Operating Systems Writeup

SETU - South East Technological University
Imbriani Paolo - W20114452
Professor Micheal McMahon

March 21, 2025

Contents

1 Practical 1 - Commands

Taking a selection of Windows CLI commands from those given below, use the online help to examine the various options and arguments, and try them out.
You're required carefully to write two A4 pages (Times 12 point or equivalent size) detailing your experiments with different options for between six and ten different commands. To get the online help for a command, type command /?
e.g.
dir /?
prompt
mkdir
color
title
tree
type
ver
print
xcopy
Type help at the windows command line prompt to see some more instructions

- Prompt - The prompt command is used to customize the text that appears before the cursor in the command prompt.

```
prompt MyPrompt$G
```

  This changes the prompt to MyPrompt>. The $G represents the > symbol.

- Mkdir - The mkdir command is used to create a new directory.

```
mkdir MyDirectory
```

  This creates a new directory called MyDirectory. To create a folder inside another folder:

```
mkdir MyDirectory\MySubDirectory
```

- Color - The color command is used to change the color of the text and background in the command prompt.

```
color 0A
```

  This sets a black background (0) with green text (A). To reset to default:

```
color
```

  To see all the available colors:

```
color /?
```

- Title - The title command is used to change the title of the command prompt window.

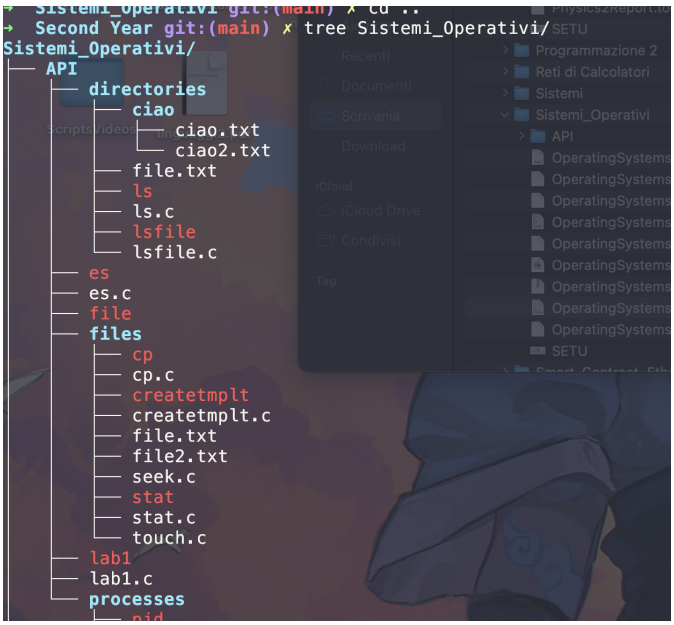
```
title MyTitle
```

  This changes the title of the command prompt window to MyTitle.

- Tree - The tree command is used to display a graphical representation of the directory structure.

```
tree
```

And it will output something like this:



Displays a simple tree structure of folders in the current directory. To include all files in the display:

```
tree /f
```

The /F option lists all files along with the folder structure.

- Type - The type command is used to display the contents of a text file.

```
type MyFile.txt
```

This displays the contents of the file MyFile.txt. Useful for quickly viewing small text files without opening them.

- Ver - The ver command is used to display the version of the operating system.

```
ver
```

This displays the version of the operating system.

What's the purpose of the first line - @ECHO OFF? Remove it and see the effect

```
@ECHO OFF
ECHO Please insert a USB memory stick
PAUSE
COPY *.txt I:\
ECHO BACKUP COMPLETE
```

- @ECHO OFF → Hides command execution lines for cleaner output.

- ECHO → Displays messages on the screen.
- PAUSE → Waits for the user to press a key before continuing.
- COPY *.txt I: → Copies all .txt files from the current folder to the USB drive (assuming it's drive I:).
- ECHO BACKUP COMPLETE → Displays a completion message.

