Shell Programming Study Material

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

1. Shell Variables

- Shell variables are placeholders for storing data like text or numbers.
- You can define variables without a type declaration in bash.

Syntax:

```
bash

variable_name="value"
```

Example:

```
name="Anna"

echo "Hello, $name"
```

- Access variable value by prefixing with `$`.
- No spaces around `=` when assigning.

2. Environmental Variables

- These variables are defined by the system or user and are accessible by all processes.
- Examples: `PATH`, `HOME`, `USER`

To view environment variables:

```
printenv
```

To set an environment variable in your current shell:

```
export VAR_NAME="value"
```

Example:

```
export PATH=$PATH:/custom/path

echo $PATH
```

3. Shell Script Commands

- Basic commands used inside scripts include:
  - `echo` - Print output
  - `read` - Take input from user
  - `cd`, `ls`, `pwd` - Directory operations
  - `if`, `for`, `while` - Flow control

A simple script example:

```
#!/bin/bash

echo "Enter your name:"

read name

echo "Hello, $name"
```

Save this as `greet.sh`, make it executable (`chmod +x greet.sh`), then run `./greet.sh`.

4. Arithmetic Operations

- Use `let`, `expr`, or `$(())` for arithmetic.

Example using `$(( ))`:

```
a=10

b=5

sum=$((a + b))

echo "Sum is $sum"
```

Supported operators: `+`, `-`, `*`, `/`, `%`, `**` (power).

1. Using expr

```
#!/bin/bash
```

```
a=10
b=5

sum=$(expr $a + $b)
echo "Sum: $sum"

diff=$(expr $a - $b)
echo "Difference: $diff"

prod=$(expr $a \* $b)
echo "Product: $prod"

quot=$(expr $a / $b)
echo "Quotient: $quot"

mod=$(expr $a % $b)
echo "Modulus: $mod"
```

2. **Using $(( )) (Recommended)**

```
#!/bin/bash
a=20
b=4

echo "Sum: $((a + b))"
echo "Difference: $((a - b))"
echo "Product: $((a * b))"
echo "Quotient: $((a / b))"
echo "Modulus: $((a % b))"
```

3. **Using let**

```
#!/bin/bash
a=15
b=3

let sum=a+b
let diff=a-b
let prod=a*b
let quot=a/b
let mod=a%b

echo "Sum: $sum"
echo "Difference: $diff"
echo "Product: $prod"
echo "Quotient: $quot"
echo "Modulus: $mod"
```

4. **Using bc for Floating Point Arithmetic**

```
#!/bin/bash
a=10.5
b=4.2

sum=$(echo "$a + $b" | bc)
diff=$(echo "$a - $b" | bc)
prod=$(echo "$a * $b" | bc)
quot=$(echo "scale=2; $a / $b" | bc)

echo "Sum: $sum"
echo "Difference: $diff"
```

```
echo "Product: $prod"

echo "Quotient: $quot"
```

Method	Supports Integers	Supports Floats	Notes
expr	✅	❌	Legacy tool
$(( ))	✅	❌	Preferred for integers
let	✅	❌	Internal Bash command
bc	✅	✅	Best for floating-point math

5. Command Substitution

Command substitution in shell scripting is a mechanism where the shell executes a specified command and then replaces the command itself with its standard output. This allows the output of one command to be used as an argument or part of another command, or assigned to a variable.

Allows use of output of a command as a variable value or in expressions.

Syntax:

```
var=$(command)
```

or

```
var=`command`
```

Example:

```
current_date=$(date)
echo "Today is $current_date"
```

Example 1: Store Date in a Variable

```
now=$(date)
echo "Current date and time: $now"
```

Output:

```
Current date and time: Mon Jul 28 19:30:45 IST 2025
```

Example 2: Count Number of Files in a Directory

```
file_count=$(ls | wc -l)
echo "Number of files: $file_count"
```

Example 3: Get the Current Logged-in User

```
user_name=$(whoami)
echo "You are logged in as: $user_name"
```

Example 4: Nesting Substitution

```
echo "Today is: $(date +%A), and user is: $(whoami)"
```

Example 5: Use Inside an If Condition

```
if [ "$(uname)" = "Darwin" ]; then
    echo "You are on macOS."
else
    echo "You are on Linux or another OS."
fi
```

Example 6: Use in a For Loop

```
for file in $(ls *.txt); do
    echo "Processing $file"
done
```

Syntax	Description
command	Legacy style
$(command)	Modern, preferred style
Nesting	Allowed: $(command $(nested_command))

6. Command Line Arguments

Command-line arguments in shell scripts are values or **parameters passed** to a script when it is executed from the terminal. These arguments allow for dynamic and versatile script behavior, as they can be used to provide input, specify options, or customize the script's operation without needing to modify the script's code directly.

