Getting Started!

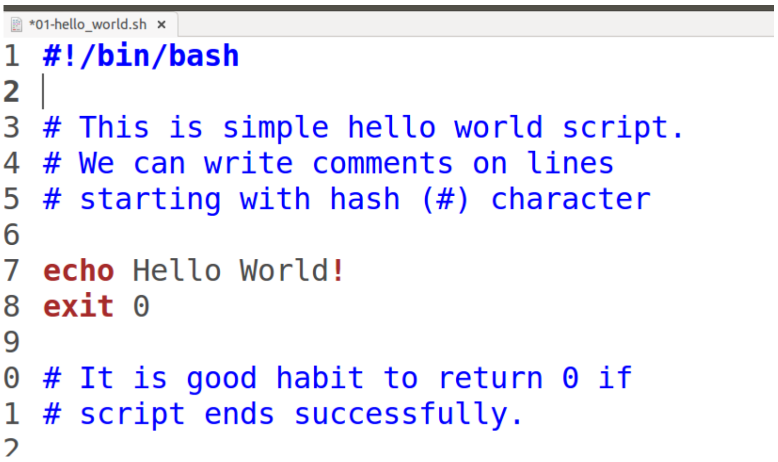
Lets get you started with basic shell scripting. We shall use the scripts in 'Introductory-scripts.tar.gz' in this tutorial.

SECTION 1

Basic components of a script file

For our first shell script, let us write a script which says "Hello World".

Open up <01-hello_world.sh>



```
1 #!/bin/bash
2 
3 # This is simple hello world script.
4 # We can write comments on lines
5 # starting with hash (#) character
6 
7 echo Hello World!
8 exit 0
9 
0 # It is good habit to return 0 if
1 # script ends successfully.
2
```

The script file has only one requirement: That the first line start with the #! directive to specify the interpreter which will process this file.

LINE 1 :

The first line tells Unix that the file is to be executed by /bin/sh. You can specify any file to execute this file. The next program illustrates this in detail. Stick to /bin/sh for now.If you're using GNU/Linux, /bin/sh is normally a symbolic link to bash.

LINES 3-6 :

These lines begin with a special symbol: #. This marks the line as a comment, and it is ignored completely by the shell.

The only exception is when the *very first* line of the file starts with `#!` - as ours does. This is a special directive which Unix treats specially. It means that irrespective of what shell from which you execute this script, the contents of this script file should be interpreted by the Bourne shell (/bin/sh -> /bin/bash).

Similarly, a Perl script may start with the line `#!/usr/bin/perl` to tell your interactive shell that the program which follows should be executed by perl. For Shell programming, we shall stick to `#!/bin/sh.`

LINES 7-8 :

Now we write commands that satisfy the purpose of the script.

Here, our purpose was to print “Hello World!” .Thus line 7 runs a command: `echo`, with two parameters, or arguments - the first is "Hello"; the second is "World!".

You can optionally specify a return code for the script. This is useful, if you run this script and then want to take some action depending on the success of the script. It is done by the exit directive. Here we return 0.

SECTION 2

2.1 Running your script file

A script file is just a file whose first line starts with an interpreter specified. Be it any script – Shell script or a Perl script, a script file should be readable and executable.

To run do the following:

```
-----------------------------------------------------------------

 $ chmod 755  01-hello_world.sh
 $ / 01-hello_world.sh
-----------------------------------------------------------------
```

This prints “Hello World!” on the interactive shell.

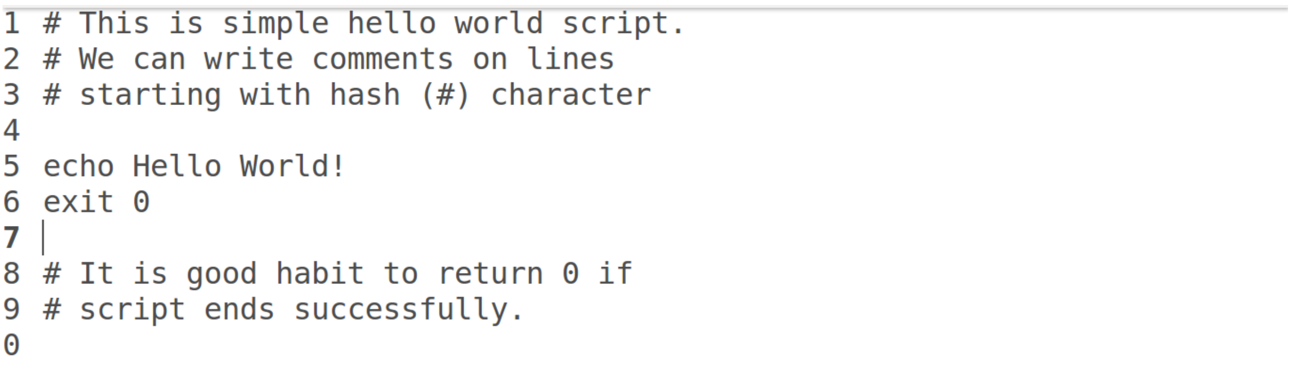
Experiment :