The first line, @ECHO OFF, is used to prevent the command prompt from displaying each command as it executes.

```
C:\Users\YourName\Desktop> ECHO Please insert a USB memory stick
Please insert a USB memory stick

C:\Users\YourName\Desktop> PAUSE
Press any key to continue . . .

C:\Users\YourName\Desktop> COPY *.txt I:\
3 file(s) copied.

C:\Users\YourName\Desktop> ECHO BACKUP COMPLETE
BACKUP COMPLETE
```

2 Practical 2 - Powershell

2.1 Test the above using a directory that you created last week.

```
user@machine:~/data/os1$

    Directory: /temp

UnixMode         User Group         LastWriteTime      Size
--------         ---- -----         -------------      ----
drwxr-xr-x       root root        2/12/2025 10:10        39
drwxr-xr-x       root root        2/12/2025 10:10        50
drwxr-xr-x       root root        2/12/2025 10:10        58
drwxr-xr-x       root root        2/12/2025 10:10        42
drwxr-xr-x       root root        2/12/2025 10:10        41
drwxr-xr-x       root root        2/12/2025 10:10        23
-rwxr-xr-x       root root        2/12/2025 10:18       502
-rwxr-xr-x       root root        2/12/2025 10:18        81
-rwxr-xr-x       root root        2/12/2025 10:18       123
-rw-r--r--       root root        2/12/2025 10:18       566
-rwxr-xr-x       root root        2/12/2025 10:18       221
-rwxr-xr-x       root root        2/12/2025 10:18       199
-rw-r--r--       root root        2/12/2025 10:18         0
-rwxr-xr-x       root root        2/12/2025 10:18      1983
-rwxr-xr-x       root root        2/12/2025 10:18       180
-rwxr-xr-x       root root        2/12/2025 10:18       140
-rw-r--r--       root root        2/12/2025 10:18       620
-rwxr-xr-x       root root        2/12/2025 10:18       186
-rwxr-xr-x       root root        2/12/2025 10:18       213
-rwxr-xr-x       root root        2/12/2025 10:18        45
-rwxr-xr-x       root root        2/12/2025 10:18        69
-rwxr-xr-x       root root        2/12/2025 10:18        17
-rwxr-xr-x       root root        2/12/2025 10:18      2079
-rwxr-xr-x       root root        2/12/2025 10:18      2043
-rwxr-xr-x       root root        2/12/2025 10:18      1126
-rwxr-xr-x       root root        2/12/2025 10:18       320
-rwxr-xr-x       root root        2/12/2025 10:18      1019
-rwxr-xr-x       root root        2/12/2025 10:18       231
-rwxr-xr-x       root root        2/12/2025 10:18       277
```

Using the directory was created last week, we can use the Get-ChildItem command to list the contents of the directory. using the following list of commands:

```
$mylist = dir
$mylist
$mylist[0].Name
$mylist[0].length
$mylist[0].Mode
$mylist[0].LastWriteTime
```

Exercise of bash scripting in Powershell:

- Simply record what happens when you run this script.
- Find out how to run scripts if they're in a directory other than the working directory.
- Study the following script and see if you can figure out what it'll do. Now type it into a file called game.ps1 (You can use copy/paste in places to reduce the labour.) Run it and see if your predictions are true.

```
################################################################
################################################################
##
## The three pathetic knock-knock jokes program!
## Date: 26/09/17
## For: BSc (Hons) Computer Forensics and Security
##
################################################################
################################################################

#############################
## initialisation section
#############################
$userReply = ""
#############################
## first question
#############################
Clear-Host

while($userReply -ne "Who is there?"){
    $userReply = read-host "Knock Knock!"
}
Clear-Host
while($userReply -ne "Orange who?"){
    $userReply = read-host "Orange"
}
Clear-Host
Write-Output "Orange you glad you created this PowerShell script?"
Start-Sleep -Seconds 5
#############################
## Second Question
#############################
Clear-Host
```

```
while($userReply -ne "Who is there?"){
    $userReply = read-host "Knock Knock!"
}
Clear-Host
while($userReply -ne "Orange who?"){
    $userReply = read-host "Orange"
}
Clear-Host
Write-Output "Oranges are oranges but this is PowerShellscripting!"
Start-Sleep -Seconds 5
############################
## Third Question
############################
Clear-Host
while($userReply -ne "Who is there?"){
    $userReply = read-host "Knock Knock!"
}
Clear-Host
while($userReply -ne "Banana who?"){
    $userReply = read-host "Banana"
}
Clear-Host
Write-Output "Orange you glad I didn't say orange?"
Start-Sleep -Seconds 5
############################
## Farewell Message
############################
Clear-Host
Write-Output "Goodbye!nn"
```

When you run game.ps1, here's the interactive sequence you'll experience:

1. The script will prompt you with "Knock Knock!"
2. You'll need to respond with "Who is there?"
3. The script will then prompt you with "Orange"
4. Until you respond with "Orange who?"
5. The script will then output "Orange you glad you created this PowerShell script?"
6. The second and third questions follow a similar pattern.
7. The script will then output "Goodbye!"

