



# Advanced Bash

Cybersecurity

Bash Scripting and Programming Day 1



# Today's Objectives

---

By the end of today's class, you will be able to:



Construct compound commands using `&&`, `|` and file redirects.



Create alias commands and save them to their `~/.bashrc` file.



Edit your `$PATH` variable to include a custom `~/scripts` directory.



Create simple bash scripts comprised of a list of commands.

# Creating Compound Commands



# Why compound commands?

# Why Compound Commands?

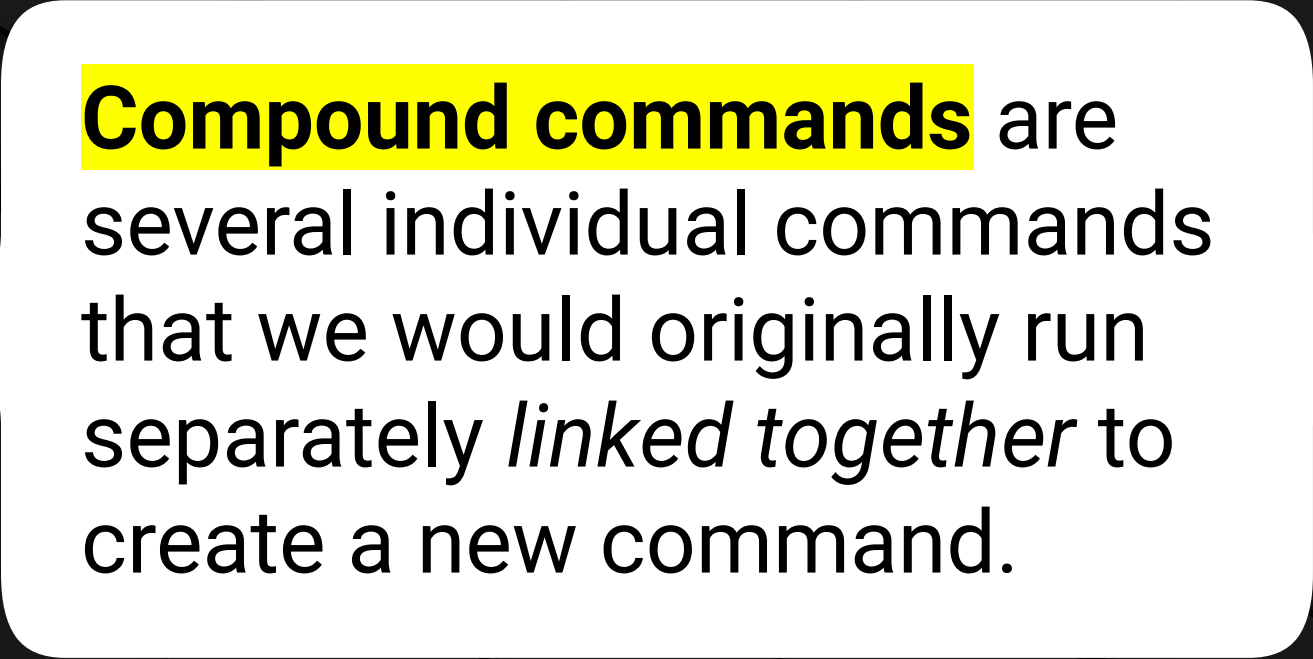
---

Navigating Linux directories, quickly searching large log files, and writing small scripts to automate tasks will save you time and energy.





**What are compound commands?**



**Compound commands** are several individual commands that we would originally run separately *linked together* to create a new command.

# Syntax Breakdown

```
file $(find / -iname '*.txt' 2>/dev/null) > ~/Desktop/text_files ; tail ~/Desktop/text_files
```

This command does the following:



Searches the entire computer for files ending in `.txt`;



Verifies that the files found are text files;



Ignores any errors it comes across;



Creates a list of all found files before saving the list to the desktop;



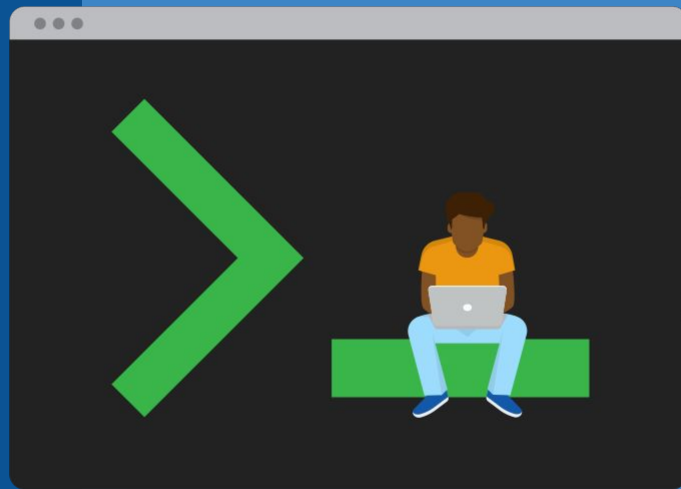
Finally, it will open the file and print the last ten lines that were added.



