Chapter 5: Flow Control

Flow control is the ability to control the flow of a program's execution- how much or what part of the program need to run, or what part need to be repeatedly run etc.

The flow control constructs that bash supports are:

1. If/else→ executes some statements when a condition is satisfied or not.....***CONDITIONAL***
2. For→executes statements for a fixed number of times.
3. While→ executing statements while a certain condition is satisfied or held.
4. Case→executes one of the statements depending upon the value of the variable.
5. Select→allows the user to select one of the possibilities from the menu.

If/else construct

The syntax of the if-else construct is:

Here, the if-else construct ends with fi(if spelt backwards) Also, you can write everything in a single line using semi colon:

if condition; then statement1; else statement2; fi

elif: else if→ used when there are multiple conditions.

If you use one or more elifs, you can think of the else clause as the "if all else fails" part.

```
if condition
then
    statement1
else
    statement2
fi
```

EXIT STATUS:

Every UNIX command returns an integer code to its calling process, here the shell, when it finishes. That is the exit status. It's not shown on the screen by default, but it **indicates success or failure** of the command.

0→ OK exit status

1 to 255→ ERROR exit status

```
gayat@GSP-HP:~$ echo $?
0
```

```
gayat@GSP-HP:~/SummerInternship$ cd SummerInternship/ ; echo $?
bash: cd: SummerInternship/: No such file or directory
1
gayat@GSP-HP:~/SummerInternship$ cd test2705/ ; echo $?
0
gayat@GSP-HP:~/SummerInternship/test2705$
```

If-else examples:

```
gayat@GSP-HP:~$ nano pnz.sh
gayat@GSP-HP:~$ chmod +x pnz.sh
gayat@GSP-HP:~$ ./pnz.sh
Enter the number:
8
8 is positive
gayat@GSP-HP:~$ nano pnz.sh
gayat@GSP-HP:~$ chmod +x pnz.sh
gayat@GSP-HP:~$ ./pnz.sh
Enter the number:
-980
-980 is negative
```

```
echo "Enter the number:"
read num

if (( num > 0 ))
then    echo "$num is positive"
elif (( num < 0 ))
then
        echo "$num is negative"
else
        echo "$num is 0"
fi
```

```
echo "Enter you age:"
read age

if (( age > 17 ))
then
        echo "Congratulations! You are eligible to vote!"
else
         echo "Oh no! You are ineligible to vote!"
fi
```

```
gayat@GSP-HP:~$ cd SummerInternship/
gayat@GSP-HP:~/SummerInternship$ ls
 123.txt   'Untitled - Copy.txt:Zone.Identifier'   and         myfilecreated.txt   test2705
 5          age.sh                                 hello.txt   new.txt             time
 5678.txt   albums                                 mouth,     'pushd ().sh'
gayat@GSP-HP:~/SummerInternship$ nano age.sh
gayat@GSP-HP:~/SummerInternship$ chmod + age.sh
gayat@GSP-HP:~/SummerInternship$ ./age.sh
Enter you age:
56
Congratulations! You are eligible to vote!
gayat@GSP-HP:~/SummerInternship$ 14
14: command not found
gayat@GSP-HP:~/SummerInternship$ chmod + age.sh
gayat@GSP-HP:~/SummerInternship$ ./age.sh
Enter you age:
14
Oh no! You are ineligible to vote!
gayat@GSP-HP:~/SummerInternship$
```

```
echo "Enter the number:"
read num

if (( $num > 0 ))
then
        echo "The number is positive!"
elif (( $num < 0 ))
then
        echo "The number is negative!"
else
        echo "The number is zero, which is neither positive nor negative!"
fi
```

```
gayat@GSP-HP:~$ nano pn.sh
gayat@GSP-HP:~$ chmod +x pn.sh
gayat@GSP-HP:~$ ./pn.sh
Enter the number:
45
The number is positive!
gayat@GSP-HP:~$ chmod +x pn.sh
gayat@GSP-HP:~$ ./pn.sh
Enter the number:
-98
The number is negative!
gayat@GSP-HP:~$ chmod +x pn.sh
gayat@GSP-HP:~$ ./pn.sh
Enter the number:
0
The number is zero, which is neither positive nor negative!
gayat@GSP-HP:~$
```

***builtin* command!**

Tells the shell to use the built-in commands and ignore any functions that have the same name as the built-in commands. Used when there exist functions with the same name as a command, and since the functions have higher priority than built-in commands, we need to use this command to run the commands instead.