To do:

1. Create another file called beverage.txt identical to drink.txt. (Hint: Use what you learned in practical 1 to do this.)
2. Issue this command again
   Get-ChildItem — select-string coffee
   and see what happens.

3. Create a file called fruit.txt with the list apple, orange, banana in it.
4. Issue this command again
   Get-ChildItem — select-string coffee
   and see what happens.

```
PS C:\Users\User1\green> Copy-Item drink.txt beverage.txt
PS C:\Users\User1\green> Get-ChildItem *.txt
Directory: C:\Users\User1\green

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         2/10/2025   10:15 AM            123 drink.txt
-a----         2/10/2025   10:15 AM            123 beverage.txt
PS C:\Users\User1\green> Get-ChildItem | Select-String coffee
drink.txt:1:coffee.
beverage.txt:1:coffee.
PS C:\Users\User1\green> Set-Content fruit.txt -Value "apple`r`norange`r`nbanana"
PS C:\Users\User1\green> Get-Content fruit.txt
apple
orange
banana
PS C:\Users\User1\green> Get-ChildItem | Select-String coffee
drink.txt:1:coffee.
beverage.txt:1:coffee.
```

Find out and explain how to get help about any cmdlet.

- Find out and explain what F7 does in PowerShell.
- What is the purposes of (a) the –whatif switch and (b) the –confirm
- Write a note to explain how you can use tab to complete a command as soon as it's unambiguous.

In PowerShell, you can obtain help for any cmdlet by typing Get-Help followed by the name of the cmdlet.

```
Get-Help Get-Process
```

This will display detailed information about the Get-Process cmdlet, including a description, syntax, parameters, examples, and more. When you press F7 in a PowerShell console, it brings up a graphical popup window displaying your command history from the current session. You can use the arrow keys to navigate through the list of previously executed commands. This is a handy feature to quickly recall and reuse commands without retyping them.
The -WhatIf switch is used to simulate the execution of a command. It shows you what would happen if the command ran but does not make any actual changes. Use -WhatIf with potentially destructive or impactful commands to verify what actions would be performed.

```
Remove-Item C:\Temp\* -WhatIf
```

The -Confirm switch forces the command to prompt for your confirmation before executing each action. This extra safety measure helps prevent accidental changes.

```
Remove-Item C:\Temp\* -Confirm
```

PowerShell supports intelligent tab completion. As you start typing a command, cmdlet name, parameter, or even file path, you can press the Tab key to auto-complete the text. If the text you've entered uniquely identifies a command or parameter, pressing Tab will automatically complete it. If **multiple completions are possible repeatedly pressing Tab cycles through the available options until you reach the one you want.**

3 Practical 3 - Powershell Part 2

```
$name = Read-Host "Please type your name"
Write-Host "Hello" $name
```

Simply record what happens when you run this script. What difference does it make if you leave out the text "Please type your name" from the first line of the script? What happens:

1. Powershell prompts you with: "Please type your name:"
2. You enter a name (e.g., John).
3. Powershell outputs: "Hello John"
4. If you don't put the text "Please type your name" in the Read-Host command, Powershell will prompt you with a blank line instead of the text.

