# Shell Scripting: Advanced (1)

## Variables

- all programs have them, Shell languages are no different

**to create a variable:**

- decide on its name, then. set its value using the = operator (no spaces around the =)

```
GREETING=”Hello World!”
```

**to use a variable:**

- use the dollar sign followed by curly braces

```
COMMAND "${VAR}"

e.g.
echo "${GREETING}"
```

if you want your variable to exist in the programs you start as an environmental variable:

```
export GREETING
```

to get rid of variable:

```
unset GREETING
```

**Always stick variables inside quotes! - shellcheck will complain otherwise**

## why double quotes?

- variables in Shell languages tend to act more like macro variables
  - **there is no penalty for using one that isn't defined, e.g.;**

```
NAME='Tom'
unset NAME
echo "Hello, '${NAME}'"
Hello, ''
```

- it will put an empty string in there

to avoid this:

use the set -o nounset flag

```
set -o nounset
```

- another use of curly braces, is that if after the variable you stick a colon question mark (:?), then if the variable isnt set, it wil produce an error:

```
echo "${NAME:? variable 1 passed to program}"
```

---

**What variables do you ALWAYS have?**

# Standard Variables

**${0}** - name of the script

**${1}, ${2}, ${3}...** - are the arguments passed to your script

**${#}** - the number of arguments passed to the script

**${@}** and **${*}** **-** all of the arguments as one big string, **will almost always want to use ${*}**

# Control Flow

if statements and for loops are available with **globbing**

- Globbing - the process of pattern matching or wildcard matching of strings, filenames, or other data
- for some unknown reason control flow commands in shell end with their name backwards:

```
if == fi
```

- except for for which ends with done

```
for file in *py; do
        python "${file}"
done
```

## if statements

```
if test -x myscript,sh; then
        ./myscript,sh
fi
```

## other loops

- only have for loop keyword, but you can do lots with it
- for will run over everything you pass it:

```
for n in 1 2 3 4 5; do
        echo -n "${n} "
done

//OUTPUT
1 2 3 4 5
```

The `seq` **command gives you a sequence of numbers:**

```
seq 5
1 2 3 4 5

for n in $(seq5); do
        echo -n "${n} "
done


OUTPUT:

1 2 3 4 5
```

- can also ask seq to seperate things with a comma: `seq -s, 5` : (outputs 1,2,3,4,5)
- there is a strange variable called the in field seperator: `IFS`
    - **The IFS variable** - which stands for **I**nternal **F**ield **S**eparator - controls what Bash calls *word splitting*. When set to a string, each character in the string is considered by Bash to separate words. This governs how bash will iterate through a sequence. For example, this script:

```
#!/bin/bash
IFS=$' '
items="a b c"
for x in $items; do
    echo "$x"
done

IFS=$'\n'
for y in $items; do
    echo "$y"
done



Would print this:

a
b
c
a b c
```

In the first for loop, IFS is set to `$' '`. (The `$'...'` syntax creates a string, with backslash-escaped characters replaced with special characters - like "\t" for tab and "\n" for newline.) Within the for loops, x and y are set to whatever bash considers a "word" in the original sequence. For the first loop, IFS is a space, meaning that words are separated by a space character. For the second loop, "words" are separated by a *newline*, which means bash considers the whole value

of "items" as a single word. If IFS is more than one character, splitting will be done on *any* of those characters.