Is it possible to run the shell script without having to specify the interpreter?

Yes!

-- Just that you will have to pass your script file as an argument to the bash comand. Which means it you are calling the interpreter on your own instead of the script asking for it.

For example, open <02-hello_world.txt>



```
# This is simple hello world script.
# We can write comments on lines
# starting with hash (#) character

echo Hello World!
exit 0

# It is good habit to return 0 if
# script ends successfully.

```

This is a normal text file. Notice that the #! is not written here . But it confirms with the shell script syntax.

So you can still run this file if you provide this to bash command:

```
$ bash  02-hello_world.txt
```

Thus, you can create your shell script file and also run it through bash command.

2.2 Summary

Any <scriptfile> with an interpreter specified in the first line (#!<interpreter>) is same as running <interpreter> <scriptfile>

Example : Run <03-grep-interpret.sh>

```
-------------------------------------------------------
$ chmod 755  03-grep-interpret.sh
$ ./ 03-grep-interpret.sh
```

Will produce same output as running

```
$ grep “hello” 03-grep-interpret.sh
-------------------------------------------------------
```

On the same lines,

```
-------------------------------------------------------
$ chmod 755  01-hello_world.sh
$ ./ 01-hello_world.sh
```

Will produce same output as running

```
$ bash 01-hello_world.sh
-------------------------------------------------------
```

SECTION 3

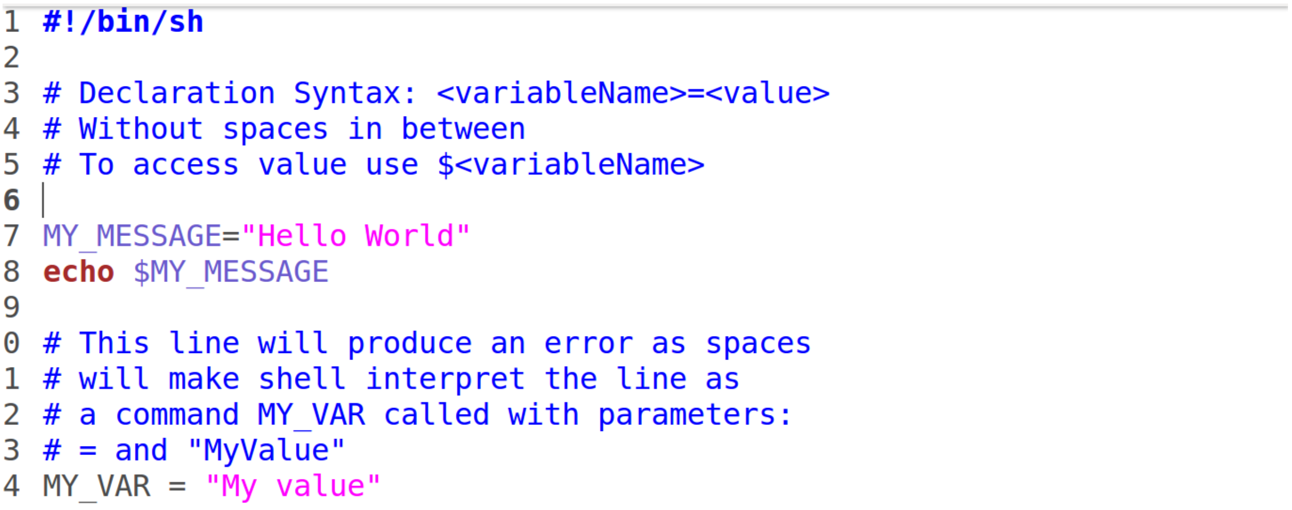
VARIABLES

3.1 Declaring variables

Variables are not typed in bash scripting. You can declare variables with the syntax

variable=value

However, note that there must be no spaces in between. Why?



```
#!/bin/sh

# Declaration Syntax: <variableName>=<value>
# Without spaces in between
# To access value use $<variableName>

MY_MESSAGE="Hello World"
echo $MY_MESSAGE

# This line will produce an error as spaces
# will make shell interpret the line as
# a command MY_VAR called with parameters:
# = and "MyValue"
MY_VAR = "My value"
```