Find out how to run scripts if they're in a directory other than the working directory. By default, PowerShell restricts script execution for security reasons. You may need to change the execution policy before running scripts: If your script is stored in `C:\Scripts\myscript.ps1` and your working directory is elsewhere, you can run it using:

```
C:\Scripts\myscript.ps1
```

or explicitly call PowerShell:

```
powershell -ExecutionPolicy Bypass -File C:\Scripts\myscript.ps1
```

```
$inputString = read-host
$value = $inputString -as [Double]
write-host "You entered: $value"
```

(a) Find out how to do the same thing that the code above does except that it'll only accept integers (such as 67). (b) Once you're found the answer, find out what happens if you type a real number (such as 67.4 or 67.8)

a. To only accept integers, you can use [Int] instead of [Double]:

```
$value = $inputString -as [Int]
```

b. If you type a real number, PowerShell will round the number down to the nearest integer.

```
do

    write-host -nonewline "Enter a numeric value: "
    $inputString = read-host
    $value = $inputString -as [Double]
    $ok = $value -ne $NULL
    if ( -not $ok ) { write-host "You must enter a numeric value" }
}
until ( $ok )
write-host "You entered: $value"
```

Alter the above program (in brown) to require the user to enter specifically integer values between 1 and 4 inclusive.

```
do
{
    write-host -nonewline "Enter an integer value (1-4): "
    $inputString = read-host
    $value = $inputString -as [Int]
    $ok = ($value -ne $NULL) -and ($value -ge 1) -and ($value -le 4)

    if ( -not $ok ) { write-host "Invalid input. Please enter a whole number between 1 and 4." }
}
until ( $ok )

write-host "You entered: $value"
```

```
------------------------------------------------------------
$i = 1
while ($i -le 20)
{
 if(($i -ne 13) -and ($i -ne 17))
 {
Write-Host $i
 }
 $i = $i + 1
}
```

- What does the above script do?
- We don't need the brackets round (i -ne 13) and (i -ne 17) in the code above. Why do we not need them? (Hint: The answer is the same as for Java). Do you think that it's a good idea to put them in even if they aren't necessary? Explain your answer.

-
  - Initialize i = 1
  - Loop while i <= 20
  - Condition Check: If i is not 13 and i is not 17, print i
  - Otherwise, skip printing.

  - Increment i by 1 each loop iteration.
  - Repeat until i exceeds 20.

- In PowerShell (like in Java), comparison operators are evaluated first. Is it a good idea to include them anyway? Using parentheses is optional but can be a good habit for clarity, especially in more complex conditions.

```
for($i = 1; $i -le 8; $i = $i + 1)
{
Write-Host $i
}
```

We can replace i =i + 1 with something shorter, in the above two scripts. What do you think it is? Try it and see. It's `$i++`.

```
do
{
    Write-Host $i;
    $i = $i + 1;
} until ($i -eq 10);
```

Oops, I've put semicolons at the end of each of the lines, in the do...until loop above. I suppose it's because of my experience in writing programs in other languages that sometimes I put semicolons at the end of a line of PowerShell script, even when they're entirely unnecessary in PowerShell. Does PowerShell forgive me for doing this? Find out, and write your conclusion. The script runs normally without errors. The semicolons do not break execution since PowerShell treats them as harmless separators.

```
$temperature = 1
switch($temperature)
{
 { $_ -lt 0 } { "Below Freezing"; break }
 0 { "Exactly Freezing"; break }
 { $_ -le 10 } { "Cold"; break }
 { $_ -le 20 } { "Warm"; break }
 default { "Hot" }
}
```

Alter this program to deal with grade categories (for example >70 is a distinction mark etc) in an examination and also allow the user to enter a grade.

```
$grade = Read-Host "Enter your exam grade"

switch ($grade -as [int])
{
    { $_ -ge 70 } { "Distinction"; break }
    { $_ -ge 60 } { "Merit"; break }
    { $_ -ge 50 } { "Pass"; break }
    { $_ -ge 40 } { "Borderline Fail"; break }
    default { "Fail" }
}
```

```
$listing = dir
$howLong = $listing.Length
$i = 0;
while($i -lt $howLong)
{
Write-Host $listing[$i].Name
$i++
}
```

1. Explain what the above example does. Modify it to show fields other than the Name field. 2. Draw up a chart to show equivalent syntaxes for different control structures/data structures among Java, Unix script, and PowerShell. You'll have to revisit this question when you've learned some Unix/Linux. 3. Compare the ways in which scripts are enabled to run in (a) Unix/Linux and (b) PowerShell. Again this is a question for review when you've done some Unix/Linux.

