Topics to be covered:

Week	Broader Topic	Topics	Tools to be covered
7	Shell Scripting	1. Shell scripting basics 2. Types of shells 3. Starting a shell 4. Create your first script - Hello world 5.Create a Linux script - Conditions/If else statements Scripts 6. Create a Linux script - Case statements script 7. For loop script 8. Do-While Loop Script 9. Create a Linux script -Exit status	Linux Terminal

1. Shell scripting basics

Shell scripting is a powerful way to automate tasks and perform complex operations in a Unix or Linux environment. Shell scripts are text files containing a series of commands that the shell interpreter executes sequentially. Here's an overview of shell scripting basics:

1. Shell Types:

- The shell is a command-line interpreter that processes commands and executes programs.

- Common shells include Bash (Bourne Again SHell), sh (Bourne Shell), csh (C Shell), ksh (Korn Shell), and more.

- Bash is the most widely used shell and is the default in many Linux distributions.

2. Script File Creation:

- Create a new file with a `.sh` extension, such as `script.sh`, to indicate it is a shell script.

- Start the script with a shebang (`#!/bin/bash`) to specify the shell to use.

3. Comments:

- Use comments (`#`) to add explanations and documentation within the script.

- Comments are ignored by the shell and do not affect the script's execution.

4. Variables:

- Variables store data for later use in the script.

- Declare a variable by assigning a value to it, e.g., `name="John"`.

- Access the value using `$` followed by the variable name, e.g., `echo $name`.

5. Command Substitution:

- Command substitution enables the assignment of a command's output to a variable.

- Use `$(command)` or backticks (`command`) to substitute the command's output.

6. Input and Output:

- Use `read` to prompt the user for input and assign it to a variable.

- Output can be displayed using `echo`, `printf`, or other commands.

- Redirect output to a file using `>`, append to a file with `>>`, or redirect input using `

2. Types of shells

Shells in Linux are command-line interpreters that provide an interface between the user and the operating system. They interpret user commands and execute them by interacting with the system's kernel. Here are some common types of shells found in Linux:

1. Bash (Bourne Again SHell):

- Bash is the most popular and widely used shell in Linux distributions.

- It is the default shell for many Linux distributions due to its rich features and compatibility with the Bourne shell.

- Bash is backward compatible with the Bourne shell (sh) and incorporates features from other shells like Korn shell (ksh) and C shell (csh).

- It supports command line editing, history, job control, shell scripting, and a wide range of features for customization.

2. Bourne Shell (sh):

- The Bourne shell is one of the original Unix shells and is the ancestor of many other shells.

- The syntax is easy to understand and basic.

- Bourne shell scripts have the `.sh` file extension and are compatible with most Unix-like systems.

3. C Shell (csh):

- The C shell provides a C-like syntax for command-line interaction.

- It introduced features like command history, aliases, and a C-like control flow.

- C shell scripts have the `.csh` or `.tcsh` file extensions.

4. Korn Shell (ksh):

- The Korn shell is an enhanced version of the Bourne shell with additional features from the C shell and the TC shell.

- It offers a more powerful scripting language with improved command line editing and advanced programming constructs.

- Korn shell scripts have the `.ksh` file extension.

5. Zsh (Z Shell):

- Zsh is an extended shell with additional features and customization options.

- It includes features from other shells like Bash, Korn shell, and C shell.

- Zsh offers advanced command line editing, spelling correction, file globbing, themes, plugins, and extensive customization capabilities.

6. Fish (Friendly Interactive SHell):

- Fish is a modern and user-friendly shell with an emphasis on simplicity and ease of use.

- It has an intuitive auto-suggestion feature and rich syntax highlighting.

- Fish offers features like easy-to-read scripting syntax, powerful tab completion, and built-in functions.

These are just a few examples of the different types of shells available in Linux. Each shell has its own set of features, syntax, and capabilities. You can choose a shell based on your preferences, requirements, and compatibility with your Linux distribution.

3. Starting a shell

In Linux, starting a shell refers to launching an interactive shell session, which allows you to enter commands and interact with the operating system. Here's a detailed guide on starting a shell in Linux:

1. Terminal Emulators:

- Linux provides terminal emulators that serve as the interface for launching and interacting with shells.

- Terminal emulators are applications that provide a text-based interface within a graphical environment.

- Common terminal emulators in Linux include GNOME Terminal, Konsole, xterm, and Terminator.