Consider an example:

```
$ grep -i “findme!” filename
```

This is a valid command that you type on the shell. The Shell interprets this command by tokenizing it by spaces as “grep” “-i” “findme!” “filename” and then constructs an argument vector out of it.

Thus, if you declare a variable as

variable<space>=<space>value

It will be interpreted as an argument vector :

[“variable” , “=” , “value”]

Thus, the shell will try to locate a command/file named “variable” and try to execute it!. So you must not use spaces when declaring the variables.

Now run <04-variable-declaration.sh> to understand this!

3.2 Reading Variables

You can read values to your declared variables. In simplest form :

read <varname> <varname> .. <varname>

will read values that you enter on the the terminal to the variables specified in read.

<05-variable-read.sh> illustrates a basic read use.

3.3 Variable scope

Variables in the bourne shell do not have to be declared, as they do in languages like C. But if you try to read an undeclared variable, the result is the empty string. You get no warnings or errors.

First, observe <06-variables-declared-undeclared.sh>:

```
#!/bin/sh
echo "MYVAR is: $MYVAR"
MYVAR="hi there"
echo "MYVAR is: $MYVAR"
```

Now run the script:

```
$ ./06-variables-declared-undeclared.sh
MYVAR is:
MYVAR is: hi there
```

MYVAR hasn't been set to any value, so it's blank. Now we give it a value before running the script. We would now expect this value to be echoed on screen.

Now run:

```
$ MYVAR=hello
$ ./06-variables-declared-undeclared.sh
MYVAR is:
MYVAR is: hi there
```

It's still not been set! What's going on?!

When you call `06-variables-declared-undeclared.sh` from your interactive shell, a new shell is spawned to run the script. This is partly because of the #!/bin/sh line at the start of the script. The new shell doesnt' inherit this variable.

Thus, We need to export the variable for it to be inherited by another program - including a shell script. For this make use of the export directive

Type:

```
$ export MYVAR="hello"
$ ./06-variables-declared-undeclared.sh
MYVAR is: hello
MYVAR is: hi there
```

Thus, we achived inheritance of values to the spawned shell script.

Now look at line 3 of the script: this is changing the value of `MYVAR` to “hi there” . But there is no way that this will be passed back to your interactive shell.

To see this , export `MYVAR` and run the script that will change `MYVAR`. Now if we read `MYVAR`, we still have the old value of “hello” :

```
$ export MYVAR="hello"
$ ./06-variables-declared-undeclared.sh
MYVAR is: hello
MYVAR is: hi there

$ echo $MYVAR
hello
```

Once the shell script exits, its environment is destroyed. Thus, the changed value of `MYVAR=hi there` wont hold after the script exits. So `MYVAR` keeps its value of `hello` within your interactive shell.

In order to receive environment changes back from the script, we must *source* the script - this effectively runs the script within our own interactive shell, instead of spawning another shell to run it.

We can source a script via the "." command:

```
$ MYVAR="hello"
$ echo $MYVAR
hello
$ . ./06-variables-declared-undeclared.sh
MYVAR is: hello
MYVAR is: hi there
$ echo $MYVAR
hi there
```

The change has now made it out into our shell again!

Note that in this case, we don't need to `export MYVAR` as we are not spawning a new shell. However had we exported it, it would still work. Just that it is not necessary.

3.4 Quoting

Quotes are used to specify literal strings. The ones used are SingleQuote(') and Double Quote(“”).

However there is a slight difference - When you write a string with double quotes, there are three special characters :

$

any substring of the string in double quotes that starts with $ is taken as a variable name and its value is replaced.