1. Retrieves a list of files and directories in the current directory (dir is an alias for Get-ChildItem).
2. Iterates through the list and prints the name of each item.
3. Prints the Name property of each file/directory ($listing[$i].Name).
4. Increments $i by 1 in each iteration.

To show additional fields like Size, LastWriteTime, and Mode, modify the script as follows:

```
$listing = dir
$howLong = $listing.Length
$i = 0
while($i -lt $howLong)
{
    Write-Host "$($listing[$i].Mode) $($listing[$i].Length)
    $($listing[$i].LastWriteTime) $($listing[$i].Name)"
    $i++
}
```

Concept	PowerShell	Java	Bash
Variable Assignment	`$x = 5`	`int x = 5;`	`x=5`
If	`if ($x -gt 10) { ... }`	`if (x > 10) { ... }`	`if [ $x -gt 10 ]; then ... fi`
For	`for ($i=0; $i -lt 10; $i++)`	`for (int i=0; i<10; i++)`	`for i in {0..9}; do ... done`
While	`while ($x -lt 10)`	`while (x < 10)`	`while [ $x -lt 10 ]; do ... done`
Switch	`switch ($var) { ... }`	`switch (var) { case 1: ... }`	`case $var in ... esac`
Array	`$arr = @(1,2,3)`	`int[] arr = {1,2,3}`	`arr=(1 2 3)`
Function	`function MyFunc { ... }`	`int myFunc(int x) { ... }`	`my_func() { ... }`

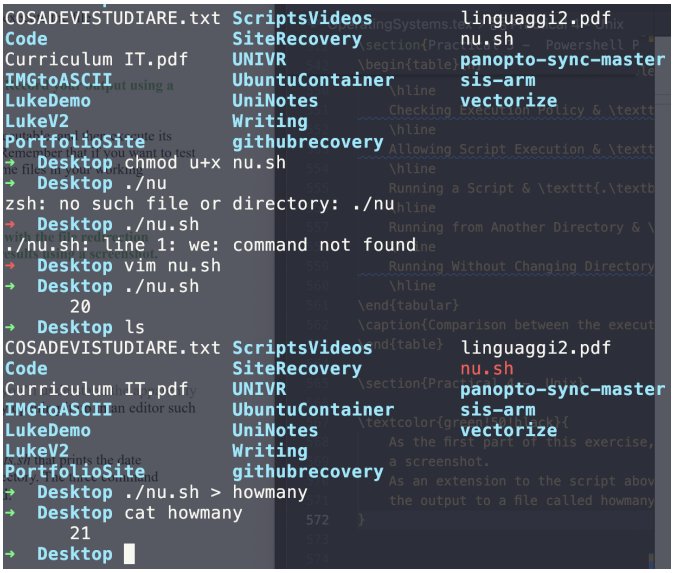
Figure 1: Syntax between PowerShell, Java e Bash

Feature	PowerShell	Unix/Linux
Default Script Execution	Disabled for security (Restricted mode)	Allowed but may require execution permissions.
Checking Execution Policy	`Get-ExecutionPolicy`	`ls -l script.sh`
Allowing Script Execution	`Set-ExecutionPolicy RemoteSigned`	`chmod +x script.sh`
Running a Script	`.\script.ps1`	`./script.sh`
Running from Another Directory	`C:\Scripts\script.ps1`	`/home/user/script.sh`
Running Without Changing Directory	`& "C:\Scripts\script.ps1"`	`bash /path/to/script.sh`

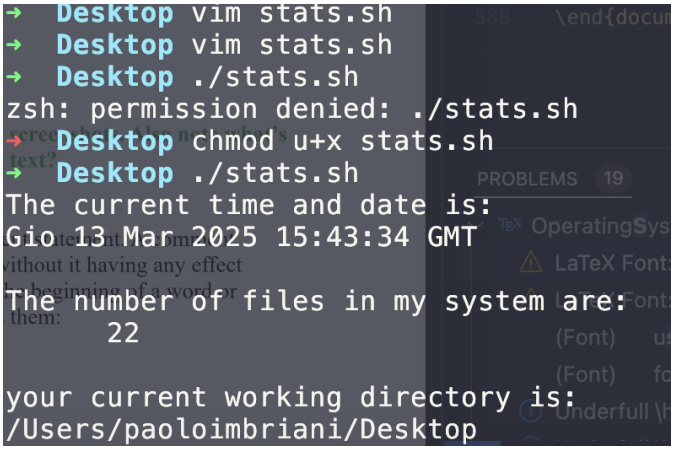
Table 1: Comparison between the execution in PowerShell e Unix/Linux

