Scripting Lesson 2: Shell Scripting Fundamentals

Topic / Session number	Scripting / Lesson 2
Objectives / Goals	1. Case Statement 2. Loops a. For Loop b. While Loop c. Until Loop 3. Break and Continue a. Break b. Continue 4. Environment Variables a. Modifying **.bashrc** to Persist Environment Variables, Aliases, or Functions 5. Streams and Pipe a. Streams b. Pipe 6. Debugging With -xv 7. Practice Questions

---

★ Objective 1: Case Statement

The `case` statement is used for multiple options and is similar to a series of `if-else` statements.

Syntax of `case`:

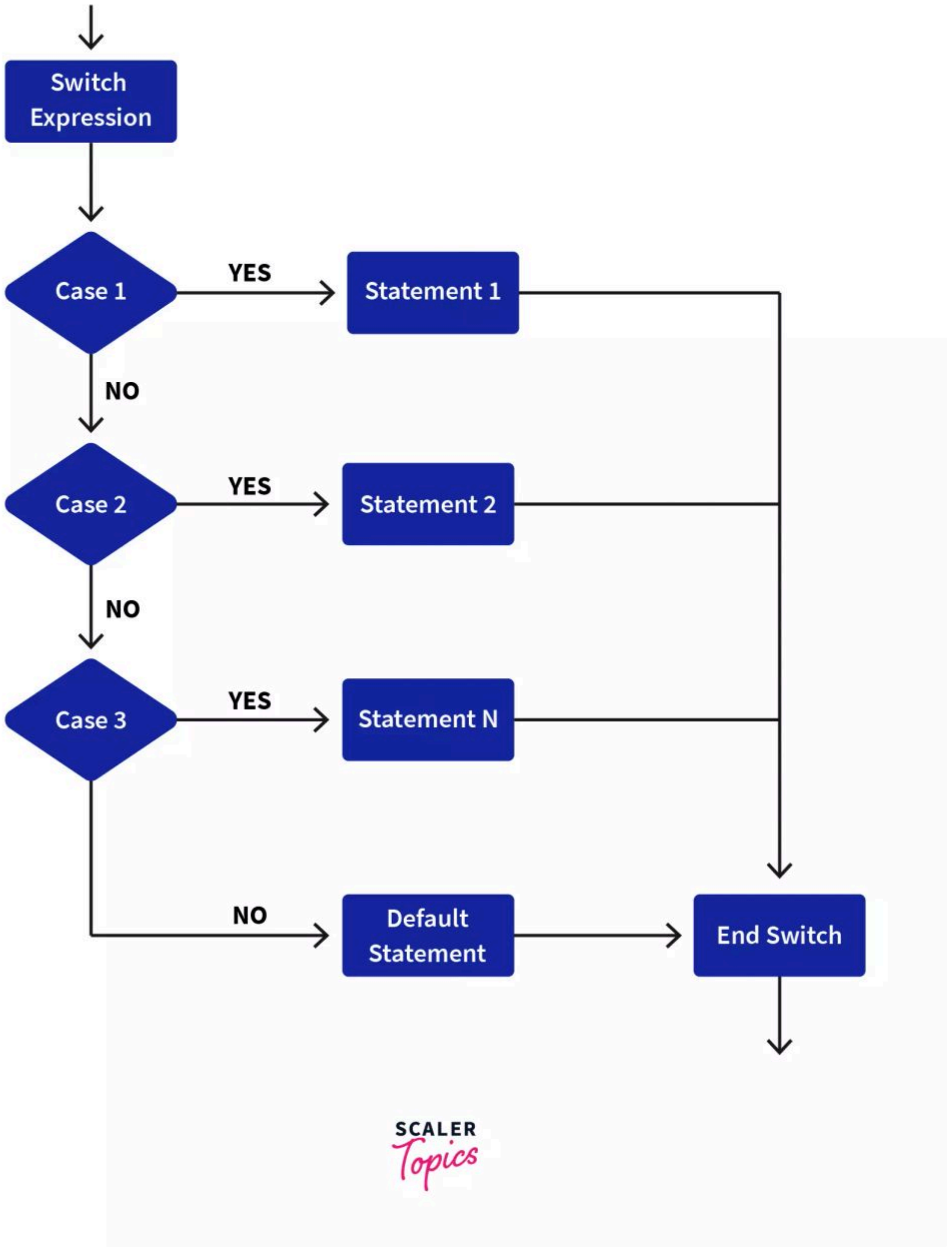
```bash
#!/bin/bash
echo "Enter a number between 1 and 3: "
read num

case $num in
  1)
    echo "You entered One."
    ;;
  2)
    echo "You entered Two."
    ;;
  3)
    echo "You entered Three."
    ;;
  *)
    echo "Invalid choice!"
    ;;
```

```
esac

