Chapter 1 - Bash Scripting - Notes

Category	Chapter 1 Bash Scripting

1. Introduction to the Linux Shell

- **Shell** = Interface between the user and the OS.
  - Takes commands, passes them to the kernel, returns output.
- **Types of shells**:
  - `sh` → Bourne shell, early standard.
  - `csh` → C shell, syntax similar to C.
  - `ksh` → Korn shell, adds scripting improvements.
  - `tcsh` → Enhanced C shell.
  - `bash` → Bourne Again Shell (most common).
- **Why Bash?**
  - Default on most Linux systems.
  - Portable across distributions.
  - Acts as both **command line interpreter** and **scripting language**.
- **Benefits of scripting**:
  - Automate tasks (backups, cleanup, monitoring).
  - Save time (no repetitive typing).
  - Combine tools with pipes and scripts.
  - Customization with aliases and flags.

2. Writing and Running Your First Script

Steps:

1. Create a file with `.sh` extension.
2. Add **shebang** line:

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

   - Tells system which interpreter to use.
   - Must be first line.
3. Add commands inside. Example:

   ```
   echo "Hello World"
   ```
4. Make file executable:

   ```
   chmod 700 hello.sh
   ```
5. Run script:

   ```
   ./hello.sh
   ```

3. Core Linux Commands for Scripting

- **File management**:
  - `ls` → list files.
  - `cp file1 file2` → copy.
  - `mv old new` → move/rename.
  - `rm file` → remove.
  - `mkdir folder` → make directory.

- **Navigation**:
  - `pwd` → print current directory.
  - `cd dir` → change directory.
- **Viewing files**:
  - `cat file` → display file.
  - `head -n 10 file` → first 10 lines.
  - `tail -f file` → last lines, update live.
  - `wc -l file` → count lines.
- **Search & filter**:
  - `grep "pattern" file` → search.
  - `sort file` → sort lines.
  - `uniq file` → remove duplicates.
  - `cut -d"," -f1 file.csv` → extract columns.

4. Commands, Arguments, and Exit Status

- **Command structure**:

  ```
  command [options] [operands]
  ```

  Example:

  `ls -ltr /bin /usr`

  - `ls` = command.
  - `ltr` = options.
  - `/bin /usr` = operands (targets).
- **Exit status (`$?`)**:
  - `0` → success.

- Non-zero → error.
- Useful in scripts for conditionals.

5. Redirection and Pipelines

- **Standard streams**:
  - `stdin (0)` → input.
  - `stdout (1)` → output.
  - `stderr (2)` → errors.
- **Redirect operators**:
  - `>` → overwrite file.
  - `>>` → append.
  - `<` → take input from file.
- **Here-docs** (multi-line input):

```
cat << EOF
Hello
World
EOF
```

- **Pipes (`|`)**:
  - Send output of one command to another.
  - Example:

```
ls | grep ".txt"
```

6. Variables

- No type declarations → always **string**.
- Define:

```
name="Arnav"
```

- Use:

```
echo $name
```

- **Parameter expansion**:

```
echo ${name}123
```

- **Quotes**:
  - `" "` → expands variables.
  - `' '` → literal.

```
var="CS"
echo "Course: $var"   # expands
echo 'Course: $var'   # literal
```

- **Scope**:
  - Global by default.
  - Use `local` inside functions.
  - `export VAR=value` → makes it available to child scripts.

7. Input and Expansions

- **User input**:

```
echo -n "Enter name: "
read user
```

```
echo "Hello $user"
```

- **Expansions**:
  - **Variable** → `${var}`
  - **Brace** → `{1..5}` expands to 1 2 3 4 5
  - **Tilde** → `~` expands to home dir.
  - **Command** → `$(date)` stores output.
  - **Arithmetic** → `$((2+3))` → 5
  - **Filename globbing**:
    - matches anything.
    - `?` matches single char.
    - `[abc]` matches a/b/c.

8. Conditionals

Syntax:

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

- **Numeric tests**:
  - `eq, -ne, -lt, -gt, -le, -ge`
- **String tests**:
  - `=, !=, -n (not empty), -z (empty)`

- **File tests**:
  - `e` exists, `f` file, `d` directory.
  - `r/-w/-x` → read/write/execute permissions.

9. Logical Operators

- **AND**: `[ cond1 -a cond2 ]` or `[ cond1 ] && [ cond2 ]`
- **OR**: `[ cond1 -o cond2 ]` or `[ cond1 ] || [ cond2 ]`
- **NOT**: `!`

10. Case Statement

```
case $var in
  1) echo "One";;
  2|3) echo "Two or Three";;
  *) echo "Other";;
esac
```

- Useful for menu-based scripts.

11. Loops

For loop

```
for item in a b c; do
  echo $item
done
```

Range

```
for i in {1..5}; do echo $i; done
```

C-style

```
for ((i=0;i<5;i++)); do
  echo $i
done
```

While

```
while [ $i -lt 5 ]; do
  ((i++))
done
```

Until

```
until [ $i -eq 5 ]; do
  ((i++))
done
```

Break & Continue

- `break` → exits loop.
- `continue` → skips to next iteration.

12. Arrays

- **Indexed arrays**:

  ```
  arr=("one" "two" "three")
  echo ${arr[0]}
  ```

- **Associative arrays**:

```
declare -A colors
colors[apple]="red"
echo ${colors[apple]}
```

- Iteration:

```
for i in "${arr[@]}"; do
  echo $i
done
```

13. Functions

- **Definition**:

```
myfunc() {
  echo "Hello $1"
}
```

- **Return values**:
  - `echo` + command substitution:

    ```
    result=$(myfunc arg)
    ```

  - `$?` → last return code (integer only).
- **Local variables**:

```
myfunc() {
  local x=5
}
```

14. Special Parameters

- `$0` → script name.
- `$1, $2 ...` → arguments.
- `$#` → number of args.
- `$*` → all args as one string.
- `$@` → all args as separate strings.
- `$?` → exit code of last command.
- `$$` → process ID of current script.

15. Debugging

- Add flags in shebang:
  - `v` → print lines as read.
  - `x` → print commands with expansion.
- Example:

```
#!/bin/bash -x
```

16. Programming vs Scripting

- **Programming languages**:
  - Compiled → faster.
  - Strong typing, libraries.
  - Good for performance-heavy apps.
- **Scripting**:
  - Interpreted → slower but easier to modify.

    - Quick prototyping.
    - Great for automation.
- **Best practice**:
    - Use compiled programs as building blocks.
    - Wrap with scripts for automation.