4 Practical 4 - Unix

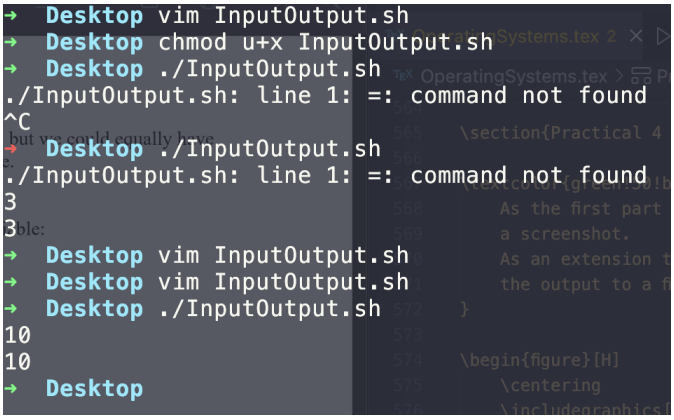
As the first part of this exercise, create and run the above scripts shell scripts. Record your output using a screenshot. As an extension to the script above use the shell script in conjuction with the file redirection operator to redirect the output to a file called howmany; record your results using a screenshot.



Write and execute the above shell script and record your result using screenshots. Also note what's the significance of the echo statement on a line without any succeding text?

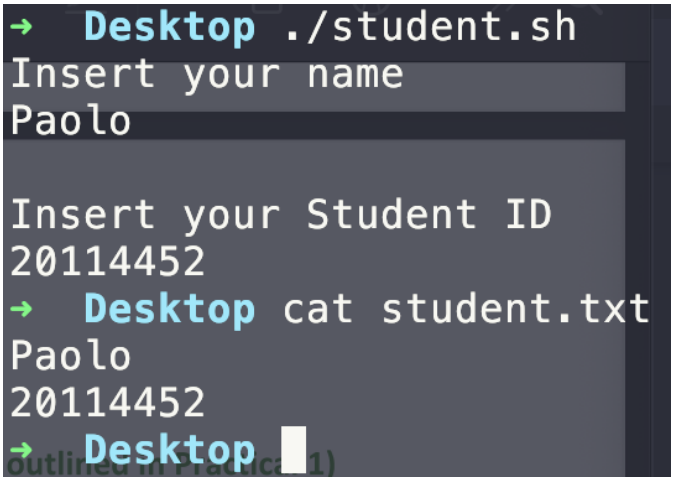


The echo with no succeding text is used to print a blank line. It's often used for formatting output or creating space between sections of a script. Put the above script into a file called InputOutput.sh and run it. Record your results using screenshots.



Write a shell script (student.sh) to ask the user to enter two fields. Their name and student ID. The script should append the data to a text file called student.txt.

```
#!/bin/bash
echo "Enter your name:"
read name
echo "Enter your student ID:"
read id
echo $name >> student.txt
echo $id >> student.txt
```



5 Practical 5 - Unix Part 2

```
x=8
y=5
expr $x + $y
```

One might have supposed that echo would work on the last line above instead of expr. Try it with echo and see what happens. Later we'll see a way of doing what this program does using echo. If you replace expr with echo, the output will be 8 + 5 instead of the sum of the two numbers. This is because echo simply prints the text, while expr evaluates the expression.