To know if a command is built-in or external:

```
gayat@GSP-HP:~$ type cd
cd is a shell builtin
gayat@GSP-HP:~$ type ls
ls is aliased to `ls --color=auto'
gayat@GSP-HP:~$ type pwd
pwd is a shell builtin
gayat@GSP-HP:~$ type cat
cat is /usr/bin/cat
gayat@GSP-HP:~$
```

```
gayat@GSP-HP:~$ builtin cd
gayat@GSP-HP:~$ builtin pwd
/home/gayat
gayat@GSP-HP:~$ builtin alias
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)" "$(h
istory|tail -n1|sed -e '\''s/^\s*[0-9]\+\s*//;s/[;&|]\s*alert$//'\'')"'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
gayat@GSP-HP:~$ builtin ls
bash: builtin: ls: not a shell builtin
gayat@GSP-HP:~$ builtin cat
bash: builtin: cat: not a shell builtin
gayat@GSP-HP:~$
```

Return:

return N→ a statement causing the surrounding function to exit with exit status; written in the end of the function. Till now, we haven't used the return statement; it returns whatever the last statement returns.

Used inside a function, not on a terminal! Also, in shell scripts that can be executed using the source command.

Also, the command exit N exits the shell script, no matter how deeply you are nested with functions!

```
gayat@GSP-HP:~/SummerInternship/test2705$ return N
bash: return: N: numeric argument required
bash: return: can only `return' from a function or sourced script
gayat@GSP-HP:~/SummerInternship/test2705$
```

Combination of Exit Statuses

These are like 'and' and 'or' constructs.

Statement1 && statement2

→ execute statement1 and if its exit status is 0, execute statement2

Statement1 || statement2

→ execute statement1 and if its exit status is 1, execute statement2

```
if statement1 && statement2
then
    ...     #If statement1 runs, it gives a 0 as exit status,
            #then statement2 is executed and expected to give a 0 too
fi
#If statement1 runs and gives 1, then statemnet 2 won't run.
#So, then will only run if both the statements run to give a 0 exit status
```

