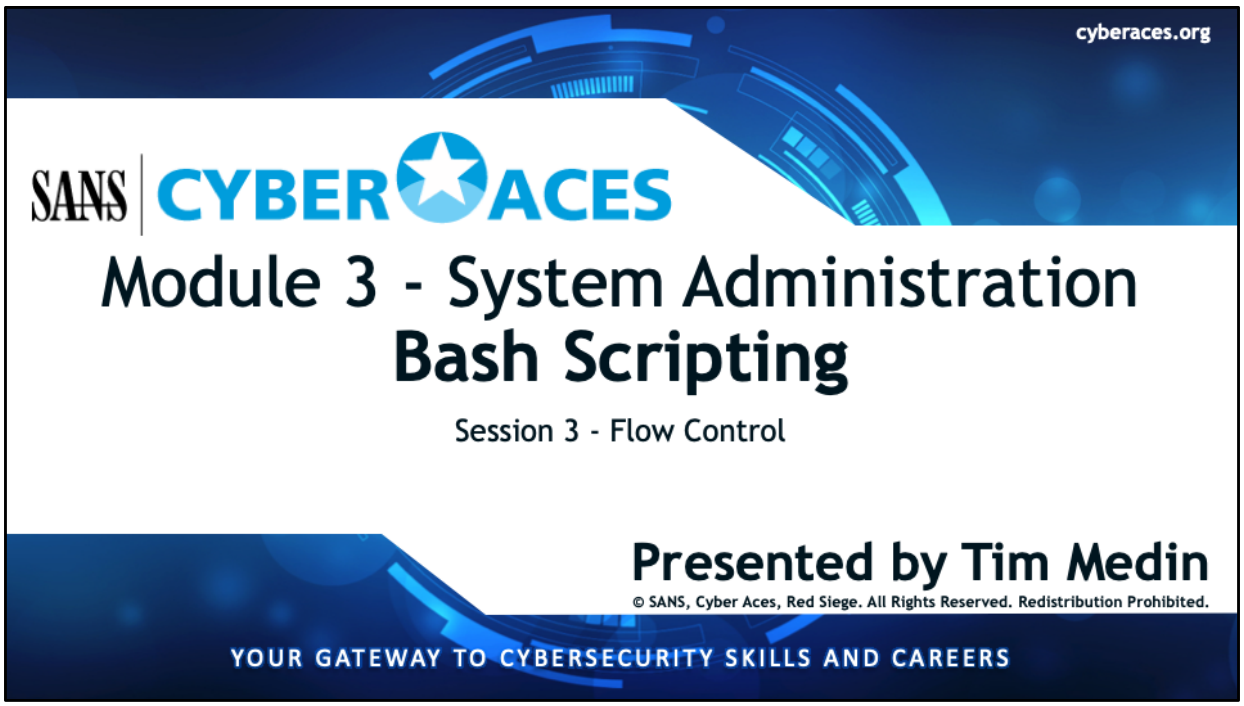
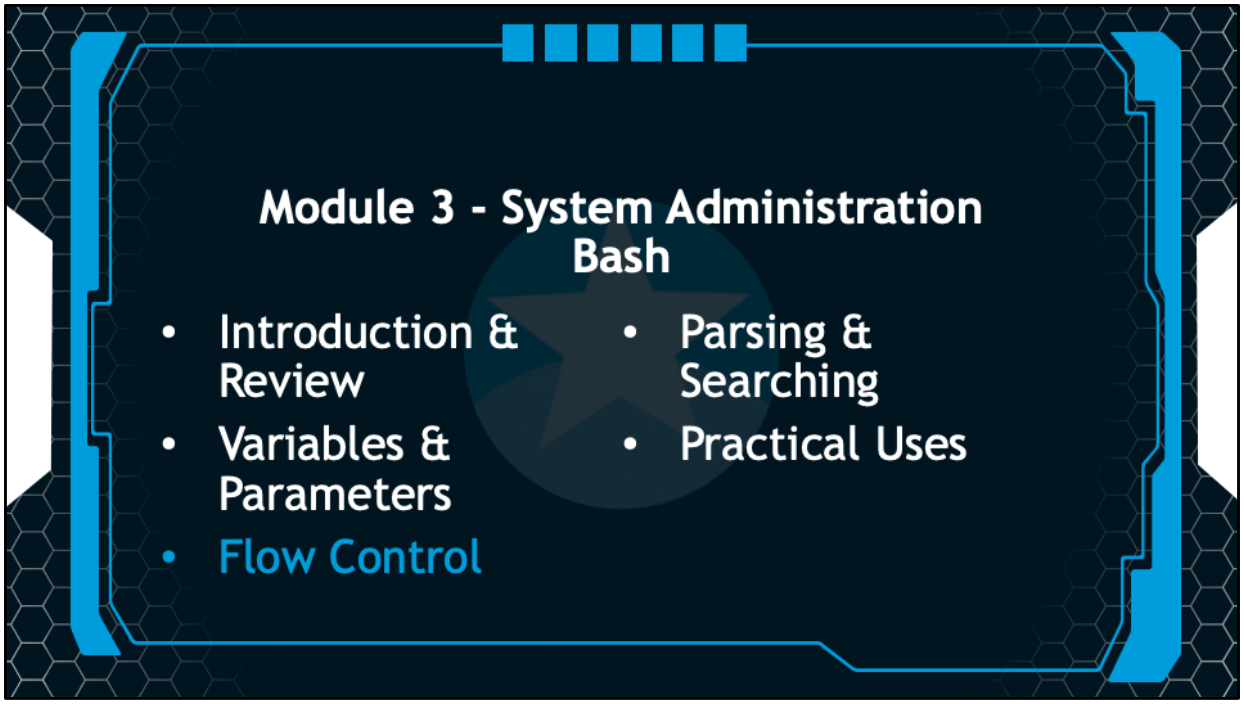