Something for you to find out: Find out how to do multiplication in Linux script. It is not as you might suppose, simply by replacing + with * To perform multiplication in a bash file, you cannot use the * operator directly as it will be interpreted as a wildcard. Instead, you can use the expr command with the * operator enclosed in quotes:

```
expr $x \* $y
```

```
# Sums two numbers supplied on the command line
#
if
    [ $# -ne 2 ]
then
    echo two args
    echo please
else
    echo sum is
    expr $1 + $2
fi
```

Modify the above code (addnums.sh) so that a third command line argument is used to specify whether the two numbers are to be added or multiplied.

```
if [ $# -ne 3 ]
then
    echo "Usage: $0 <num1> <num2> <operation>"
    exit 1
else
    if [ $3 = "add" ]
    then
        echo "Sum is: $(($1 + $2))"
    elif [ $3 = "multiply" ]
    then
        echo "Product is: $(($1 * $2))"
    else
        echo "Invalid operation. Please use 'add' or 'multiply'."
    fi
fi
```

```
# This script removes all files in the working directory
echo "This will remove all files in the current working
directory!"
echo "Are you sure (y/n)?"
read response
if
    [ $response = "y" ]
then
    rm *
    echo files removed
else
    echo not removed
if
```

To do: 4. Rewrite the above program with using != rather than =. 5. Modify the above script to replace rm * with rm –i * and note the difference.

```
echo "Are you sure (y/n)?"
read response
if [ $response != "y" ]
then
    echo "not removed"
else
    rm -i *
    echo "files removed"
fi
```

The -i flag prompts the user for confirmation before deleting each file. This adds an extra layer of safety to prevent accidental deletions.

```
read mark
if
    [ $mark -lt 50 ]
then
    echo \Sorry, not passed!"
fi
```

What difference does it make if the inverted commas around Sorry, not passed!! is omitted? If the inverted commas are omitted, the script will produce an error because the shell will interpret the exclamation marks as special characters. The script will not run correctly without the quotes.

```
while
    [ condition ] space after [ and space before ]
do
    commands
done
```

Something to do: I think you're allowed to put while [condition] on one line with the arbornet linux, but check it out for yourself if you have an account on arbornet. Am I right or am I wrong? Yes, you can put the while condition on one line without issues. The shell does not require the condition to be on a separate line.

```
a=10
b=20
if [ $a != $b ]
then
 echo "$a != $b : a is not equal to b"
else
 echo "$a != $b: a is equal to b"
fi
if [ $a -lt 100 -a $b -gt 15 ]
then
 echo "$a -lt 100 -a $b -gt 15 : returns true"
else
 echo "$a -lt 100 -a $b -gt 15 : returns false"
fi
if [ $a -lt 100 -o $b -gt 100 ]
then
 echo "$a -lt 100 -o $b -gt 100 : returns true"
else
 echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

```
if [ $a -lt 5 -o $b -gt 100 ]
then
 echo "$a -lt 100 -o $b -gt 100 : returns true"
else
 echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

To do: 8. Modify the program at the end of section 6 to print all the numbers between 1 and 20 except the supposedly unlucky 13. 9. Modify your answer to the above problem to exclude 17 also. 10. Modify the above program to print all the numbers between 1 and 20 except a number provided by the user after a request to do so by the program. 11. Write a program to print out the series 1, 3, 6, 10, 15, 21... until the number in the series below just below 200.

```
increment=1
num=1
while
    [ $num -le 20 ]
do
    if
        [ $num -ne 13 ] || [ $num -ne 17 ]
    then
        echo $num
        num=$(($num + $increment))
    fi
done
```

```
################################
increment=1
num=1
echo "Enter the number to exclude:"
read exclude
while
    [ $num -le 20 ]
do
    if
        [ $num -ne $exclude ]
    then
        echo $num
        num=$(($num + $increment))
    fi
done
```

```
################################
increment=1
num=1
while
    [ $num -le 200 ]
do
    echo $num
    increment=$(($increment + 1))
    num=$(($num + $increment))
done
```

12. Write a Linux script to give the user up to 10 chances to guess a number the computer is "thinking of". For example, suppose the computer is thinking of the number 53, say, and the user has to guess that. Assume the user only knows that the number is between, 40 and 70, say; thus he/she has a sporting

chance. This is a game of chance only, since there's no skill involved. When the game is finished a report is given to the user telling him/her whether he guessed correctly or not.

```
#!/bin/bash
number=53
chances=0
echo "Guess a number between 40 and 70:"
while [ $chances -lt 10 ]
do
    read guess
    if [ $guess -eq $number ]
    then
        echo "Congratulations! You guessed correctly."
        exit 0
    else
        echo "Incorrect guess. Try again."
        chances=$(($chances + 1))
    fi
done
echo "Out of chances. The number was $number."
```