SECTION 6

PROCESSING OPTIONS & READING FILES

SECTION CHEAT SHEET

WHILE LOOPS:

while loops run a set of commands ***while*** a certain condition is true, hence their name.

while loops will continue to run until either:

1. The condition command that they're provided with becomes false (i.e. returns a non-zero exit code)
2. The loop is interrupted.

Syntax for the while loop:

```
while condition; do
  commands...
done
```

Example Script:

```
#!/bin/bash

read -p "Enter your number: " num

while [ "$num" -gt 10 ]; do
  echo "$num"
  num=$(( "$num" - 1 ))
done
```

Remember!

You must ensure that the condition you provide the while loop does become false at some point to avoid infinite loops!

HANDLING COMMAND LINE OPTIONS:

The getopts command enables bash to **get** the the **opt**ion**s** provided to the script on the command line.

However, getopts does not get all the options at once; it only gets the very next option on the command line each time it is run.

Therefore, the getopts command is often used as part of a while loop, to ensure that all command line options are processed.

Syntax for the getopts command

```
getopts "optstring" variable
```

You can call variable whatever you like. However, it is conventionally called “opt” because it stores the most recent **opt**ion that getopts has found.

Syntax for optstrings

Any single letter we place in the optstring is considered as its own option.

getopts can only process one-letter options (long-form options such as --all are not supported.)

For Example:

if we wanted to accepted the options "-A" and "-b", we could write:

```
getopts "Ab" variable
```

Notice how **options are case-sensitive.**

Option Arguments

Sometimes, options can accept arguments of their own. For example, let’s say we had the command:

```
ourscript -A 10
```

In order to allow the -A option to accept an argument, such as “10”, we would need to place a colon (:) after the letter “A” in the optstring, like so:

```
getopts "A:b" variable
```

Whatever argument that is provided with an option is stored in the $OPTARG shell variable.

So, if we ran the command ourscript -A 10, the $OPTARG variable would store the value of 10 when the getopts command processed the -A option

Practical examples

The getopts command is often used in conjunction with a while loop so that we ensure that each option on the command line gets processed.

In order to allow the script to perform different actions based on the options that are provided, we often also put a case statement inside the while loop, with one case for each option.

Syntax for using the getopts command with a while loop and case statement:

```
while getopts "A:b" variable; do
        case "$variable" in
                A)
                   commands
                   ;;
                b)
                   commands
                   ;;
                \?)
                   commands
                   ;;
        esac
done
```

When an unexpected option is provided to the getopts command, it stores a literal question mark inside the variable.

Therefore, it is good practice to create a \? case to respond to any invalid options. The backslash (\) ensures that the ? is interpreted literally, and not as a special globbing pattern character.

Example Script:

```
#!/bin/bash

while getopts "c:f:" opt; do
      case "$opt" in
        c)  # convert from celsius to farenheit
            result=$(echo "scale=2; ($OPTARG * (9 / 5)) + 32" | bc)
            ;;
        f)  # convert from fahrenheit to celsius
             result=$(echo "scale=2; ($OPTARG - 32) * (5/9)" | bc)
             ;;
        \?)
             Echo "Invalid option provided"
             ;;
    esac
    echo "$result"
done
```

READ-WHILE LOOPS:

Read-while loops are simply while loops that use the read command as their test command

They are used to read lines of output one by one, and do something for each line.

A read-while loop can be used to iterate over the contents of files, or over the output of a command (or pipeline)

ITERATING OVER THE CONTENTS OF FILES

```
while read line; do
  commands...
done < file
```

Example Script: Iterating over a file line by line, and printing each line out

```
#!/bin/bash


while read line; do
  echo "$line"
done < file1.txt
```

ITERATING OVER THE OUTPUT FROM COMMANDS

This is achieved using a technique known as **process substitution**. Process substitution simply allows us to treat the output of a command (or commands) as a file.

Syntax for process substitution

```
<(command) # You can run one command...
<(command1 | command2 | ... | commandN) #... Or an entire pipeline
```

We can then simple read the output of the command into the while loop as if it was a file:

```
while read line; do
  commands...
done < <(command)
```

Example Script: Iterating over each line of output from a command

```
#!/bin/bash

while read line; do
  echo "$line"
done < <(ls $HOME)
```