# Let's Review

---

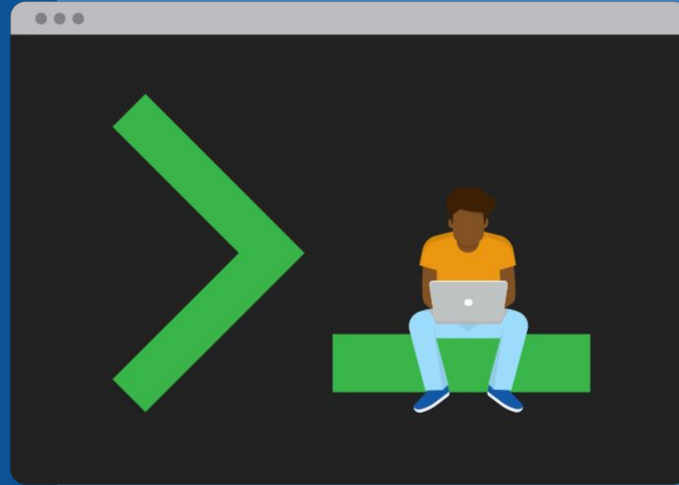
We've already chained commands using the following `>`, `>>`, and `|`.



# Chaining with > and >>

```
ls > list.txt
```

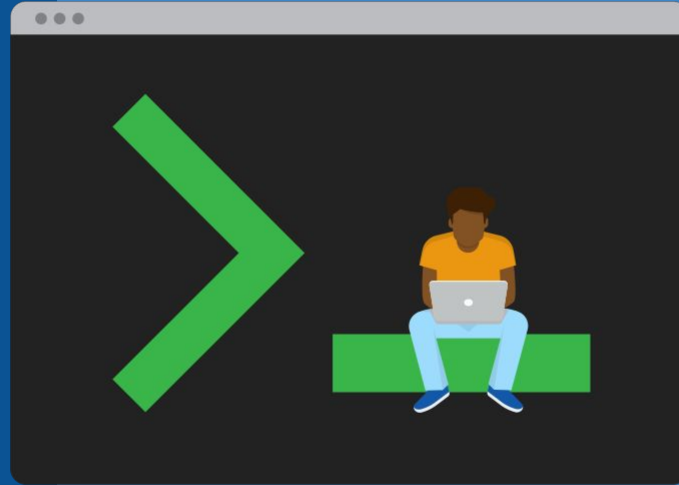
- This command takes the output of the ls command and sends it into a new file named `list.txt`.
- If the file list.txt already exists, it is overwritten with the output of the ls command.



# Chaining with > and >>

```
> list.txt
```

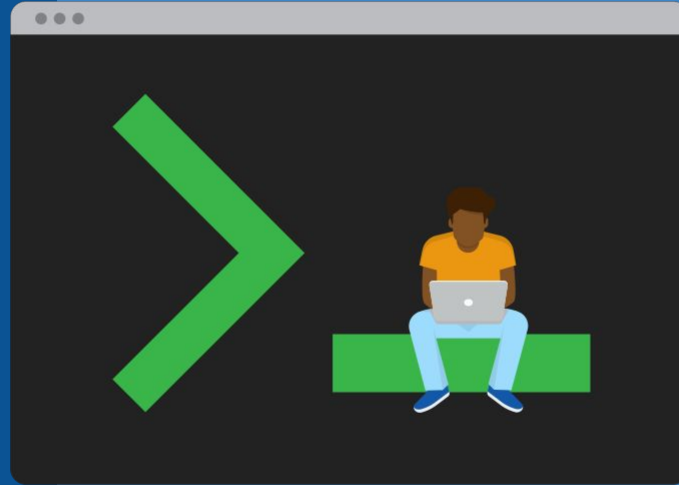
- Without a command in front of >, there is no output to send to the list.txt file.
- However, the file is still written, without output, creating a blank file. If the file `list.txt` exists, it is overwritten with nothing.



