User Input and Output in Shell Scripting

Definition: read is used to take input from the user in shell script. It waits for user to type value and then stores it to variable.

Syntax:

read variable_name # whatever the value typed by user is stored in variable

Example:

```
#!/bin/bash
echo “Enter your Name: ”
read name
echo “ Hello, $name! Welcome to shell scripting.”
```

Options or flags of read command:

1. **Adding a prompt Inline: -p**
   Instead of using echo, you can add a **prompt with -p**.

```
#!/bin/bash
read –p “Enter your age: “ age
echo “ you are $age years old.”
```

2. **Hide User Input : -s silent**
   Use -s (silent) to **hide input** when typing.

```
#!/bin/bash
read –sp “Enter your password: “ password
echo “ Password received.”
```

3. **Multiple input:**
   You can read multiple values at once.

```
#!/bin/bash
read -p "Enter your first name and last name: " fname lname
echo "First Name: $fname"
echo "Last Name: $lname"
```

Conditional Statements

Definition: Conditional statements allow a shell script to **decide what to do** based on whether a condition is **true or false**.

1. **Basic if statement:**

Syntax:

```
if [ condition ]
then
   commands
fi
```

Example:

```
#!/bin/bash

age=20

if [ $age -ge 18 ] ## -ge is flag for greater and equal to

then

   echo "You are an adult."

fi
```

2. **If-else statement:**

Syntax:

```
if [ condition ]
then
commands_if_true
else
commands_if_false
fi
```

Example:

```
#!/bin/bash
read -p "Enter your age: " age

if [ $age -ge 18 ]
then
   echo "You are eligible to vote."
else
   echo "You are not eligible to vote."
fi
```

3. If-elif-else statement:

Syntax:

```
if [ condition1 ]
then
   commands1
elif [ condition2 ]
then
   commands2
else
   commands3
fi
```

Example:

```
#!/bin/bash
read -p "Enter marks: " marks

if [ $marks -ge 90 ]
then
   echo "Grade: A"
elif [ $marks -ge 75 ]
then
   echo "Grade: B"
elif [ $marks -ge 50 ]
then
   echo "Grade: C"
else
   echo "Grade: Fail"
fi
```

Operators in conditions:

Comparison Operators (numers):

Operator	Meaning
-eq	equal
-ne	not equal
-gt	greater than
-lt	less than
-ge	greater than or equal
-le	less than or equal

Example:

```
#!/bin/bash
a=10
b=20

if [ $a -eq $b ]
then
   echo "a is equal to b"
fi

if [ $a -ne $b ]
then
   echo "a is not equal to b"
fi

if [ $a -lt $b ]
then
   echo "a is less than b"
fi

if [ $b -ge 10 ]
then
   echo "b is greater than or equal to 10"
fi
```

String Operators:

Operator	Meaning
=	equal
!=	not equal
-z	string is empty
-n	string is not empty

Example:

```
#!/bin/bash

name="Shubham"

empty=""

if [ "$name" = "Shubham" ]

then

   echo "Name is Shubham"
```

```
fi

if [ "$name" != "DevOps" ]

then

   echo "Name is not DevOps"

fi

if [ -z "$empty" ]

then

   echo "The variable 'empty' has no value"

fi

if [ -n "$name" ]

then

   echo "The variable 'name' is not empty"

fi
```

File Test operators:

Operator	Meaning
-f file	File exists and is a regular file
-d file	Directory exists
-e file	File/directory exists
-r file	File is readable
-w file	File is writable
-x file	File is executable

Example:

```
#!/bin/bash
file="/etc/passwd"

dir="/home"

if [ -f "$file" ]

then

   echo "$file exists and is a regular file"

fi

if [ -d "$dir" ]

then

   echo "$dir exists and is a directory"

fi

if [ -e "$file" ]

then

   echo "$file exists (file or directory)"

fi

if [ -r "$file" ]
```

```
then
   echo "$file is readable"
fi
if [ -w "$file" ]
then
   echo "$file is writable"
fi
if [ -x "$file" ]
then
   echo "$file is executable"
else
   echo "$file is NOT executable"
fi
```

LOOPS in Shell Scripting

1. For loop:

Definition:

The for loop is used to go through every list of items, numbers or executable files and commands for each item.