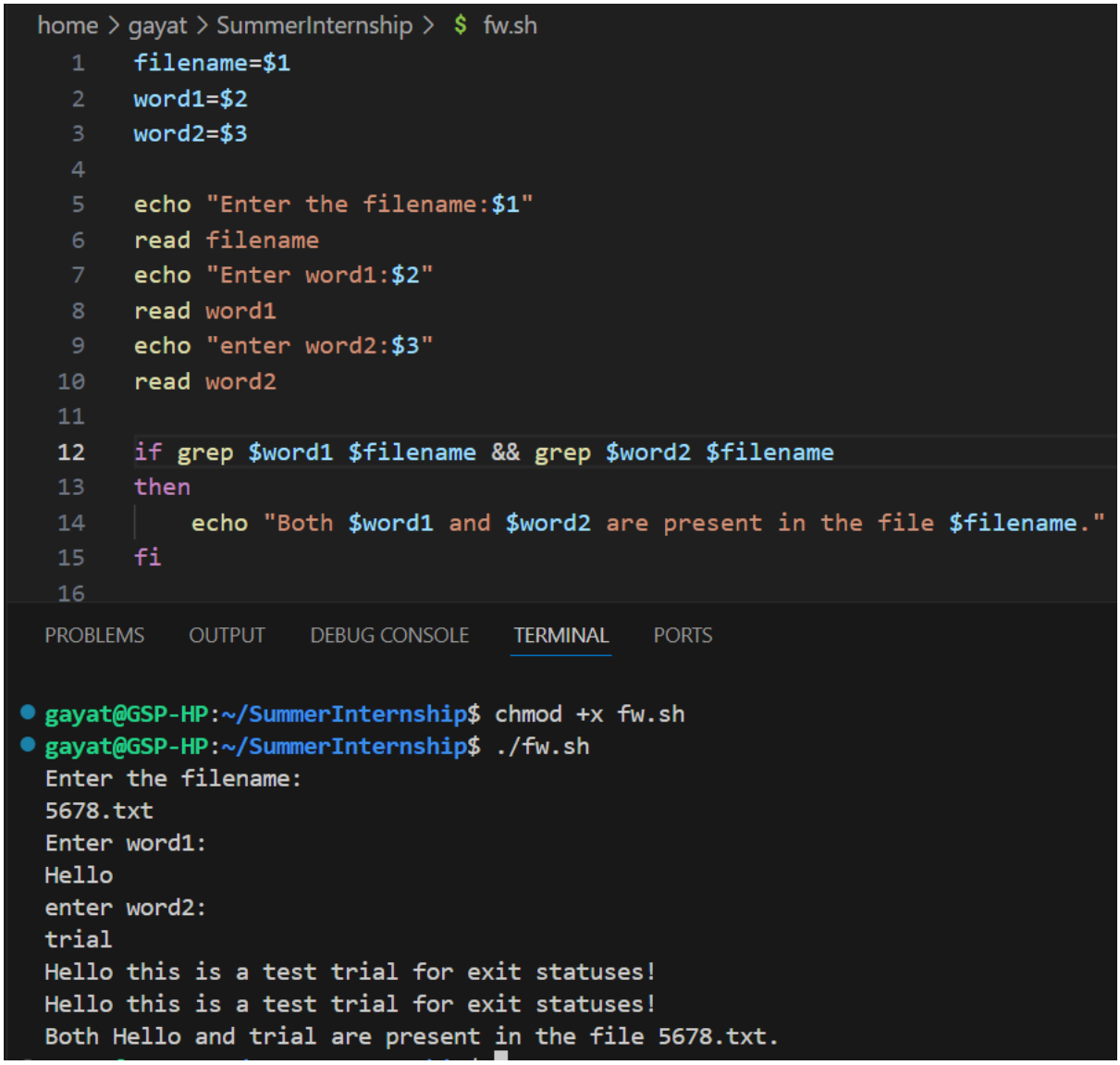
```
if statement1 || statement2
then
    ...     #If statement1 runs, it gives a 1 as exit status,
            #then statement2 is executed. If statement1 doesn't succeed, statement2 won't run.
fi
#If statement1 runs and gives 0, then statemnet 2 won't run and then runs.
#So, 'then' will only run if both either the statement1 or 2 run to give a 0 exit status
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x grep.sh
gayat@GSP-HP:~/SummerInternship$ ./grep.sh
Enter the filename:
5678.txt
enter word1:
Hello
enter word2:
test
Hello this is a test trial for exit statuses!
The Hello or test is present in the file 5678.txt
```

albums | $ grep.sh

home > gayat > SummerInternship > $ grep.sh

```
1   filename=$1
2   word1=$2
3   word2=$3
4
5   echo "Enter the filename:$1"
6   read filename
7   echo "enter word1:$2"
8   read word1
9   echo "enter word2:$3"
10  read word2
11
12  if grep $word1 $filename || grep $word2 $filename
13  then
14      echo "The $word1 or $word2 is present in the file $filename"
15  fi
```



CONDITION TESTS

Exit status is the only thing that can be tested using an if construct

2 other ways to check if a condition is working:

1. [...]
2. [[....]]→ different from the first one in that the words pathname and splitting are not performed on the words within the brackets.

[condition] is actually a statement just like any other, except that the only thing it does is return an exit status that tells whether condition is true.

