# Chapter 2b. scripting  introduction

Shells like **bash** and **Korn** have support for programming constructs that can be saved as **scripts**. These **scripts** in turn then become more **shell** commands. Many Linux commands are **scripts**. **User profile scripts** are run when a user logs on and **init scripts** are run when a **daemon** is stopped or started.

This means that system administrators also need basic knowledge of **scripting** to understand how their servers and their applications are started, updated, upgraded, patched, maintained, configured and removed, and also to understand how a user environment is built.

The goal of this chapter is to give you enough information to be able to read and understand scripts. Not to become a writer of complex scripts.

## 2b.1. prerequisites

You should have read and understood **part - shell expansion** and **part - pipes and commands** before starting this chapter.

## 2b.2. hello world

Just like in every programming course, we start with a simple **hello_world** script. The following script will output **Hello World**.

```
echo Hello World
```

After creating this simple script in **vi** or with **echo**, you'll have to **chmod +x hello_world** to make it executable. And unless you add the scripts directory to your path, you'll have to type the path to the script for the shell to be able to find it.

```
[clim@sop ~]$ echo echo Hello World > hello_world
[clim@sop ~]$ chmod +x hello_world

[clim@sop ~]$ ./hello_world
Hello World

[clim@sop ~]$
```

## 2b.3. she-bang

Let's expand our example a little further by putting **#!/bin/bash** on the first line of the script. The **#!** is called a **she-bang** (sometimes called **sha-bang**), where the **she-bang** is the first two characters of the script.

```
#!/bin/bash

echo Hello World
```

You can never be sure which shell a user is running. A script that works flawlessly in **bash** might not work in **ksh**, **csh**, or **dash**. To instruct a shell to run your script in a certain shell, you can start your script with a **she-bang** followed by the shell it is supposed to run in. This script will run in a bash shell.

```
#!/bin/bash
echo -n hello

echo A bash subshell `echo -n hello`
```

This script will run in a Korn shell (unless **/bin/ksh** is a hard link to **/bin/bash**). The **/etc/shells** file contains a list of shells on your system.

```
#!/bin/ksh
echo -n hello

echo a Korn subshell `echo -n hello`
```

## 2b.4. comment

Let's expand our example a little further by adding comment lines.

```
#!/bin/bash

#

# Hello World Script

#

echo Hello World
```

## 2b.5. variables

Here is a simple example of a variable inside a script.

```
#!/bin/bash

#

# simple variable in script

#

var1=4

echo var1 = $var1
```

Scripts can contain variables, but since scripts are run in their own shell, the variables do not survive the end of the script.

```
[clim@sop ~]$ echo $var1


[clim@sop ~]$ ./vars
var1 = 4

[clim@sop ~]$ echo $var1
```

## 2b.6. sourcing a script

Luckily, you can force a script to run in the same shell; this is called **sourcing** a script.

```
[clim@sop ~]$ source ./vars
var1 = 4

[clim@sop ~]$ echo $var1 4

[clim@sop ~]$
```

The above is identical to the below.

```
[clim@sop ~]$ . ./vars
var1 = 4

[clim@sop ~]$ echo $var1 4

[clim@sop ~]$
```

## 2b.7. troubleshooting a script

Another way to run a script in a separate shell is by typing **bash** with the name of the script as a parameter.

```
clim@sop~/test$ bash runme 42
```

Expanding this to **bash -x** allows you to see the commands that the shell is executing (after shell expansion).

```
clim@sop~/test$ bash -x runme

+ var4=42

+ echo 42

42

clim@sop~/test$ cat runme

# the runme script
var4=42

echo $var4
clim@sop~/test$
```

Notice the absence of the commented (#) line, and the replacement of the variable before execution of **echo**.

## 2b.8. prevent setuid root spoofing

Some user may try to perform **setuid** based script **root spoofing**. This is a rare but possible attack. To improve script security and to avoid interpreter spoofing, you need to add **--** after the **#!/bin/bash**, which disables further option processing so the shell will not accept any options.

```
#!/bin/bash -
or

#!/bin/bash --
```

Any arguments after the **--** are treated as filenames and arguments. An argument of - is equivalent to --.

## 2b.9. practice: introduction to scripting

0. Give each script a different name, keep them for later!
1. Write a script that outputs the name of a city.
2. Make sure the script runs in the bash shell.
3. Make sure the script runs in the Korn shell.
4. Create a script that defines two variables, and outputs their value.
5. The previous script does not influence your current shell (the variables do not exist outside of the script). Now run the script so that it influences your current shell.
6. Is there a shorter way to **source** the script ?
7. Comment your scripts so that you know what they are doing.