arguments is just 1, the second and the third line are executed anyway:

```
$ ./ex3 a
parameter 1: a
parameter 2:
parameter 3:
```

We should intelligently find out the number of arguments provided to a script and then treat them accordingly. For this, we need a mechanism like a for loop, which iterates over the contents of an item. So, the script becomes:

```
#!/bin/bash
count=1
```

```
3       for i in "$@";
4       do
5           echo "parameter $count : $i"
6           ((count++))
7       done
```

The count variable is used to keep track of the position of the argument being displayed, while the `$@` variable is actually an array (i.e., a variable that stores multiple values of the same type) which contains all of the positional arguments. The final line, `((count++))` increments the value of count and is called an *arithmetic expression* in bash. Such expressions must always occur inside the double brackets to be evaluated. The output of this script is:

```
1           $ ./ex4 a b c d e f
2           parameter 1 : a
3           parameter 2 : b
4           parameter 3 : c
5           parameter 4 : d
6           parameter 5 : e
7           parameter 6 : f
8           $ ./ex4 a
9           parameter 1 : a
10          $ ./ex4
11          $
```

1.4 Understanding if then else

1.5 Understanding for

1.6 Understanding while and until

1.7 Understanding case