
Day 8 - Shell Scripting & Commands

1. Working with Arrays in Bash

In Bash, I can create arrays using indexed positions. Here's an example:

```
a[0]="zara"
a[1]="askdhkasdh"
a[2]="adakjhsdj"
```

To access elements, I use:

```
echo "${a[0]}"  # Output: zara
echo "${a[1]}"  # Output: askdhkasdh
```

Bash arrays are zero-indexed, so `a[0]` refers to the first element.

---

2. Conditional Statements in Bash

Bash supports different conditional checks using `if` statements.

Checking Balance and Withdrawal Conditions

I set up some variables:

```
balance=500
withdrawl=1200
daily_limit=1000
account_type="savings"
description=""
```

Equality and Comparison Operators

`-eq` (Equal to):

```
if [ $balance -eq 5000 ]; then
  echo "Balance is exactly 5000"
fi
```

- This checks if the balance is exactly 5000.

`-ne` (Not equal to):

```
if [ $withdrawl -ne 1000 ]; then
  echo "Withdrawn amount is not 1000"
fi
```

- This ensures the withdrawal is different from 1000.

`-gt` (Greater than), `-le` (Less than or equal to):

```
if [ $balance -gt $withdrawl ]; then
  echo "You have a valid balance to withdraw money"
fi
```

- This checks if I have sufficient balance to withdraw.

Logical AND (`-a`) & OR (`-o`) Operators

```
if [ $withdrawl -le $balance -a $withdrawl -le $daily_limit ]; then
  echo "Transaction approved"
else
  echo "Transaction not approved"
fi
```

`-a` (AND) ensures both conditions are met for transaction approval.

```
if [ $withdrawl -le $balance -o $balance -ge 500 ]; then
  echo "Customer is valuable to the bank"
fi
```

- `-o` (OR) checks if at least one condition holds true.

Logical NOT (`!`) and Extended Conditions (`[[ ... ]]`)

```
if [[ ! $withdrawl -le $balance || $balance -ge 500 ]]; then
  echo "Customer is valuable to the bank"
```

```
fi
```

- Here, `!` negates the condition.

String Comparisons

Checking if a variable contains a specific string:

```
if [ "$account_type" = "savings" ]; then
  echo "This is a savings account"
fi
```

-
    - `=` checks for string equality.
    - `!=` checks for inequality.

Checking if a variable is empty:

```
if [ -z "$description" ]; then
  echo "Description is not provided"
fi
```

- `-z` checks if the string is empty.

---

3. User Input in Bash

Reading User Input

I can prompt users for input using `read`.

```
read -t 5 -p "Quick 5 sec: " pin
```

- `-t 5` sets a timeout of 5 seconds for input.

```
echo "Enter your name"
read name
echo "$name"
```

This stores user input in the `name` variable and echoes it.

Reading Multiple Inputs

```
read -p "Enter account number and password: " acn password
echo $acn
echo $password
```

- `-p` allows me to display a prompt while reading input.

Reading Sensitive Input (Silent Mode)

```
read -s -p "Enter password: " p
```

- `-s` hides user input, useful for passwords.

---

4. Case Statements in Bash

Instead of multiple `if-else` statements, I use a `case` statement for cleaner code.

```
read -p "Enter selection [1-3]: " selection
case $selection in
  1) accounttype="checking"; echo "You have selected checking";;
  2) accounttype="saving"; echo "You have selected saving";;
  3) accounttype="current"; echo "You have selected current";;
  *) accounttype="random"; echo "Random selection";;
esac
```

- Each case pattern ends with `)`, and `;;` marks the end of a case block.
- `*` is the default case (similar to `else`).

---

5. Using `grep` for Searching Text

The `grep` command helps me search for specific patterns in a file.

Basic Search

```
grep "selection$" case.sh
```

- `$` ensures the search term appears at the end of a line.

```
grep -Ril "selection" case.sh
```

- `-R` (Recursive): Searches in subdirectories.
- `-i` (Ignore case): Case-insensitive search.
- `-l` (List files): Shows only filenames containing the pattern.

Using Character Classes and Wildcards

Find lines with any digit (`[0-9]`):

```
grep "[0-9]" case.sh
```

-

Find lines with letters (`[a-zA-Z]`):

```
grep "[a-zA-Z]" case.sh
```

-

Find lines containing vowels (`[aeiou]`):

```
grep "[aeiou]" case.sh
```

-

Using `*` (matches zero or more occurrences):

```
grep "s*n" case.sh
```

This matches "sn", "ssn", "sssn", etc.

```
grep "se*n" case.sh
```

Matches "sn", "sen", "seen", etc.

```
grep "selecti*n" case.sh
```

- Matches "selection", "selectiion", "selectiiion", etc.

Using `.` (matches any single character):

```
grep "sel.n" case.sh
```

Matches "selan", "selbn", "selxn", etc.

```
grep "selicti.n" case.sh
```

- Matches "selection", "selictiyn", etc.

Summary of Options Used

Command	Option	Purpose
`read`	`-p`	Prompt message before input
`read`	`-t`	Set timeout for input
`read`	`-s`	Hide input for passwords
`grep`	`-R`	Recursive search in directories
`grep`	`-i`	Case-insensitive search
`grep`	`-l`	Display filenames with matching text
`if`	`-eq`	Equal to (numeric comparison)
`if`	`-ne`	Not equal to (numeric comparison)
`if`	`-gt`	Greater than
`if`	`-le`	Less than or equal to
`if`	`-a`	Logical AND
`if`	`-o`	Logical OR

`if`	`-z`	Check if string is empty

These are some of the fundamental Bash commands and scripting techniques I have explored today!