Got all that? The next question is, why are we setting IFS to a string consisting of a tab character and a newline? Because it gives us better behavior when iterating over a loop. By "better", I mean "much less likely to cause surprising and confusing bugs". This is apparent in working with bash arrays:

```
IFS=','
for n in $(seq -s, 5); do
        echo -n "${n}"
done

OUTPUT
1 2 3 4 5
```

## case statements

- In the below case you can do pattern matching

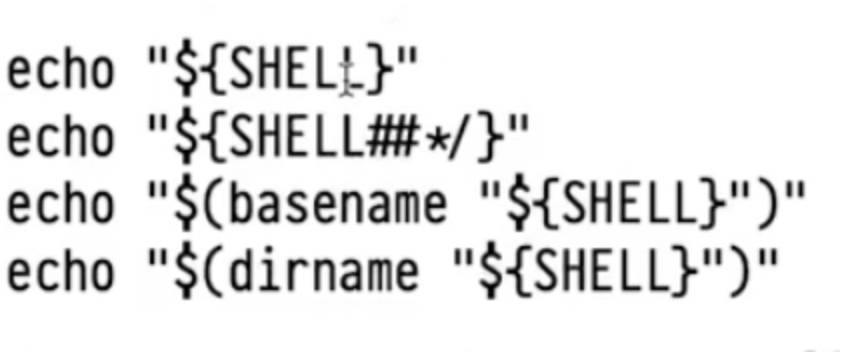
```
#remove everything up to the last / from ${SHELL}
case "${SHELL##*/}" in   #shell variable usually has path to you
        bash) echo "im using bash" ;; // if it says bash then do
        zsh) echo "im a zsh user" ;;
        *) echo "something else" ;;
esac
```

- so the syntax is the pattern you want to match, closing bracket, then end the line with 2 semicolons
- end the case with `esac` line (case backwards)

**But what is the ##* in this: ${SHELL##*/}??? :**

# Base name and Dirname

- using “${VAR##*/}” is a useful shell expansion trick that removes **EVERYTHING** up to the last patern
  - it will try match the pattern as best it can, in this case */... it has matched 0 or more things ( * ) followed by a slash ( / ) and it will try and match it for as long as possible and returns only the filename
  - e.g. using it with the `SHELL` variable, which usually contains something like `usr/bin/bash/` it will remove the usr and bin parts leaving only `bash` which is the name of the file neatly presented without unnecessary info
- instead of remembering this, use `$(basename “${shell}”)` to get the same info
  - it takes the basename of whatever path you pass it



```
echo "${SHELL}"
echo "${SHELL##*/}"
echo "$(basename "${SHELL}")"
echo "$(dirname "${SHELL}")"
```

*All of the above commands are roughly equivalent*

### another basename trick

if you give it a second argument it will remove that **suffix from the basename** e.g.:

- the below code runs through all the jpg files in the folder
- the `convert` command does image conversion

```
for f in *.jpg; do
        convert "${f}" "$(basename "${f}" .jpg).png"
```

```
done
```

- this converts a jpg file into png

# Pipelining in Shellscripting

- a great part of shell scripting is pipelines
- something that makes shellscripting so powerful
- it is often useful to build commands out of chains of other commands
  - **i.e. take the output of one command and feed it into the input of another**
- the `ps` command lists all the processes on the computer

How would you find out how many processes *Firefox* is using?

- you can combine `ps` with `grep` to do some searching:

```
ps -A | grep -i firefox
```

- so i find all the processes with `ps` and then search for which ones match firefox by using `grep`

Example output:

```
43172   ??   SpU    0:10.69   /usr/local/bin/firefox
59551   ??   Sp     0:00.06   /usr/local/lib/firefox/firefox   -contentproc   -appDir
 7023   ??   SpU    0:06.10   /usr/local/lib/firefox/firefox   -contentproc   {a032331
59478   ??   SpU    0:00.21   /usr/local/lib/firefox/firefox   -contentproc   {3cd651d
47320   ??   SpU    0:00.60   /usr/local/lib/firefox/firefox   -contentproc   {50d5261
26734   ??   SpU    0:00.18   /usr/local/lib/firefox/firefox   -contentproc   {68aa722
  308   ??   SpU    0:00.16   /usr/local/lib/firefox/firefox   -contentproc   {bd6ff5f
42479   ??   SpU    0:00.14   /usr/local/lib/firefox/firefox   -contentproc   {d874750
45572   ??   Rp/2   0:00.00   grep                             -i             firefox
```