2. Launching a Shell:

- To start a shell, open a terminal emulator from your Linux distribution's application menu or use a keyboard shortcut like Ctrl+Alt+T (may vary depending on the distribution).

- Once the terminal emulator is open, it will display a command prompt where you can enter commands.

3. Default Shell:

- When a terminal emulator starts, it automatically launches a default shell defined in the system configuration.

- The default shell is typically Bash (Bourne Again SHell) in most Linux distributions.

- However, you can configure your system to use a different default shell if desired.

4. Switching Shells:

- If you want to switch to a different shell, you can do so by entering the command for the desired shell at the command prompt.

- For example, to switch to the Zsh shell, you would enter `zsh` and press Enter.

- This will start a new interactive shell session using the specified shell.

5. Exiting a Shell:

- To exit a shell session, you can use the `exit` command or press Ctrl+D.

- This will terminate the shell session and return you to the terminal emulator or close the terminal window, depending on the configuration.

6. Shell Options and Customization:

- Shells provide various options and customization settings to personalize your shell experience.

- You can customize shell prompt appearance, define aliases, set environment variables, and configure other preferences to suit your needs.

- Shell configuration files like `.bashrc` (for Bash) or `.zshrc` (for Zsh) are used to define these customizations.

It's important to note that Linux offers multiple shells with different features and capabilities. The process of starting a shell remains the same regardless of the specific shell you choose. By starting a shell in Linux, you gain access to a powerful command-line interface for executing commands, managing files and directories, and performing various system tasks.

4. Create your first script - Hello world

Certainly! A straightforward "Hello, World!" script made using Linux shell scripting is shown here:

1. Open a text editor and create a new file. You can name it `hello.sh` or any other desired name with the `.sh` file extension.

2. To the script, insert the following lines:

```bash
#! /bin/bash
# This is a basic script to print "Hello, World!"
echo "Hello, World!"
```

3. Save the file and exit the text editor.

4. Make the script executable by running the following command in the terminal:

```bash
chmod +x hello.sh
```

This command grants the script execute permissions.

5. Run the script by executing the following command:

```bash
./hello.sh
```

With this, the script will run and produce the following output: Saying "Hello, World!"

Explanation:

- The `#!/bin/bash` line at the beginning of the script is called a shebang and specifies the interpreter (in this case, `/bin/bash`) to execute the script.

-The 'echo' command is used to print text to the terminal. "Hello, World!" is printed in this instance.

- The `chmod +x` command makes the script executable by granting execute permissions to the script file.

- Finally, running the script with `./hello.sh` executes the script and displays the output.

Your "Hello, World!" script was successfully built and run using Linux shell scripting.

5.Create a Linux script - Conditions/If else statements Scripts

Certainly! Here's an example of a Linux shell script that demonstrates the usage of conditional statements (if-else) to perform different actions based on certain conditions:

```bash
#!/bin/bash

# Example script with if-else statements

# Prompt the user for their age
read -p "Enter your age: " age

# Check the age and display a message accordingly
if [ $age -ge 18 ]; then
  echo "You are an adult."
else
  echo "You are a minor."
fi

# Prompt the user for a number
read -p "Enter a number: " number

# Check if the number is positive, negative, or zero
if [ $number -gt 0 ]; then
  echo "The number is positive."
elif [ $number -lt 0 ]; then
```

```
  echo "The number is negative."
else
  echo "The number is zero."
fi

# Check if a file exists
file_path="/path/to/file.txt"
if [ -f "$file_path" ]; then
  echo "The file $file_path exists."
else
  echo "The file $file_path does not exist."
fi
```

Save the script with a `.sh` file extension (e.g., `conditional_script.sh`), make it executable (`chmod +x conditional_script.sh`), and then run it (`./conditional_script.sh`). The script will prompt the user for their age, a number, and check the existence of a file, displaying the appropriate messages based on the conditions.

6. Create a Linux script - Case statements script

Certainly! Here's an example of a Linux shell script that uses the `case` statement to handle different options:

```bash
#!/bin/bash

# Prompt the user for input
echo "Select an option:"
echo "1. Option 1"
echo "2. Option 2"
echo "3. Option 3"
echo "4. Quit"
```

```
# Read user input
read choice

# Process the input using a case statement
case $choice in
  1)
    echo "You selected Option 1."
    # Add your code for Option 1 here
    ;;
  2)
    echo "You selected Option 2."
    # Add your code for Option 2 here
    ;;
  3)
    echo "You selected Option 3."
    # Add your code for Option 3 here
    ;;
  4)
    echo "Quitting the script."
    exit 0
    ;;
  *)
    echo "Invalid option. Please select a valid option."
    ;;
esac
```

In this script, the user is presented with a menu of options (1, 2, 3, and 4). The script reads the user's input and uses the `case` statement to execute different code blocks based on the selected option.