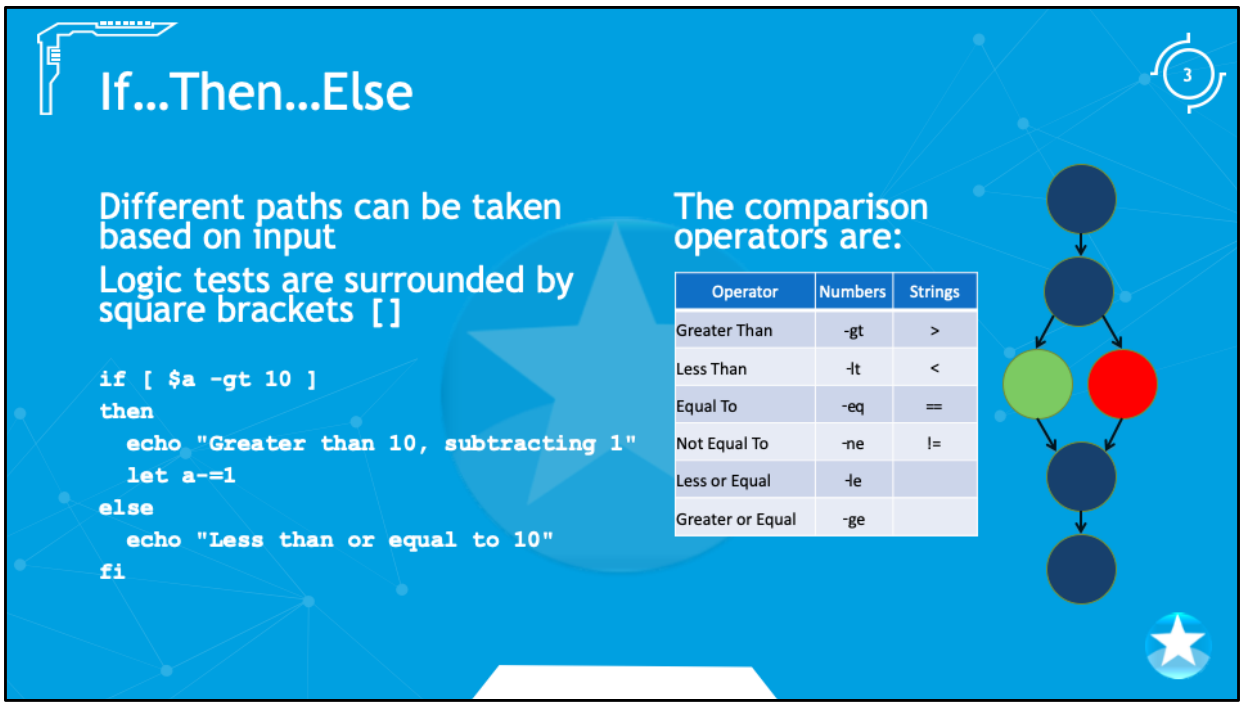
Welcome to Cyber Aces, Module 3! This module provides an introduction to the Bash Scripting. In this session we will be discussing Flow Control.