Syntax:

```
for variable in list
do
   commands
done
```

Example:

```
#!/bin/bash
directory="/etc"

for file in "$directory"/*
do
   echo "File: $file"
done
```

2. While loop:

Definition:

The while loop executes commands repeatedly as long as a condition is true.

Syntax:

```
while [ condition ]
do
   commands
done
```

Example:

```
#!/bin/bash
while ! systemctl is-active --quiet nginx
do
   echo "Nginx is not running. Waiting..."
   sleep 5
done
```

```
echo "Nginx is running!"
```

3. Until loop:

Definition:

keep going until condition becomes true, An until loop executes a block of code **as long as the condition is false**. It stops when the condition becomes true.

Syntax:

```
until [ condition ]
do
   # commands to execute
done
```

Example:

```
#!/bin/bash

file="/tmp/config.txt"

# Loop until the file exists
until [ -f "$file" ]
do
   echo "Waiting for $file to be created..."
  #touch “$file” # line to create file
   sleep 2
done

echo "File $file is available!"
```

Functions in Shell Scripting

Definition:

A function is a block of code that performs a specific task and can be called multiple times in a script

Benefits:

- Avoids code repetition
- Makes scripts modular
- Improves readability

Syntax:

```
function_name() {
   commands
}
### another way to declare the function
function function_name {
   commands
}
```

Rules:

Function name rules:

- Must start with a **letter or underscore** (not a number).
- Should not contain spaces.
- Avoid using bash keywords (like if, then, fi, for).

No parentheses needed unless you are using the () form.

- Example: my_func { ... } is invalid.
- Correct: my_func() { ... } valid

Functions must be defined before calling them in the script (order matters).

Arguments are passed like script arguments:

- $1, $2, ... → positional parameters inside the function.
- $@ → all arguments.
- $# → number of arguments.

Example:

```
#!/bin/bash

greet() {
   echo "Hello, Shubham! Welcome to Shell Scripting."
}

# Call the function
greet
```

Function with Arguments

- $1, $2, … represent **positional arguments** passed to the function.

Example:

```
#!/bin/bash


greet_user() {

   echo "Hello, $1! You are learning $2."

}

greet_user "Shubham" "DevOps"
```

Returning Values from Functions

- Use return to return a **numeric value** (0–255).
- For strings, you **echo** the value and capture it.

Example: numeric return

```
#!/bin/bash

add_numbers() {

   sum=$(( $1 + $2 ))

   return $sum

}

add_numbers 5 10

echo "Sum returned: $?"
```

```
#!/bin/bash


add_numbers() {
   sum=$(( $1 + $2 ))
   echo $sum   # print the result
}
result=$(add_numbers 5 10)   # capture the echo output
echo "Sum is: $result"
```

Example: string return

```
#!/bin/bash


get_username() {
   echo "$USER"
}


username=$(get_username)
echo "Current user is $username"
```

Use cases in DevOps:

- **Check service status** (nginx, docker, etc.)
- **Backup automation** — define a function for each folder
- **User management** — create a function to add users
- **Deployments** — modularize deployment steps for different environments

Command Line Arguments

Definition: Command line argument are values passed to script when it is executed to make script dynamic and reusable

Variable	Meaning
$0	Script name
$1	First argument
$2	Second argument
$3	Third argument
$#	Number of arguments passed
$@	All arguments as separate strings
$*	All arguments as a single string
$$	Process ID of the script
$?	Exit status of last command

Example:

```
#!/bin/bash

echo "Script name: $0"

echo "First argument: $1"

echo "Second argument: $2"

echo "Total arguments: $#"
```

Case Statement:

Definition: Case statement is a conditional statement used to execute commands based on the value of a variable or expression.

Why we use:

Multiple condition handling

Syntax:

```
case variable in
   pattern1)
      commands
      ;;
   pattern2)
      commands
      ;;
   *)
      default_commands
      ;;
esac
```

Example: write a script to check the status of common services (`nginx`, `docker`, `mysql`).

```
#!/bin/bash


service=$1   # take first argument as input


case $service in
   nginx)
      systemctl status nginx
      ;;
   docker)
```

```
    systemctl status docker
    ;;
  mysql)
    systemctl status mysql
    ;;
  *)
    echo "Service not recognized. Please use: nginx, docker, or mysql."
    ;;
esac
```