```

Explanation: `case` is useful for making decisions based on multiple possible values.



Case is also known as “switch” in other programming languages

---

Objective 2: Loops

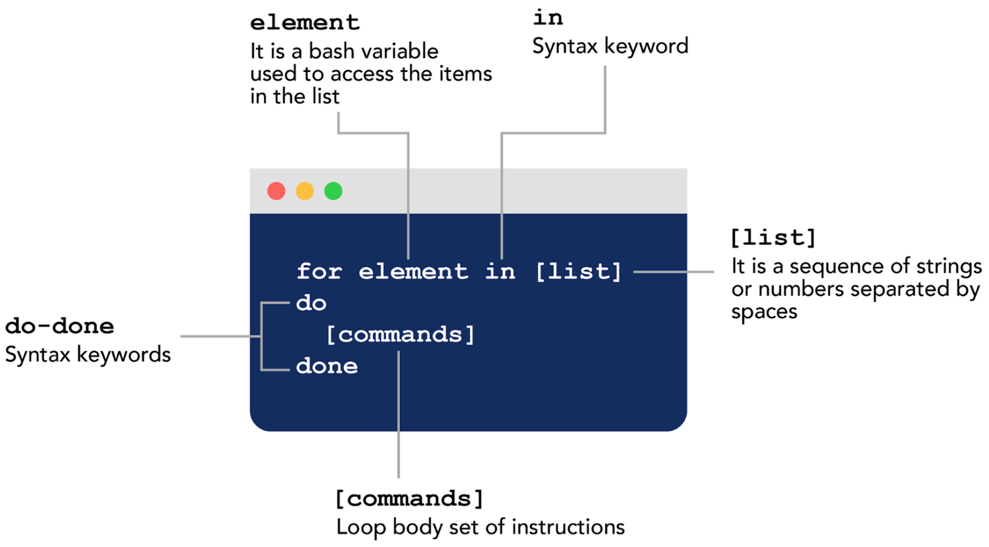
For Loop

Used to iterate over a list of items or a range.

```
#!/bin/bash
for i in {1..5}
```

```
do
  echo "Number $i"
done

```


element
It is a bash variable used to access the items in the list
in
Syntax keyword
[list]
It is a sequence of strings or numbers separated by spaces
do-done
Syntax keywords
[commands]
Loop body set of instructions

```
for element in [list]
do
  [commands]
done
```

Three-Expression For Loop

This type of for loop share a common heritage with the C programming language.

It is characterized by a three-parameter loop control expression; consisting of an initializer (EXP1), a loop-test or condition (EXP2), and a counting expression/step (EXP3).

```
for (( EXP1; EXP2; EXP3 ))
do
    command1
    command2
    command3
done
## The C-style Bash for loop ##
for (( initializer; condition; step ))
do
  shell_COMMANDS
done
```

Example that sets the counter “c” to 1 and condition to “c” is less than or equal to 5:

```
#!/bin/bash
for (( c=1; c<=5; c++ ))
do
   echo "Welcome $c times"
done
```

Output:

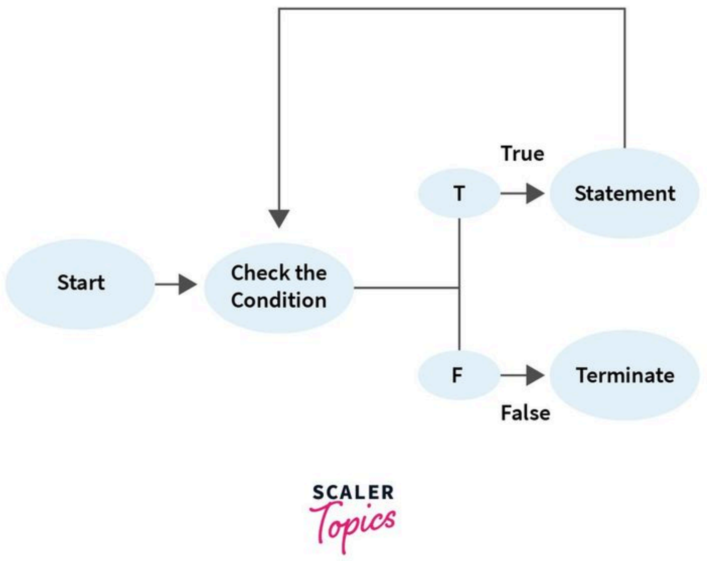
```
Welcome 1 times
Welcome 2 times
Welcome 3 times
Welcome 4 times
Welcome 5 times
```

While Loop

Executes a block of code as long as a condition is true.

```bash
#!/bin/bash
count=1
while [ $count -le 5 ]
do
  echo "Count $count"
  ((count++))
done

```

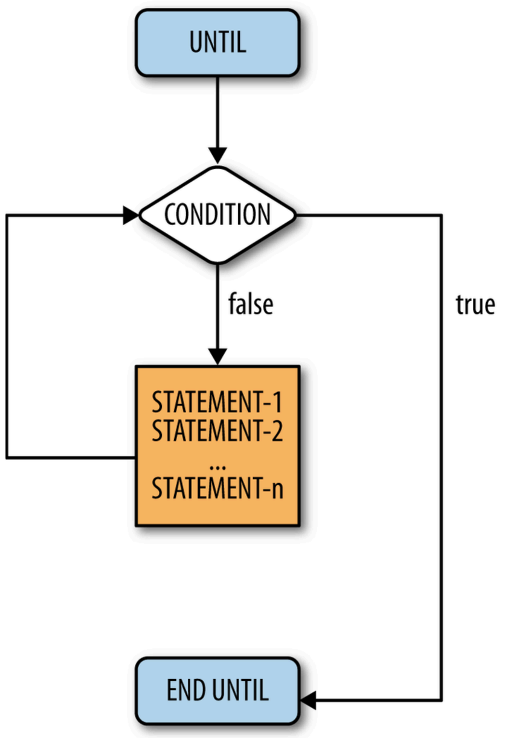


Until Loop

Executes a block of code until a condition is true.

```bash
#!/bin/bash
count=1
until [ $count -gt 5 ]
do
  echo "Count $count"
  ((count++))
done

```



> **Analogy**: Loops are like making a list of instructions you repeat over and over until a task is complete. For example, a repetitive chore like sweeping the floor until it's clean.

★ Objective 3: Break and Continue

Break

You can do early exit with break statement inside the for loop. You can exit from within a FOR, WHILE or UNTIL loop using break. General break statement inside the for loop:

```
for I in 1 2 3 4 5
do
  statement1   # Executed for all values of "I", up to a disaster-condition if any.
  statement2
  if (disaster-condition)
  then
    break       # Abandon the loop.
  fi
  statement3   # While good and, no disaster-condition.
done
```

The following shell script will go though all files stored in `/etc` directory. The for loop will be abandon when `/etc/resolv.conf` file found:

```
#!/bin/bash
# Count the number of DNS name servers in the /etc/resolv.conf if found
for file in /etc/*
do
    # Check if file is found
    if [ "${file}" == "/etc/resolv.conf" ]
    then
        countNameservers=$(grep -c nameserver /etc/resolv.conf)
        echo "A total of ${countNameservers} DNS nameservers defined in ${file}"
        break
    fi
done
```

Outputs:

```
Total of 2 DNS nameservers defined in /etc/resolv.conf
```

Continue

To resume the next iteration of the enclosing FOR, WHILE or UNTIL loop use continue statement.

```
for I in 1 2 3 4 5
do
  statement1  # Executed for all values of "I", up to a disaster-condition if any.
  statement2
  if (condition)
  then
    continue   # Go to next iteration of I in the loop and skip statement3
  fi
  statement3
done
```

This script makes a backup of all file names specified on command line. If a `.bak` file already exists, it will skip the cp command:

```
#!/bin/bash
FILES="$@"
for f in $FILES
do
    # If .bak backup file exists, read next file
    if [ -f ${f}.bak ]
    then
        echo "Skiping $f file..."
        continue  # read next file and skip the cp command
    fi
    # If we reached here it means no backup file exists,
    # just use the cp command to copy the file
    /bin/cp $f $f.bak
done
```

★ Objective 4: Environment Variables

Environment variables are used to store information that can affect the behavior of running processes.

