Session #2

Shell script

- Variable
- Wildcard
- Escape characters
- Loop
- Test
- Case
- External Program
- Function

Variable

Just about every programming language in existence has the concept of variables - a symbolic name for a chunk of memory to which we can assign values, read and manipulate its contents.

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.

```
MY_MESSAGE="Hello World"
echo $MY_MESSAGE
```

Variable

Example:

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

Variable

Example:

```
$ ./myvar2.sh
MYVAR is:
MYVAR is: hi there
```

If your variable is not set, it will be interpreted as an empty string.

Variable

There are a set of variables which are set for you already, and most of these cannot have values assigned to them.

These can contain useful information, which can be used by the script to know about the environment in which it is running.

Variable

The first set of variables we will look at are **$0** .. **$9** and **$#**.

The variable **$0** is the basename of the program as it was called.

$1 .. $9 are the first 9 additional parameters the script was called with.

The variable **$@** is all parameters **$1** .. whatever. The variable **$***, is similar, but does not preserve any whitespace, and quoting, so "File with spaces" becomes "File" "with" "spaces". As a general rule, use **$@** and avoid **$***.

$# is the number of parameters the script was called with.

Variable

```
#!/bin/sh
echo "I was called with $# parameters"
echo "My name is $0"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

Variable

```
$ /home/yoann/var3.sh
I was called with 0 parameters
My name is /home/yoann/var3.sh
My first parameter is
My second parameter is
All parameters are
```

Variable

```
$ ./var3.sh hello world earth
I was called with 3 parameters
My name is ./var3.sh
My first parameter is hello
My second parameter is world
All parameters are hello world earth
```

Variable

```
#!/bin/sh
while [ "$#" -gt "0" ]
do
  echo "\$1 is $1"
done
```

You can check your parameters before starting your script.

Variable

curly brackets usage:

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

That's not all, though - these fancy brackets 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).

Variable

There is another syntax, ":=", which sets the variable to the default if it is undefined:

```
echo "Your name is : ${myname:=Yoann Cerda}"
```

Wildcard

Wildcards are really nothing new if you have used Unix at all before.

It is not necessarily obvious how they are useful in shell scripts though. This section is really just to get the old grey cells thinking how things look when you're in a shell script - predicting what the effect of using different syntaxes are.

Wildcard

Think first how you would copy all the files from /tmp/a into /tmp/b. All the .txt files? All the .html files?

Hopefully you will have come up with:

```
$ cp /tmp/a/* /tmp/b/
```

```
$ cp /tmp/a/*.txt /tmp/b/
```

```
$ cp /tmp/a/*.html /tmp/b/
```

Escape characters

Certain characters are significant to the shell; we have seen, for example, that the use of double quotes (") characters affect how spaces and TAB characters are treated, for example:

```
$ echo Hello       World
Hello World
$ echo "Hello       World"
Hello     World
```

To escape special characters, we use \

Escape characters

Certain characters are significant to the shell; we have seen, for example, that the use of double quotes (") characters affect how spaces and TAB characters are treated, for example:

```
$ echo Hello       World
Hello World
$ echo "Hello       World"
Hello     World
```

To escape special characters, we use \

Escape characters

How would you print:

Hello “world”

on your terminal?

Loop

Most languages have the concept of loops: If we want to repeat a task twenty times, we don't want to have to type in the code twenty times, with maybe a slight change each time.

As a result, we have for and while loops in the Bourne shell. This is somewhat fewer features than other languages, but nobody claimed that shell programming has the power of C.

Loop

For loop:

```
#!/bin/sh
for i in 1 2 3 4 5
do
  echo "Looping ... number $i"
done
```

Loop

While loop:

```
while [ condition ]
do
        command1
        command2
        commandN
done
```

Loop

While loop:

```
#!/bin/sh
INPUT_STRING=hello
while [ "$INPUT_STRING" != "bye" ]
do
  echo "Please type something in (bye to quit)"
  read INPUT_STRING
  echo "You typed: $INPUT_STRING"
done
```

Test

Test is used by virtually every shell script written. It may not seem that way, because test is not often called directly. test is more frequently called as [. [is a symbolic link to test, just to make shell programs more readable.

Test

```
if SPACE [ SPACE "$foo" SPACE = SPACE "bar" SPACE ]
```

The syntax of test conditions is extremely strict. If you forget one SPACE, your shell will not be able to understand it and your script will simply crash.

Test