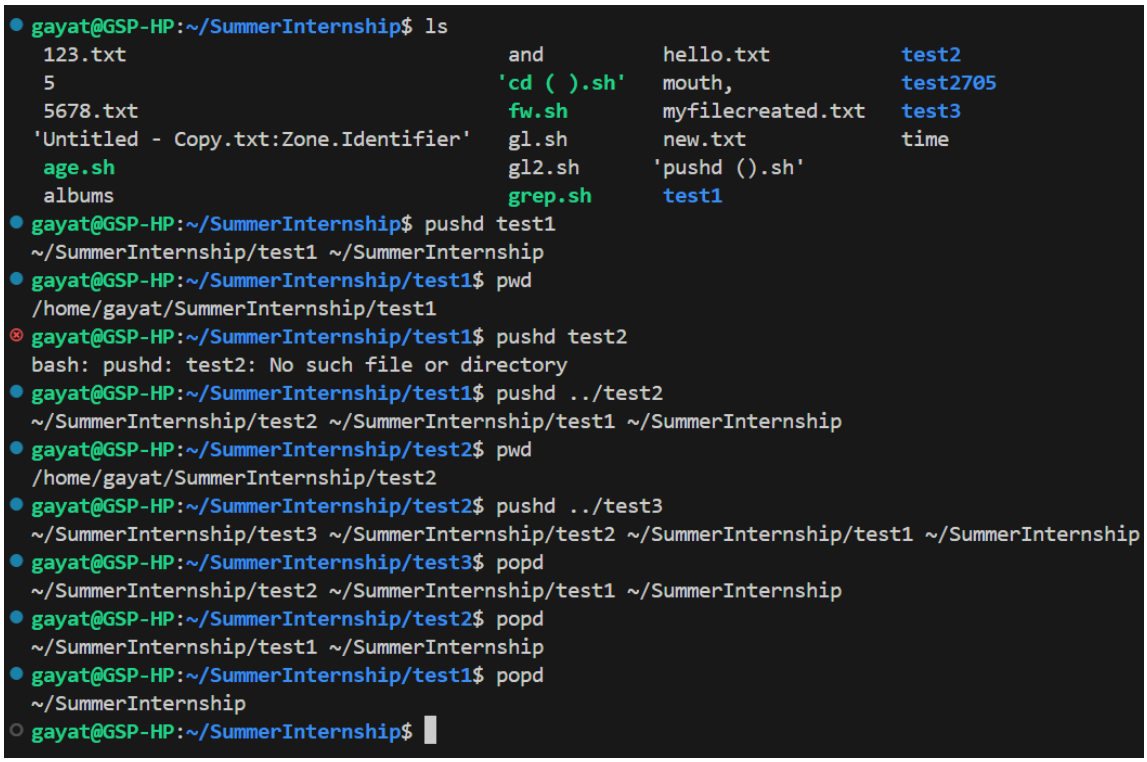
STRING COMPARISONS- used with the []→ called test commands

Operator	True if...
str1 = *str2*[4]	*str1* matches *str2*
str1 != *str2*	*str1* does not match *str2*
str1 < *str2*	*str1* is less than *str2*
str1 > *str2*	*str1* is greater than *str2*
-n *str1*	*str1* is not null (has length greater than 0)

Operator	True if...
-z *str1*	*str1* is null (has length 0)

[4] Note that there is only one equal sign (=). This is a common source of error.

Rewind: (Below) The exact use of pushd and popd



Examples of string comparison!

home > gayat > SummerInternship > $ comp.sh

```
1
2   echo "Enter the first word:"
3   read word1
4   echo "Enter the second word:"
5   read word2
6
7   if [ "$word1" == "$word2" ]; then
8       echo "They are the same!"
9   else
10      echo "Aww! They are different!"
11  fi
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
gayat@GSP-HP:~/SummerInternship$ ./stg1\=\"Hello\".sh
They are different.
gayat@GSP-HP:~/SummerInternship$ chmod +x comp.sh
gayat@GSP-HP:~/SummerInternship$ ./comp.sh
Enter the first word:
Hello
Enter the second word:
hello
Aww! They are different!
gayat@GSP-HP:~/SummerInternship$ chmod +x comp.sh
gayat@GSP-HP:~/SummerInternship$ ./comp.sh
Enter the first word:
Hi
Enter the second word:
Hi
They are the same!
```

albums | fw.sh | stg1="Hello".sh | comp.s

home > gayat > SummerInternship > stg1="Hello".sh

```
1  stg1="Hello"
2  stg2="world"
3
4  if [ "$stg1" == "$stg2" ]; then
5      echo "They are equal."
6  else
7      echo "They are different."
8  fi
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
gayat@GSP-HP:~$ cd SummerInternship/
gayat@GSP-HP:~/SummerInternship$ chmod +x stg1\=\"Hello\".sh
gayat@GSP-HP:~/SummerInternship$ ./stg1\=\"Hello\".sh
They are different.
```