\

Used to escape the special characters (“ $ \ `)

`

Any thing written between a pair of backticks (` `) is treated as a command and a new shell is spawned to execute the command

Example:

```
 “Hi! This is $BASH”  = Hi! This is /bin/bash
 'Hi! This is $BASH'  = Hi! This is $BASH
```

Run <07-simple-quotes.sh> to find out !

3.5 Special Variables

There are certain useful built-in shell variables. Some of them are described here:

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.
$*	$* is replaced with all command line arguments. Whitespace is *not* preserved, i.e. "one", "two three", "four" would be changed to "one", "two", "three", "four". This variable is not used very often, "$@" is the normal case, because it leaves the arguments unchanged.
$@	"$@" is replaced with all command line arguments, enclosed in quotes, i.e. "one", "two three", "four". Whitespace within an argument is preserved.
$?	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.

Now run <08-special-variables.sh> along with some commandline parameters.

For example:

```
$./08-special-variables.sh Welcome to the jungle!

I was called with 4 parameters
My name is 08-special-variables.sh
My first parameter is Welcome
My second parameter is to
All parameters are Welcome to the jungle!
All parameters are Welcome to the jungle!
```

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.

Now run <09-dollat-dollstar.sh> along with some commandline parameters.

For example:

```
$./09-dollat-dollstar.sh one two three

one two three

There are 1 elements in $*

one

two

three

There are 3 elements in $@
```

Exit Status

The **$?** 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:

```
$./08-special-variables.sh Welcome to the jungle!

I was called with 4 parameters
My name is 08-special-variables.sh
My first parameter is Welcome
My second parameter is to
All parameters are Welcome to the jungle!
All parameters are Welcome to the jungle!

$echo $?
0
```

This prints the default exit status of our script which is 0.

3.6 Variable Substitution and Default Values

Variable substitution is useful if we want to use the value of a variable along with some other text.

Suppose a variable foo has value 'sun' and we want to display value of foo and shine. This is illustrated in <10-param-substn.sh> :

```
--------------------------------------------------

foo=sun
echo $fooshine     # $fooshine is undefined
echo ${foo}shine   # displays the word "sunshine"

--------------------------------------------------
```

One cannot use $fooshine directly as there is not such variable called 'fooshine'. Instead we want to substitute value of foo along with "shine". So we print `{$foo}shine`.

This type of using a variable name within the braces is called Variable/Parameter substition.

Braces have a another, much more powerful use. We can deal with issues of variables being undefined or null (in the shell, there's not much difference between undefined and null).

Using Default Values

Consider the following code <11-default-values-no-braces.sh> which prompts the user for input, but accepts defaults:

```
------------------------------------------------------------------------------------------------------------

#!/bin/sh
echo "What is your name [ `whoami` ] ?" #Note that we call whoami in backticks
read myname
if [ -z "$myname" ]; then # -z tests if myname is a non empty string
  myname=`whoami`
fi
echo "Your name is : $myname"

------------------------------------------------------------------------------------------------------------
```

Now when you run <11-default-values-no-braces.sh> by not entering your name, it will print the value of `whoami`:

```
----------------------------------------------------

./ 11-default-values-no-braces.sh
What is your name [ raghav ] ?

Your name is : raghav
----------------------------------------------------
```

Now when you run <11-default-values-no-braces.sh> by entering your name, it will print the value that you entered:

```
----------------------------------------------------

./ 11-default-values-no-braces.sh
What is your name [ raghav ] ?
Rohit
Your name is : Rohit
----------------------------------------------------
```

This could be done better using a shell variable feature.

By using curly braces and the special ":-" usage, you can specify a default value to use if the variable is unset:

Syntax : *${ <varname> :- <default value> }*

This will test varname for null value and if so, will assign the default value.

Now Consider the following code <12-default-values-braces.sh>

```
-----------------------------------------------------------------------------------------------

/bin/bash
echo -en "What is your name [ `whoami` ] "
read myname
echo "Your name is : ${myname:-`whoami`}"

-----------------------------------------------------------------------------------------------
```

This will take an input myname. If you simply hit Enter myname will be empty and thus the code will substitute it with the defualt value.

The output will be same as that of <11-default-values-no-braces.sh >.

CONCLUSION

Thus, you know now how to write small shell scripts. The usage of variables is very important and this document along with the attached scripts clearly illustrate the ways varialbles can be used and scoped.

Beyond this , you will learn conditional stattements, looping statements and functions. But the underlying concept of quoting and variables remains the same.