# Chaining with > and >>

```
ls >> list.txt
```

- >> will append the output of the ls command to the list.txt file.
- If the list.txt file does not exist, it is created.
- Therefore, using >> instead of > is always safer, unless you want the file to be overwritten.



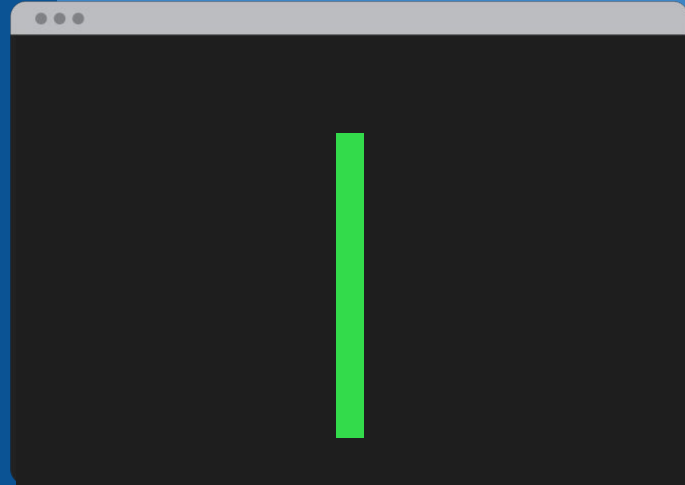
# Piping with |

---

The pipe ( | ) takes the output of one command and sends it to the input of another command.

Compound commands with pipes typically follow this format:

```
program -options arguments |  
program -options | program  
-options | program -options
```



# Review: Piping with |

---

For example:

```
ls -l | grep '.txt'
```



ls -l creates a list of files.



| pipes the list from ls into the command that follows.



grep searches the files from ls for the string that follows.



.txt matches any file that contains .txt in the filename.

# Review: Piping with |

---

Some common programs that users will pipe to include:



| head prints only the first 10 lines of output.



| tail prints only the last 10 lines of output.



| sort sorts the output alphabetically



| sed searches and replaces parts of the output.



| awk display only specified parts of the output.

# Review: Piping with |

---

```
cat /etc/passwd | grep sysadmin | awk -F ':' '{print $6}'
```



cat /etc/passwd dumps the contents of /etc/passwd to output.



| pipes that output into the command that follows.



grep sysadmin` displays lines that contain `sysadmin`.



| pipes that output into the command that follows.



awk -F ':' '{print \$6}' only prints the sixth field of the line.

- awk usually looks for a space to use as a field separator, but in this case we want it to separate the line by a colon, because `/etc/passwd` uses colons to separate its fields.

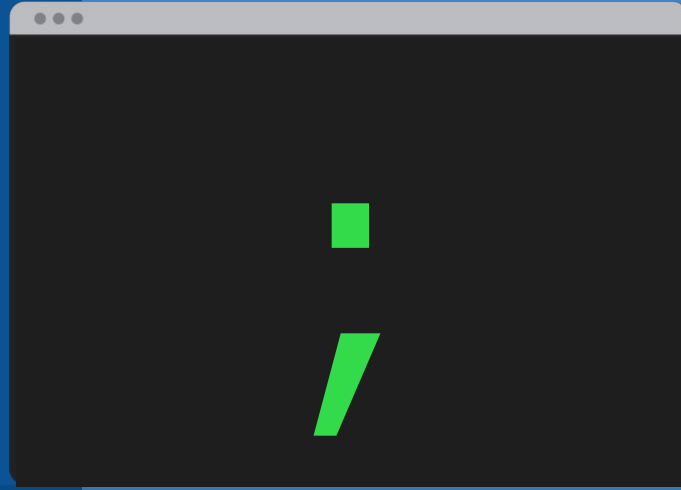


## Other Methods

---

We can also use a semicolon ( ; ) to run a series of commands back to back.

When using ;, each command is running on its own. It is not sending its output to the next command. Therefore, each command can have its own arguments.



## Combining with ;

Rather than running this:

```
$ mkdir dir
$ cd dir
$ touch file
$ ls -l
-rw-r--r-- 1 user user 0
Sep  4 15:33 file
```

We can use one command:

```
$ mkdir dir; cd dir; touch
file; ls -l

-rw-r--r-- 1 user user 0 Sep
4 15:33 file
```

# Combining with ;

---

Note the misspelling of “dir”:

```
mkdir dir; cd dor; touch file; ls
```

-1

This command would fail because we are trying to move into the directory dor, which has not been created.