- `$0` - script name
- `$1, $2, ...` - positional arguments passed to the script
- `$#` - number of arguments
- `$@` or `$*` - all arguments

Example script using arguments:

```
#!/bin/bash

echo "Script name: $0"

echo "First argument: $1"

echo "Number of arguments: $#"


./args_demo.sh hello world
```

When you run a shell script like:

./myscript.sh arg1 arg2 arg3

Inside the script:

- $0 is the script name (myscript.sh)
- $1 is arg1
- $2 is arg2
- $3 is arg3
- $@ is all arguments
- $# is the number of arguments

Example 1: Basic Usage

args_demo.sh

```
#!/bin/bash

echo "Script Name: $0"
echo "First Arg: $1"
echo "Second Arg: $2"
echo "Total Args: $#"
```

Run it:

```
./args_demo.sh hello world
```

Output:

```
Script Name: ./args_demo.sh
First Arg: hello
Second Arg: world
Total Args: 2
```

Example 2: Loop Over All Arguments

```
#!/bin/bash

echo "All Arguments:"
for arg in "$@"
do
  echo "$arg"
done
```

Example 3: Check Argument Count

```
#!/bin/bash

if [ "$#" -ne 2 ]; then
  echo "Usage: $0 <source> <destination>"
  exit 1
fi

echo "Copying from $1 to $2"
cp "$1" "$2"
```

Run:

```
./copy.sh file.txt /Users/demo/
```

Example 4: Use Shift to Process Arguments

```
#!/bin/bash

while [ "$#" -gt 0 ]; do
  echo "Argument: $1"
  shift
done
```

Example 5: Named Arguments with Flags

```
#!/bin/bash

while [[ $# -gt 0 ]]; do
  case $1 in
    -f|--file) FILE="$2"; shift ;;
    -d|--dir) DIR="$2"; shift ;;
    *) echo "Unknown option: $1"; exit 1 ;;
  esac
  shift
done

echo "File: $FILE"
```

```
echo "Directory: $DIR"
```

Run it like:

```
./myscript.sh -f input.txt -d /Users/demo/
```

Run with: `./script.sh arg1 arg2`

Variable	Meaning
`$0`	Script name
`$1…$9`	First to ninth arguments
`$@`	All arguments as separate words
`$*`	All arguments as one string
`$#`	Number of arguments
`shift`	Shifts arguments to the left

7. Conditional Execution

- Commands chained with `&&` (AND) or `||` (OR)

Example:

```
mkdir newdir && cd newdir
```

`cd` runs only if `mkdir` succeeds.

8. if Statement Format

```
if [ condition ]; then
   # commands
elif [ condition ]; then
   # commands
else
   # commands
fi
```

Example:

```
#!/bin/bash
echo "Enter a number:"
read num

if [ $num -gt 0 ]; then
   echo "Positive number"
elif [ $num -lt 0 ]; then
   echo "Negative number"
else
   echo "Zero"
fi
```

9. Test - String Comparison

String comparisons are fundamental for controlling script flow based on textual data.

- Use `[ ]` with operators:
  - `-z str` - string is empty
  - `-n str` - string is not empty
  - `str1 = str2` - strings are equal
  - `str1 != str2` - strings are not equal

Example:

```
read input
if [ -z "$input" ]; then
   echo "Empty input"
else
   echo "Input is not empty"
fi
```

1. Equality (= or ==)

```
#!/bin/bash
str1="hello"
str2="hello"

if [ "$str1" = "$str2" ]; then
  echo "Strings are equal"
else
  echo "Strings are not equal"
fi
```

You can also use == with [[ ]]:

```
if [[ "$str1" == "$str2" ]]; then
  echo "Equal"
fi
```

2. Not Equal (!=)

```
str1="hello"

str2="world"


if [ "$str1" != "$str2" ]; then

  echo "Strings are not equal"

fi
```

3. Check if String is Empty or Not Empty:

```
str=""


if [ -z "$str" ]; then

  echo "String is empty"

fi
```

Not Empty:

```
str="data"


if [ -n "$str" ]; then

  echo "String is not empty"
```

```
fi
```

4. Lexicographical Comparison (<, >)

```
str1="apple"
str2="banana"

if [[ "$str1" < "$str2" ]]; then
  echo "$str1 comes before $str2"
fi
```

Use [[...]] for < and > (not [...]), otherwise you'll get an error due to redirection interpretation.

5. Case-Insensitive Comparison

```
str1="HELLO"
str2="hello"

if [[ "${str1,,}" == "${str2,,}" ]]; then
  echo "Strings match (case-insensitive)"
fi
```

6. Using case Statement (Pattern Matching)

```
read -p "Enter choice (yes/no): " input
```

```
case "$input" in
 [Yy][Ee][Ss]) echo "You said yes";;
 [Nn][Oo]) echo "You said no";;
 *) echo "Invalid input";;
esac
```

10. The Case Statement

The case statement is used to match a value against multiple patterns. It's like a switch-case in other programming languages (C, Java, etc.).

- Provides a way to select commands based on pattern match.