stg1="Hello".sh | user.sh

home > gayat > SummerInternship > user.sh

```
1
2  echo "Enter the username:"
3  read name
4
5  if [ "$name" != "" ]; then
6      echo "Hello $name! Nice to meet you and welcome to Bash scripting!"
7  else
8      echo "Aww, No username avaliable to greet!"
9  fi
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
 5                                  gl.sh              test1
 5678.txt                           gl2.sh             test2
'Untitled - Copy.txt:Zone.Identifier'  grep.sh         test2705
 age.sh                            hello.txt          test3
 albums                            mouth,             time
 and                               myfilecreated.txt  user.sh
'cd ( ).sh'                        new.txt
 comp.sh                           'pushd ().sh'
gayat@GSP-HP:~/SummerInternship$ chmod +x user.sh
gayat@GSP-HP:~/SummerInternship$ ./user.sh
Enter the username:
Gayathri
Hello Gayathri! Nice to meet you and welcome to Bash scripting!
gayat@GSP-HP:~/SummerInternship$ chmod +x user.sh
gayat@GSP-HP:~/SummerInternship$ ./user.sh
Enter the username:

Aww, No username avaliable to greet!
gayat@GSP-HP:~/SummerInternship$
```

```
$ stg1="Hello".sh   $ user.sh   $ pass.sh
home > gayat > SummerInternship > $ pass.sh
1   password="secret123"
2
3   echo "Enter passcode:"
4   read passcode
5
6   if [ "$password" == "$passcode" ]; then
7       echo "Valid Passcode! welcome in!"
8   else
9       echo "Incorrect passcode! try again."
10  fi
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

gayat@GSP-HP:~/SummerInternship$ chmod +x pass.sh
gayat@GSP-HP:~/SummerInternship$ ./pass.sh
Enter passcode:
heko
Incorrect passcode! try again.
gayat@GSP-HP:~/SummerInternship$ chmod +x pass.sh
gayat@GSP-HP:~/SummerInternship$ ./pass.sh
Enter passcode:
secret123
Valid Passcode! welcome in!
gayat@GSP-HP:~/SummerInternship$
```

```
1   echo "Enter a word:"
2   read str
3
4   first_char=${str:0:1}
5
6   if [[ "$first_char" == A || "$first_char" == a ]]; then
7       echo "Starts with a; hence valid!"
8   else
9       echo "Starts with another letter; hence invalid!"
10  fi
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter a word:
Apple
Starts with a; hence valid!
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter a word:
orange
Starts with another letter; hence invalid!
```

if ["$first_char" == A] || ["$first_char" == a];

This also gives the same output

FILE ATTRIBUTE CHECKING WITH OPERATORS:

A single command is considered each time.

Operator	Meaning
-e	File **exists**
-f	File **exists and is a regular file**
-d	File **is a directory**
-r	File is **readable**

Operator	Meaning
-w	File is **writable**
-x	File is **executable**
-s	File is **not empty**
!	Logical NOT (e.g., ! -e)

For && and ||, we use [[ ]] or separate [] for each part.

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the filename:
Age
The file named Age doesn't exist here! Please check the name and try again!
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the filename:
5678.txt
the file named 5678.txt exits!
It's a regular file.
It's a readable file.
It's a writable file.
It's a non empty file.
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the filename:
Hello
The file named Hello doesn't exist here! Please check the name and try again!
```

```
echo "Enter the filename:"
read filename

if [ -a "$filename" ]; then
    echo "the file named $filename exits!"
    [[ -f "$filename" ]] && echo "It's a regular file."
    [[ -d "$filename" ]] && echo "It's a directory."
    [[ -r "$filename" ]] && echo "It's a readable file."
    [[ -w "$filename" ]] && echo "It's a writable file."
    [[ -x "$filename" ]] && echo "It's a executable file."
    [[ -s "$filename" ]] && echo "It's a non empty file."
else
    echo "The file named $filename doesn't exist here! Please check the name and try again!"
fi
```

Differences between ||, -o and &&, -a

&&, ||→ used between full commands, not inside [...]

-a, -o→ inside a single test expression→ also uses [[....]]

Integer Conditionals:

For arithmetic tests, different from character string comparisons like < and>

They're necessary if you want to combine integer tests with other types of tests within the same conditional expression.

Test	Comparison
-lt	Less than
-le	Less than or equal
-eq	Equal

Test	Comparison
-ge	Greater than or equal
-gt	Greater than
-ne	Not equal

for construct:

Also known as the looping construct, it is used to report multiple results instead of a single one. A for loop allows you to repeat a section of the code for a fixed number of times. A loop variable is set during each loop/ iteration, and is set to a fixed value.

The chief difference between the for loop in shell and in C programming is the fact that it doesn't let the user specify a number of times to iterate or a range of values over which to iterate; instead lets the user give a fixed list of values. (It lets you give a range of values to consider, not a single value. If you want counting or ranges, you need to use seq, brace expansion ({1..5}), or C-style loops in Bash.)

The syntax for for loop:

```
for var in list
do
   commands
done
```

```
1
2  for fruit in apple banana cherry
3  do
4      echo "I like $fruit"
5  done
6
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A
A.sh    age.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
I like apple
I like banana
I like cherry
gayat@GSP-HP:~/SummerInternship$
```

Some for loop examples:

To print the numbers:

```
for num in {1..10}
do
    echo "$num"
done
```

```
for num in 1 2 3 4 5 6 7 8 9 10
do
    echo "$num"
done
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
1
2
3
4
5
6
7
8
9
10
```

To print even numbers between 2 and 20:

```
for num in {2..20}
do
    if (( "$num" % 2 == 0 )); then
    echo "$num"
    fi
done
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
2
4
6
8
10
12
14
16
18
20
```

To create 5 files with names file1.txt, file2.txt etc:

```
for i in {1..5}
do
touch "file$i.txt"
done
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
gayat@GSP-HP:~/SummerInternship$ ls
 123.txt                              'cd ( ).sh'   gl.sh              'pushd ().sh'
 5                                    comp.sh       gl2.sh             'stg1="Hello".sh'
 5678.txt                             file1.txt     grep.sh             test1
 A.sh                                 file2.txt     hello.txt           test2
'Untitled - Copy.txt:Zone.Identifier' file3.txt     mouth,              test2705
 age.sh                               file4.txt     myfilecreated.txt   test3
 albums                               file5.txt     new.txt             time
 and                                  fw.sh         pass.sh             user.sh
```

Countdown:

```
for i in {10..1}
do
    echo "$i"
done
echo "Blast Off"
```

```
gayat@GSP-HP:~/SummerInternship$ ./A.sh
10
9
8
7
6
5
4
3
2
1
Blast Off
```

Multiplication product:

```bash
for num in {1..5}
do
echo "$num * 2 = $((2 * num))"
done
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
1 * 2 = 2
2 * 2 = 4
3 * 2 = 6
4 * 2 = 8
5 * 2 = 10
```

Note: for arithmetic evaluation, use (())→check the above example for the same.

Q. Print command-line arguments one by one. Write a script that loops over all the arguments passed to it and prints them:

```bash
for arg in "$@"
do
    echo "Argument: $arg"
done
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh 1 2 3 4 5
Argument: 1
Argument: 2
Argument: 3
Argument: 4
Argument: 5
```

Q. Print the current directory's filenames. Loop through the output of ls and print the name of each file.

```bash
for op in $(ls)
do
echo "$op"
done
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
123.txt
5
5678.txt
A.sh
Untitled
-
Copy.txt:Zone.Identifier
age.sh
albums
and
cd
(
).sh
comp.sh
file1.txt
```

Checking for a file from a list:

```bash
echo "Enter the filename:"
read filename
found=false

for file in file1.txt file2.txt notes.txt
do
    if [[ "$file" == "$filename" ]]; then
    found=true
    break
    fi
done

if $found; then
    echo "Exist"
    else
    echo "Missing."
    fi
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the filename:
notes.txt
Exist
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the filename:
hello.txt
Missing.
```

RECURSION:

A function calls itself to solve smaller parts of the problems. *(“To solve this big thing, I’ll*

solve a smaller version of the same thing... and repeat that until it's small enough to finish easily.")

Key Parts of Recursion:

1. **Recursive call** – function calling itself.
2. **Base case** – when to **stop** recursion (to avoid infinite loop).

<u>case command:</u>

Bash's case construct lets you test strings against patterns that can contain wildcard characters.