However, the commands touch and ls will still run.

## Combining with &&

A better operator to use in the previous case is the &&.

The && will run the next command only if the first command were successful.



&&

## Combining with &&

---

```
mkdir dir && cd dir && touch file && ls -l
```

cd would only run if mkdir were successful. touch would only run if cd were successful. ls would only run if `touch` were successful.

```
mkdir dir && cd dor && touch file && ls -l
```

Only mkdir dir and cd dor would run. cd dor fails, so touch and ls are ignored.



# Instructor Demonstration

---

## Chaining Commands Review

# Combining Commands:

In the previous demo, we covered how to chain commands using the following:

|    |   |  |
|----|---|--|
| >  | <code>ls &gt; list.txt</code>   | Takes the output of the <code>ls</code> command and sends it into a new <code>.txt</code> file if it does not already exist.                         |
| >> | <code>ls &gt;&gt; list.txt</code>   | Takes the output of the <code>ls</code> command and sends it into a <code>.txt</code> file. If <code>.txt</code> does not exist, it will be created. |
|    | <code>ls -l   grep '*.txt'</code>   | "Pipes", or sends the output of one command and sends it as the input into the following commands.   |
| ;  | <code>mkdir dir; cd dir; touch file; ls -l</code>                               | Each command will run regardless of the outcome of the preceding command.  |
| && | <code>mkdir dir &amp;&amp; cd dir &amp;&amp; touch file &amp;&amp; ls -l</code> | The next command is only run if the previous command was successful.   |



## Activity: Compound Commands

In this activity, you will audit a new system.

In order to simplify the process, we will combine several commands together.

Suggested Time:

15 Minutes





Time's Up! Let's Review.