- the bottom line is unnecessary as it isnt really firefox but grep using the ford firefox in its own process therefore it is included in the output

- there is also a bunch of other necessary info in the output that is hard to understand

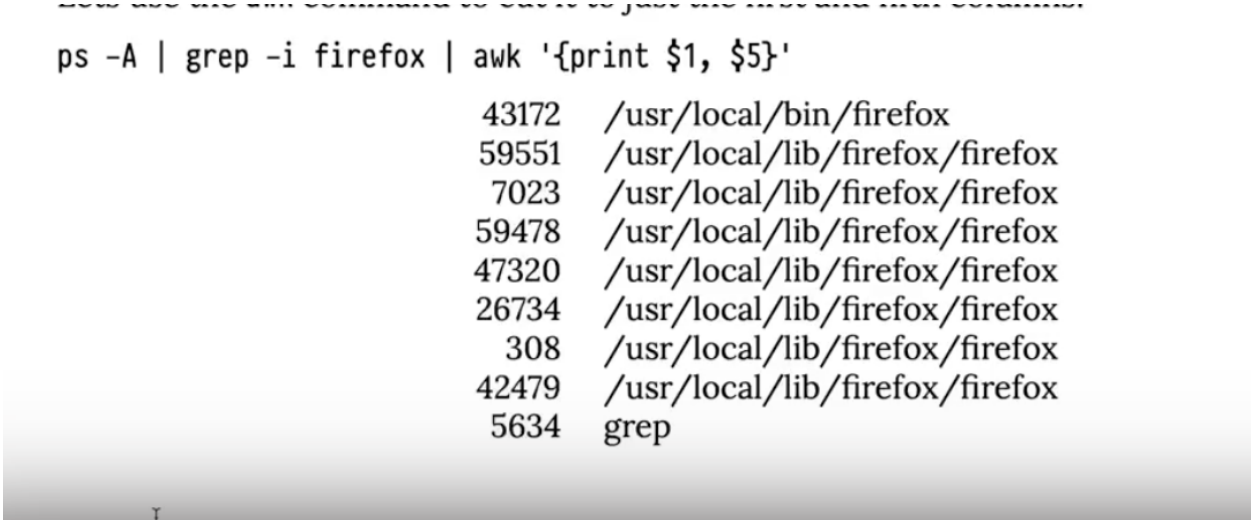
## Reducing info using awk

- to reduce this info use the `awk` **command**
- `awk` is good at splitting things up into different components
  - by default it assumes things are seperated by single spaces

the command:

```
ps -A | grep -i firefox | awk '{print $1, $5}'
```

- will print the first field and the 5th field seperated by a space:




- this cuts the output to just the first and fifth columns

#### Removing the unnecessary grep line

- we know that grep will always be the last thing as it is the command we have just started therefore it will have the last *process id*
- you could use the head command:

```
ps -A | grep -i firefox | awk '{print $1, $5}' | head -n -1
```


**The original point of this example was to count the number of processes not list them**

- we can finally pipe it into wc with the -l (line count flag), as each process will be a single line

```
ps -A | grep -i firefox | awk '{print $1, $5}' | head -n -1 | wc


OUTPUT:
8
```

# Other piping techniques

- The " `|` " pipe copies *standard output* to *standard input*
- the " `>` " pipe copies *standard output* to a named file:

```
ps -A >processes.txt
```

- the “ `>>` “ pipe appends *standard output* to a named file
  - usually when you pipe output into a named file it will erase the file to begin with
- the “ `<` “ pipe reads a file **into** *standard input* which can then be used as input for another command

```
grep firefox <processes.txt
```

- the “ `<<<` “ pipe takes a **string** and places the string onto *standard input*
- you can also copy and merge streams if you know their file descriptors
  - e.g. appending `2>&1` to a command will run it with *standard error* **merged** into *standard output*