Setting Environment Variables:

```
#!/bin/bash
export MY_VAR="Hello"
echo $MY_VAR

```

Unsetting Environment Variables:

```bash
#!/bin/bash
unset MY_VAR

```

Lifecycle of Environment Variables:

- **Temporary**: Only for the current session.
- **Permanent**: Can be added to configuration files like `.bashrc`.

Why Environment Variables Are Accessible from Child Processes

When a process (such as a shell) starts, it inherits environment variables from its **parent process**. These environment variables are stored in the environment of the process, and when the process creates a **child process**, the child inherits these variables.

> **Analogy**: Think of a **parent process** as a teacher in a school who gives instructions to students (the **child processes**). The teacher's instructions (environment variables) are passed down to the students, so they know what to do.

How Environment Variables are Passed to Child Processes

When a parent process (such as a terminal shell) spawns a child process, it passes along the environment variables. These variables are visible and can be accessed by the child process. This is important because many programs or scripts may need to know certain system configurations (like `PATH`, `USER`, etc.) or user-defined settings.

```bash
#!/bin/bash
export MY_VAR="Hello from parent process!"
echo "Inside parent process, MY_VAR = $MY_VAR"

# Creating a child process (e.g., running another script)
./child_script.sh

```

In `child_script.sh`:

```bash
#!/bin/bash
echo "Inside child process, MY_VAR = $MY_VAR"

```

Expected output:

```
Inside parent process, MY_VAR = Hello from parent process!
Inside child process, MY_VAR = Hello from parent process!

```

Explanation: In the above example, the environment variable `MY_VAR` is set in the parent process. When the child script (`child_script.sh`) is executed, it has access to the variable because the environment is inherited.

Modifying `.bashrc` to Persist Environment Variables, Aliases, or Functions

To make environment variables, aliases, or functions **persistent** (so they are available every time you open a new terminal), you can modify the `.bashrc` file in your home directory. The `.bashrc` file is sourced each time a new shell session starts, so any changes to this file will automatically be applied.

Environment Variables: You can add environment variables to `.bashrc` so that they are available to all future terminal sessions:

Open `.bashrc` with your preferred editor:

```
nano ~/.bashrc

```

Add the environment variable at the end of the file:

```
export MY_VAR="Persistent environment variable"

```

After saving the file, run:

```
source ~/.bashrc

```

This will apply the changes in the current session without needing to restart the terminal.

Aliases: An **alias** allows you to create shortcuts for commands. For example, you can define an alias for `ls -al` as `ll` to simplify your work.

Add the following line to `.bashrc`:

```
alias ll="ls -al"

```

After saving and sourcing the `.bashrc`:

```
source ~/.bashrc

```

Now, every time you type `ll` in the terminal, it will execute `ls -al`.

Functions: You can also define **functions** in `.bashrc` for reuse. For example, a function to greet the user:

```
greet() {
  echo "Hello, $1! Welcome to the terminal."
}

```

After saving and sourcing `.bashrc`, you can call the `greet` function in any terminal session:

```
greet Alice

```

Example of a Complete `.bashrc` Modification:

Here's an example of a `.bashrc` file with environment variables, aliases, and functions:

```
# .bashrc file

# Set environment variables
export MY_VAR="Persistent environment variable"
export PATH="$PATH:/new/directory"
```

```bash

# Aliases
alias ll="ls -al"
alias gs="git status"

# Functions
greet() {
    echo "Hello, $1! Welcome back."
}

# Source the file
source ~/.bashrc

```

Important Notes

- **Environment Variables**: Once you add an environment variable in `.bashrc` and source the file, the variable will persist across new terminal sessions. Any processes launched from this shell will inherit these variables.
- **Aliases**: Aliases are also a great way to speed up your workflow and reduce typing effort. For example, creating an alias for a command you use often, like `gs` for `git status`, can save time.
- **Functions**: Defining functions in `.bashrc` is useful for tasks you perform often. For example, if you find yourself running the same series of commands repeatedly, you can encapsulate them in a function.