# Questions?

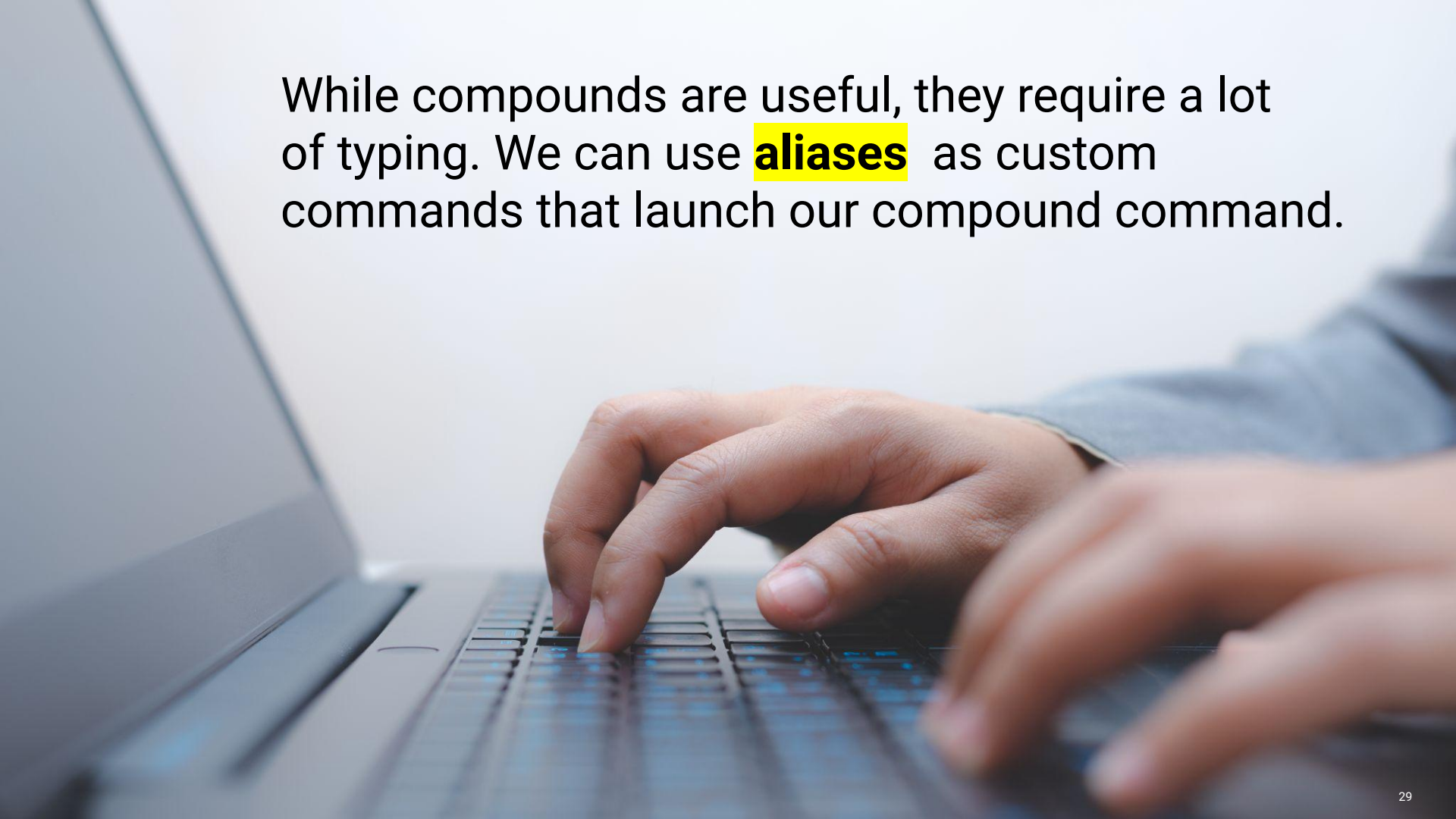


# Changing Aliases



**What are aliases?**

While compounds are useful, they require a lot of typing. We can use **aliases** as custom commands that launch our compound command.



# Syntax Breakdown

```
alias lh="ls -lah"
```



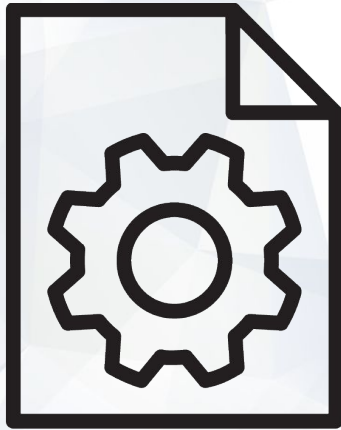
`alias` indicates we are creating an alias.



`lh` is our custom command.



`ls -lah` is the command that runs when we use our alias `lh`.



**In the next demo, we will create custom commands using aliases and save the configuration file so we can use them again whenever we log in.**



# Instructor Demonstration

---

## Creating Aliases





## Activity: Creating Aliases

In this activity, you will create several aliases and save them to your `~/.bashrc` file.

Suggested Time:

15 Minutes



Time's Up! Let's Review.