Syntax:

```
case expression in
   pattern1)
      commands ;;
   pattern2)
      commands ;;
   *)
      default commands ;;
esac
```

Example:

```
read choice
case $choice in
   1) echo "Option 1 selected";;
   2) echo "Option 2 selected";;
   *) echo "Invalid option";;
esac
```

Example 1: Day Checker

```
#!/bin/bash

read -p "Enter a day: " day

case "$day" in
  "Monday") echo "Start of the week." ;;
  "Friday") echo "Almost weekend!" ;;
  "Sunday") echo "Relax, it's Sunday." ;;
  *) echo "Just another day." ;;
esac
```

Example 2: Menu-Driven Program

```
#!/bin/bash

echo "1. Show Date"
```

```
echo "2. Show Calendar"

echo "3. Show Current Directory"

echo "4. Exit"


read -p "Enter choice [1-4]: " choice


case "$choice" in

  1) date ;;

  2) cal ;;

  3) pwd ;;

  4) echo "Goodbye!"; exit ;;

  *) echo "Invalid choice" ;;

esac
```

Example 3: Case-Insensitive Matching

```
#!/bin/bash


read -p "Do you want to continue (yes/no)? " answer


case "$answer" in

  [Yy][Ee][Ss]) echo "Continuing..." ;;

  [Nn][Oo]) echo "Exiting..."; exit ;;

  *) echo "Invalid input. Please type yes or no." ;;

esac
```

When to Use a case Statement?

- Replacing multiple if-elif-else for better readability
- Menu-driven scripts
- Pattern matching (yes/no, file extensions, numeric choices)
- Cleaner handling of multiple string cases

11. While Statement

- Executes commands repeatedly as long as the condition is true.

Syntax:

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

Example:

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

12. Break & Continue Statement

- `break` exits the nearest loop.
- `continue` skips to the next iteration of the loop.

Example:

```
for i in {1..10}
do
   if [ $i -eq 5 ]; then
      break
   fi
   echo $i
done
```

13. Until Statement

- Similar to `while` but runs until the condition becomes true.

Syntax:

```
until [ condition ]
do
   commands
done
```

Example:

```
count=1
until [ $count -gt 5 ]
do
   echo "Count: $count"
```

```
    ((count++))
done
```

14. Shell Functions

- Functions allow grouping commands to reuse code.

Syntax:

```
function_name() {
    commands
}
```

Example:

```
greet() {
    echo "Hello, $1"
}

greet "Anna"
```

15. Using Arrays

- Arrays store multiple values indexed by numbers.

Defining and accessing arrays:

```
my_array=(apple banana mango)
echo ${my_array[0]}  # apple
```

Loop over array elements:

```
for item in "${my_array[@]}"
do
   echo $item
done
```

SFTP

SFTP stands for **Secure File Transfer Protocol**. It is a secure and efficient method for transferring files in the Linux environment. It is a command-line tool used in mostly Linux, UNIX-based operating systems.

It is an extension of SSH(Secure Shell) and encrypts the command and data during transmission. It is a protocol for securely transferring files from a remote server to a local machine. before SFTP, FTP was used to transfer files but it was unsecured. An attacker can read the communication between a remote server and a local machine.

Advantages of SFTP

- SFTP ensures that it encrypts the data and commands.
- SFTP checks the data integrity, and whether any data is tampered with or lost during transmission.
- SFTP checks the user authentication with a valid username and password.
- SFTP supports most of the operating systems, which makes them portable to use.
- SFTP is user-friendly, it is easy to use either in command line or graphical mode.
- SFTP allows their command for scripting and automation.
- SFTP allows multiple users to access and transfer the file securely.

Option	Description
-b batchfile	Gives the name of a batch file that contains SFTP commands.
-B buffer_size	Determines the size of the file transmission buffer.
-P port	Specifies the remote host's port to use for the connection.
-v	For verbose mode, it generate the detailed report
-h	It displays the helps information for particular command
pwd	Prints the current working directory
cd directory_name	Change the directory
get remote_file [local_path]	It is used to download the file from remote server
put local_file [remote_path]	It is used to upload the file in remote server
rm filename	It removes a file from the distant server
mkdir directory_name	It create a directory in remote server
rmdir directory_name	It remove the directory from remote server
chmod permisssions filename	It sets the permissions on files and directories

Summary Table of Commands and Syntax

Topic	Syntax/Command Example
Variable Assignment	`var="value"`
Environment Variable	`export VAR="value"`
Arithmetic	`sum=$((a + b))`
Command Substitution	`result=$(command)`
Command Line Arguments	`$1, $2, $#, $@`
if Statement	`if [ condition ]; then ... fi`
String Test	`[ -z "$var" ]` or `[ "$a" = "$b" ]`
Case Statement	`case expr in pattern) commands ;; esac`
While Loop	`while [ condition ]; do commands; done`
Break & Continue	`break` and `continue` inside loops
Until Loop	`until [ condition ]; do commands; done`
Functions	`func() { commands; }`
Arrays	`arr=(val1 val2)` and `${arr[index]}`

Use case:

• Create shall script to read file line by line

• Concat files

• Split files

• Touch a file

• Check File Size

du -h filename.txt

stat filename.txt

• Record count, word count, AWK, SED operations

• Head & Tail commands

• STFP

• Connect to DB and execute DDL, DML