```
if [ ... ]
then
  # if-code
else
  # else-code
fi
```

Test

```
if  [ something ]; then
 echo "Something"
 elif [ something_else ]; then
   echo "Something else"
 else
   echo "None of the above"
fi
```

Primary	Meaning
[`-a FILE`]	True if `FILE` exists.
[`-b FILE`]	True if `FILE` exists and is a block-special file.
[`-c FILE`]	True if `FILE` exists and is a character-special file.
[`-d FILE`]	True if `FILE` exists and is a directory.
[`-e FILE`]	True if `FILE` exists.
[`-f FILE`]	True if `FILE` exists and is a regular file.
[`-g FILE`]	True if `FILE` exists and its SGID bit is set.
[`-h FILE`]	True if `FILE` exists and is a symbolic link.
[`-k FILE`]	True if `FILE` exists and its sticky bit is set.
[`-p FILE`]	True if `FILE` exists and is a named pipe (FIFO).
[`-r FILE`]	True if `FILE` exists and is readable.
[`-s FILE`]	True if `FILE` exists and has a size greater than zero.
[`-t FD`]	True if file descriptor `FD` is open and refers to a terminal.
[`-u FILE`]	True if `FILE` exists and its SUID (set user ID) bit is set.
[`-w FILE`]	True if `FILE` exists and is writable.
[`-x FILE`]	True if `FILE` exists and is executable.
[`-O FILE`]	True if `FILE` exists and is owned by the effective user ID.
[`-G FILE`]	True if `FILE` exists and is owned by the effective group ID.
[`-L FILE`]	True if `FILE` exists and is a symbolic link.
[`-N FILE`]	True if `FILE` exists and has been modified since it was last read.
[`-S FILE`]	True if `FILE` exists and is a socket.
[`FILE1 -nt FILE2`]	True if `FILE1` has been changed more recently than `FILE2`, or if `FILE1` exists and `FILE2` does not.
[`FILE1 -ot FILE2`]	True if `FILE1` is older than `FILE2`, or is `FILE2` exists and `FILE1` does not.
[`FILE1 -ef FILE2`]	True if `FILE1` and `FILE2` refer to the same device and inode numbers.
[`-o` OPTIONNAME]	True if shell option "OPTIONNAME" is enabled.
[`-z` STRING]	True of the length if "STRING" is zero.
[`-n` STRING] or [STRING]	True if the length of "STRING" is non-zero.
[STRING1 == STRING2]	True if the strings are equal. "=" may be used instead of "==" for strict POSIX compliance.
[STRING1 != STRING2]	True if the strings are not equal.
[STRING1 < STRING2]	True if "STRING1" sorts before "STRING2" lexicographically in the current locale.
[STRING1 > STRING2]	True if "STRING1" sorts after "STRING2" lexicographically in the current locale.
[ARG1 OP ARG2]	"OP" is one of `-eq`, `-ne`, `-lt`, `-le`, `-gt` or `-ge`. These arithmetic binary operators return true if "ARG1" is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to "ARG2", respectively. "ARG1" and "ARG2" are integers.

Case

The case statement saves going through a whole set of if .. then .. else statements. Its syntax is really quite simple:

```
echo "Please talk to me ..."
while :
do
  read INPUT_STRING
  case $INPUT_STRING in
          hello)
                  echo "Hello yourself!"
                  ;;
          bye)
                  echo "See you again!"
                  break
                  ;;
          *)
                  echo "Sorry, I don't understand"
                  ;;
  esac
done
echo
echo "That's all folks!"
```

External Program

External programs are often used within shell scripts; there are a few builtin commands (echo, which, and test are commonly built in), but many useful commands are actually Unix utilities, such as tr, grep, expr and cut.

```
$ export myvar=`whoami`
$ echo $myvar
$ yoann
```

Function

A function may return a value in one of four different ways:

- Change the state of a variable or variables
- Use the exit command to end the shell script
- Use the return command to end the function, and return the supplied value to the calling section of the shell script
- echo output to stdout, which will be caught by the caller just as c=`expr $a + $b` is caught

This is rather like C, in that exit stops the program, and return returns control to the caller. The difference is that a shell function cannot change its parameters, though it can change global parameters.

Function

```
add_a_user()
{
  USER=$1
  PASSWORD=$2
  shift; shift;
  # Having shifted twice, the rest is now comments ...
  COMMENTS=$@
  echo "Adding user $USER ..."
  echo useradd -c "$COMMENTS" $USER
  echo passwd $USER $PASSWORD
  echo "Added user $USER ($COMMENTS) with pass $PASSWORD"
}
```