# Questions?



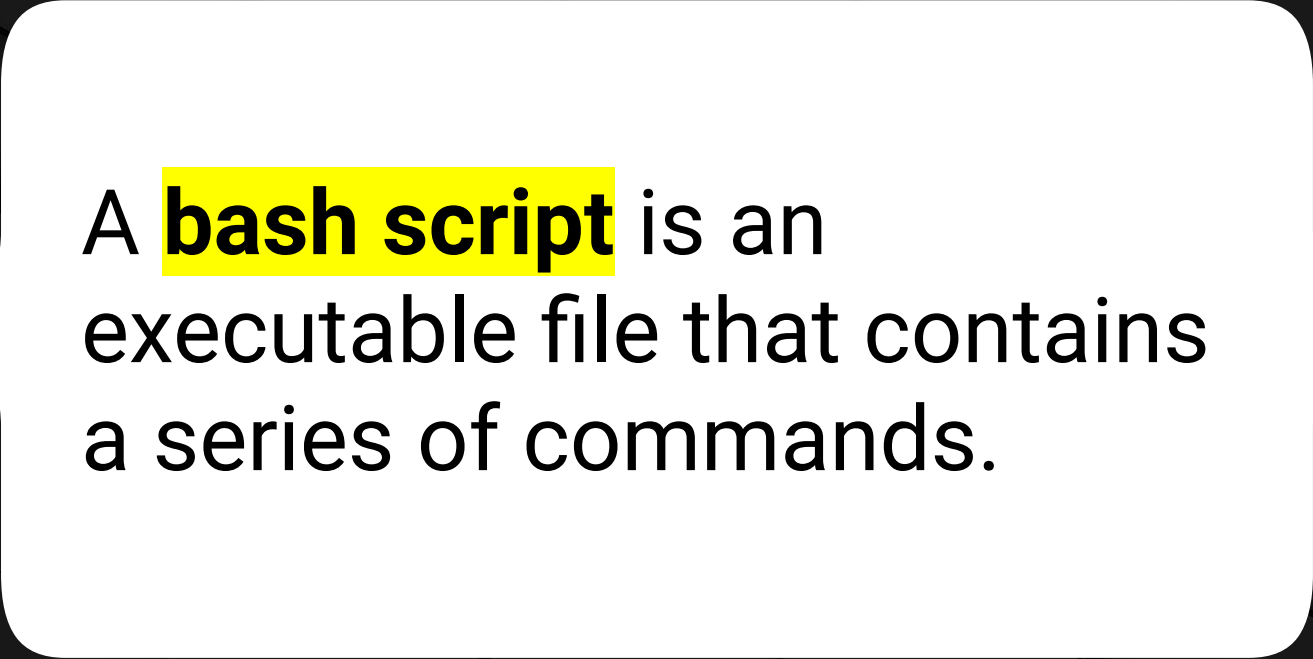


Countdown timer

15:00

(with alarm)

# My First Bash Script



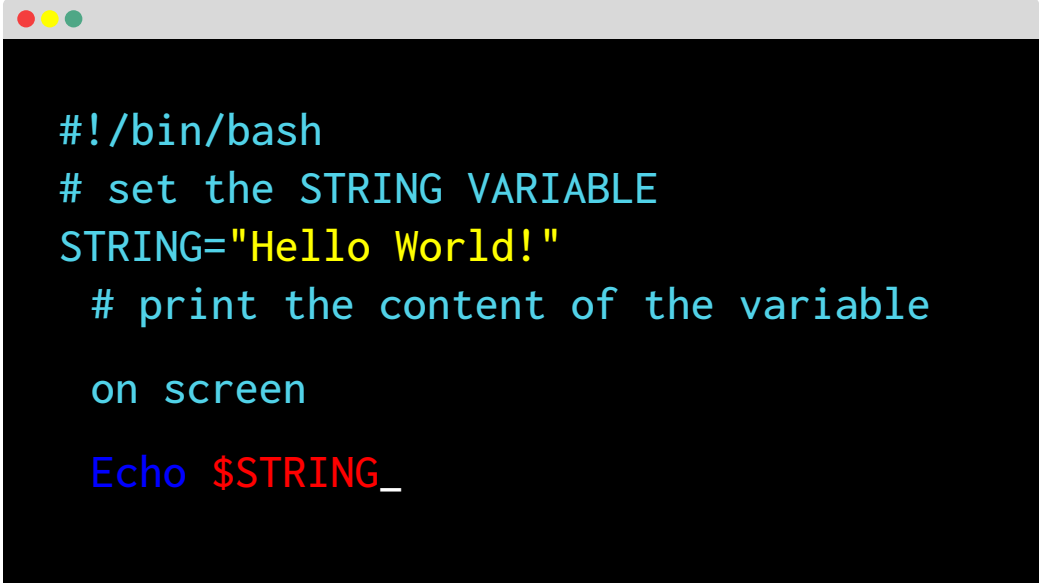
A **bash script** is an executable file that contains a series of commands.

# Variables

---

When the bash script is executed, these commands will run one by one until they are all executed.

A fundamental system administrator skill is creating a bash script and then scheduling it to run at a regular time using cron.



```
#!/bin/bash
# set the STRING VARIABLE
STRING="Hello World!"
# print the content of the variable
on screen
Echo $STRING_
```

# Variable Demo

---

In the following demo, we will use:



Basic Variables



Built-In Variables



Common Expansion



Variables in Scripts





# Instructor Demonstration

---

## My First Bash Script



## Activity: My First Bash Script

In this activity, you will work in groups of two to create a script that completes system audit steps automatically.

Suggested Time:

20 Minutes



Time's Up! Let's Review.

# Questions?



# Custom Commands

# Custom Commands

Now, we will create a custom command that runs our script.

- In order to do this, we'll have to look under the hood of what happens when we run commands.
- We'll also look a built-in variable known as the PATH variable.





# Instructor Demonstration

---

**PATH**



## Activity: Custom Commands

In this activity, you will continue to add more commands into your script. Then, you will save the script to a directory which will be added to your \$PATH.

---

Suggested Time:

25 Minutes





Time's Up! Let's Review.

# Questions?



*The  
End*