⭐ Objective 5: Streams and Pipe

In Bash, streams and pipes are powerful tools that allow you to direct the flow of data from one command or process to another. Here's how they work:

Streams

Bash has three standard streams:

1. `stdin` (standard input, file descriptor 0): This is the **default input stream**, which provides input to commands.
2. `stdout` (standard output, file descriptor 1): This is the **default output stream** for a command.
3. `stderr` (standard error, file descriptor 2): This is the **default error output stream** for a command.

You can redirect these streams using the `<`, `>`, `>>`, `2>`, and `&>` operators.

- To redirect `stdout`, you use `>` to overwrite a file or `>>` to append:

  ```bash
  echo "Hello World" > output.txt  # Overwrites or creates output.txt with the content
  echo "Hello Again" >> output.txt # Appends to output.txt
  ```

- To redirect `stderr`, you use `2>`:

  ```bash
  ls non_existing_file 2> error.txt # Redirects the error message to error.txt
  ```

- To redirect both `stdout` and `stderr`, you can use `&>`:

  ```bash
  command &> output.txt # Both stdout and stderr will go to output.txt
  ```

Pipe

The pipe `|` operator is used to pass the output of one command as input to another command.

- Here's a simple example:

  ```
  cat file.txt | grep "search_term"
  ```

  In this command, the output of `cat file.txt` is passed as input to `grep "search_term"`, which filters the output for lines containing "search_term".
- You can chain multiple commands with pipes:

  ```
  cat file.txt | grep "search_term" | sort | uniq
  ```

Pipes work because they connect the `stdout` of the command on the left to the `stdin` of the command on the right. This is an example of inter-process communication and is a fundamental concept in Unix-like operating systems.

When you're using streams and pipes, remember that:

- Redirection can overwrite files, so use it cautiously.
- You can redirect `stdin` from a file to a command using `<`.
- You can use `tee` to output to both the terminal and a file.
- Use `xargs` if you need to build and execute command lines from standard input.

⭐ Objective 6: Debugging With -xv

Depending on the Bourne shell version you have, the error messages it gives can be downright useless. For instance, it might say just

```
End of file unexpected
```

Here are a few tricks to use to get a little more information about what's going on.

Use -xv

Start your script like this:

```
#!/bin/sh -xv
```

The `-xv` shows you what's happening as the shell reads your script. The lines of the script will be shown as the shell reads them. The shell shows each command it executes with a plus sign (`+`) before the command.

Note that the shell reads an entire loop (*for*, *while*, etc.) before it executes any commands in the loop.

If you want to run a script with debugging but you don't want to edit the script file, you can also start the shell explicitly from the command line and give the options there:

```
./script.sh -xv
```

Debugging output is usually pretty long, more than a screenful, so I pipe it to a pager like `less`. But the shell sends its debugging output to *stderr*, so I pipe both **stdout and stderr** to the pager.

```
./script.sh 2>&1 | less
```

★ Objective 7: Practice Questions

1. Write a script using a `case` statement to determine the day of the week based on a number input (1 for Monday, 2 for Tuesday, etc.).
2. Write a script that uses a `for` loop to print numbers from 1 to 10.
3. Create a script that checks if a directory exists, and if not, creates it.
4. Write a script that takes an environment variable as input and displays its value.
5. Create a script that runs a `while` loop and counts down from 10 to 1.
6. Create a script that uses environment variables. Set an environment variable, use it inside a script, and print its value.
7. Modify the `.bashrc`:
   a. Add an alias for a command you use frequently (e.g., `ll` for `ls -al`).
   b. Add a function to greet the user by name, e.g., `greet John`.
8. Create a script that modifies an environment variable. Write a script that changes the value of an environment variable and checks if the change is applied using `echo` in the script.
9. Test environment variable inheritance:
   - Create a script that sets an environment variable and then launches a second script to read that variable.
   - Print the value of the environment variable in the second script to confirm it's inherited.