To use this script, you can follow these steps:

1. Open a text editor and paste the script into a new file.

2. Save the file with a ".sh" extension, for example, "case_script.sh".

3. Open a terminal and navigate to the directory where the script is saved.

4. Make the script executable by running the command: `chmod +x case_script.sh`.

5. Run the script by executing: `./case_script.sh`.

6. Follow the on-screen prompts, entering the desired option.

Based on the selected option, the script will execute the corresponding code block. You can customize each option's code block to perform specific actions or implement desired functionality.

7. For loop script

Certainly! Here's an example of a Linux script that includes case statements and a for loop:

```bash
#!/bin/bash

# Example script with case statements and for loop

# Prompt the user for input
echo "Enter a fruit name (apple, banana, orange): "
read fruit

# Perform actions based on user input using a case statement
case $fruit in
  apple)
    echo "You selected apple."
    # Add any specific actions for apple here
    ;;
  banana)
    echo "You selected banana."
    # Add any specific actions for banana here
    ;;
```

```
  orange)
    echo "You selected orange."
    # Add any specific actions for orange here
    ;;
  *)
    echo "Invalid selection. Please choose apple, banana, or orange."
    exit 1
    ;;
esac

# Example for loop to iterate over a list of numbers
echo "Printing numbers 1 to 5:"
for i in 1 2 3 4 5; do
  echo $i
done

# End of the script
```

To use this script:

1. Open a text editor and paste the above code into a new file.

2. Save the file with a ".sh" extension, for example, "myscript.sh".

3. Make the script executable by running the command: `chmod +x myscript.sh`.

4. Execute the script by running the command: `./myscript.sh`.

The script starts by prompting the user to enter a fruit name. It then uses a case statement to perform different actions based on the user's input. In this example, it displays a message for each fruit: apple, banana, or orange. For an invalid selection, it provides an error message and exits the script.

Following the case statement, there is a for loop that iterates over a list of numbers (1 to 5) and prints them on the terminal.

8. Do-While Loop Script:

```
#!/bin/bash

# Initialize a counter
counter=1

# Perform an action at least once using a do-while loop
while :
do
  echo "Action performed: $counter"

  # Increment the counter
  ((counter++))

  # Ask the user if the action should be repeated
  echo "Do you want to perform the action again? (y/n)"
  read choice

  # Exit the loop if the user chooses not to repeat the action
  if [ "$choice" == "n" ]; then
    break
  fi
done
```

This script demonstrates a do-while loop in Linux. It initializes a counter variable and enters an infinite loop. It performs an action (printing the counter value) and increments the counter on each iteration. Then, it asks the user if they want to repeat the action. If the user enters "n" for "no," the loop is exited. Otherwise, the loop continues, and the action is repeated.

You can save each script in separate files (e.g., case_script.sh and do_while_script.sh), make them executable using the chmod +x command, and run them in a Linux environment.

9. Create a Linux script -Exit status

Certainly! Here's an example of a simple Linux script that demonstrates the concept of exit status:

```bash
#!/bin/bash

# This script checks if a file exists and prints an appropriate message based on its existence.

filename="example.txt"

if [ -f "$filename"]; then
  echo "File '$filename' exists."
  exit 0 # Exit with a success status (0) if the file exists
else
  echo "File '$filename' does not exist."
  exit 1 # Exit with a failure status (non-zero) if the file does not exist
fi
```

In the above script:

- The script starts with the shebang `#!/bin/bash`, which indicates that it should be executed using the Bash shell.

- The `filename` variable is set to the name of the file we want to check for existence.

- The script uses an `if` statement to check if the file exists using the `-f` condition, which tests if the given file is a regular file.

- If the file exists, it prints a message indicating its existence and exits with a status of 0 (success).

- If the file does not exist, it prints a message indicating its non-existence and exits with a status of 1 (failure).

You can save the script in a file, such as `check_file.sh`, and make it executable using the command `chmod +x check_file.sh`. Then, you can run the script using `./check_file.sh`, and it will output the appropriate message based on the existence of the file and exit with the corresponding status.