```
case "$variable" in
    pattern1)
        # commands for pattern1
        ;;
    pattern2)
        # commands for pattern2
        ;;
    pattern3 | pattern4)
        # commands if pattern3 OR pattern4 matches
        ;;
    *)
        # default case (if no pattern matches)
        ;;
esac
```

|are used to separate patterns on a single line, so that if the expression matches one of them, the corresponding statement is executed.

Examples of case:

```
echo "Enter the day:"
read day

case "$day" in
"Monday")
   echo "New week begins today!"
   ;;
"Friday")
   echo "End of the workday!"
   ;;
"Saturday" | "Sunday")
   echo "Weekend on!"
   ;;
*)
   echo "Just a weekday!"
   ;;
esac
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the day:
Monday
New week begins today!
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the day:
Wednesday
Just a weekday!
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the day:
Friday
End of the workday!
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the day:
Saturday
Weekend on!
```

```
echo "Enter the file:"
read file

case "$file" in
   *.txt)
      echo "It's a textfile!"
      ;;
   *.sh)
      echo "It's a script!"
      ;;
   *.pdf)
      echo "It's a pdf!"
      ;;
   *)
      echo "Unknown file type!"
      ;;
esac
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the file:
5678.txt
It's a textfile!
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the file:
test2705
Unknown file type!
```

```
echo "Enter the operator:"
read operator
echo "Enter the numbers a and b:"
read a b

case "$operator" in
"+") echo "The sum is "$(( a + b ))""
;;
"-") echo "The difference is "$(( a - b ))""
;;
"*") echo "The product is "$(( a * b ))""
;;
"/") echo "The quotient is "$(( a / b ))""
;;
"%") echo "The reminder is "$(( a % b ))""
;;
*) echo "Invalid operator, try again!"
;;
esac
```

```
Enter the operator:
+
Enter the numbers a and b:
45 55
The sum is 100
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the operator:
-
Enter the numbers a and b:
4 7
The difference is -3
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the operator:
*
Enter the numbers a and b:
45 78
The product is 3510
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the operator:
/
Enter the numbers a and b:
65 5
The quotient is 13
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the operator:
%
Enter the numbers a and b:
65 5
The reminder is 0
```

select command:

A command only seen in bash and K shell!

select allows you to generate simple menus easily. It has concise syntax, but it does quite a lot of work.

1. Generate a menu for each item in a list
2. prompts the user for a number.

3. Stores the selected choice in the variable name and the selected number in the built-in variable REPLY

4. Executes the statements in the body

5. Repeats the process forever (but see below for how to exit)

```
select variable
 [in
#list of th things
]
do
# commands or statements to be executed
$variable
done
```

Examples:

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
1) Apple
2) Orange
3) Mango
4) Exit
#? 1
fruit is Apple!
#? 2
fruit is Orange!
#? 3
fruit is Mango!
#? 4
Goodbye!
gayat@GSP-HP:~/SummerInternship$
```

```
select fruit in Apple Orange Mango Exit
do
    case $fruit in
    "Apple") echo "fruit is $fruit!"
    ;;
     "Orange") echo "fruit is $fruit!"
    ;;
     "Mango") echo "fruit is $fruit!"
    ;;
     "Exit") echo "Goodbye!"
      break ;;
    *) echo "Invalid option!"
    ;;
     esac
done
```

```
select option in "List files" "Show current directory" "Exit"
do
    case $option in
        "List files") ls
        ;;
        "Show current directory") pwd
        ;;
        "Exit") echo "Exiting.."
        break;;
        *) echo "invalid option!"
        ;;
        esac
    done
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
1) List files
2) Show current directory
3) Exit
#? 1
 123.txt                            'cd ( ).sh'   gl.sh              'pushd ().sh'
 5                                   comp.sh      gl2.sh             'stg1="Hello".sh'
 5678.txt                            file1.txt    grep.sh             test1
 A.sh                                file2.txt    hello.txt           test2
'Untitled - Copy.txt:Zone.Identifier' file3.txt    mouth,              test2705
 age.sh                              file4.txt    myfilecreated.txt   test3
 albums                              file5.txt    new.txt             time
 and                                 fw.sh        pass.sh             user.sh
#? 2
/home/gayat/SummerInternship
#? 3
Exiting..
```

Makes a list for the user to which the user types in the number for each command required which is then run by the shell. The select loop The break statement exits from

the *select* loop. This is generated by the PS3, which contains the prompt string #? When you press Enter again, the whole menu is printed again. A break is necessary for exiting the select when the user makes a valid choice.

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Enter the numbers a and b:
6 7
1) +
2) -
3) /
4) *
5) %
6) Exit
#? 1
13
#? 2
-1
#? 3
0
#? 4
42
#? 5
6
#? 78
invalid option
#? 6
Exiting
gayat@GSP-HP:~/SummerInternship$
```

```
echo "Enter the numbers a and b:"
read a b

select operator in "+" "-" "/" "*" "%" "Exit"
do
   case $operator in
   "+") echo "$(( a + b ))";;
   "-") echo "$(( a - b ))";;
   "*") echo "$(( a * b ))";;
   "/") echo "$(( a / b ))";;
   "%") echo "$(( a % b ))";;
   "Exit") echo "Exiting"
   break;;
   *) echo "invalid option"
   ;;
   esac
done
```

```
select day in "Monday" "Tuesday" "Wednesday" "Thursday" "Friday" "Saturday" "Sunday" "Exit"
do
   case $day in
   "Monday") echo "Ah, Monday blues!!" ;;
   "Tuesday" | "Wednesday" | "Thursday" ) echo "And the week moves ahead to weekend!" ;;
   "Friday") echo "Weekend mode on!" ;;
   "Saturday" | "Sunday") echo "It's rejuvenation time!";;
   "Exit")  echo "Exiting"
   break;;
   *) echo "Not a day that I know!"
   ;;
   esac
done
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
1) Monday     3) Wednesday  5) Friday     7) Sunday
2) Tuesday    4) Thursday   6) Saturday   8) Exit
#? 1
Ah, Monday blues!!
#? 2
And the week moves ahead to weekend!
#? 3
And the week moves ahead to weekend!
#? 4
And the week moves ahead to weekend!
#? 5
Weekend mode on!
#? 6
It's rejuvenation time!
#? 7
It's rejuvenation time!
#? 890
Not a day that I know!
#? 76
Not a day that I know!
#? 8
Exiting
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
1) dark
2) light
3) blue
4) Exit
#? 1
Theme set to dark
#? 2
Theme set to light
#? 3
Theme set to blue
#? 45
Not an available theme option
#? 4
Exiting
gayat@GSP-HP:~/SummerInternship$
```

```
select theme in "dark" "light" "blue" "Exit"
do
    case $theme in
    "dark") echo "Theme set to $theme" ;;
    "light") echo "Theme set to $theme" ;;
    "blue") echo "Theme set to $theme" ;;
    "Exit")  echo "Exiting"
    break;;
    *) echo "Not an available theme option"
    ;;
    esac
done
```

While and until constructs:

They both allow a section of code to be run repetitively while (or until) a certain condition becomes true.

Until→ "Do statements until command runs correctly."

While→ “Do the statements while these commands are true.”

```
while condition do
    command
done
```

```
until command;
 do
 #statement..
 done
```

Syntax of until and while

In a while construct, the loop is executed as long as the condition is fulfilled/ true. In until construct, the loop is executed as long as the condition is unsatisfied/ false.

You can convert a until to while by negating the condition.

Examples:

```
i=1
while [ $i -le 5 ]
do
  echo "Number: $i"
  i=$((i + 1))
done
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

```
input=""
while [ "$input" != "yes" ]
do
echo "Type yes to continue:"
read input
done
echo "you typed yes. Moving on.."
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Type yes to continue:
yes
you typed yes. Moving on..
```

```
i=1
until [ $i -gt 5 ]
do
    echo "Number: $i"
    i=$((i + 1))
done
```

```
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Here, until the number is greater than 5, print the number

```
num=0
while [ "$num" -ne 6 ]
do
    echo "Guess the number:"
    read num
done
echo "Way to go!You guessed it!"
```

```
gayat@GSP-HP:~/SummerInternship$ chmod +x A.sh
gayat@GSP-HP:~/SummerInternship$ ./A.sh
Guess the number:
89
Guess the number:
0
Guess the number:
7
Guess the number:
6
Way to go!You guessed it!
```

```
gayat@GSP-HP:~
Number: 0
Number: 2
Number: 4
Number: 6
Number: 8
Number: 10
gayat@GSP-HP:~
```

```
i=0
until [ "$i" -gt 10 ]
do
    echo "Number: $i"
    i=$((i + 2))
done
```