This training material was originally developed to help students, teachers, and mentors prepare for the Cyber Aces Online Competition. This module focuses on the basics of system administration and scripting. . This session is part of Module 3, System Administration. This module is split into three sections, Bash, PowerShell, and Python. In this session, we will continue our examination of Bash.

The three modules of Cyber Aces Online are Operating Systems, Networking, and System Administration.

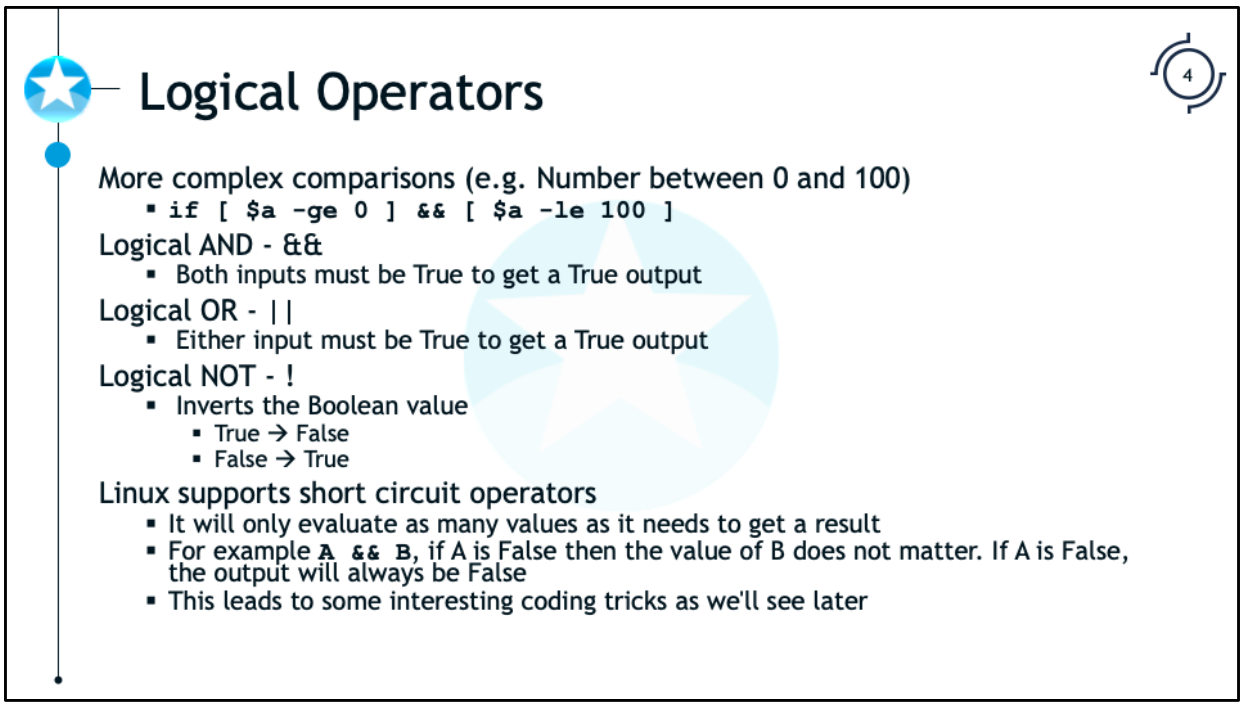
For more information about the Cyber Aces program, please visit the Cyber Aces website at https://CyberAces.org/.



These operators are usually used in a branching mechanism, so that code is executed only if certain conditions are true.

Not only can you compare numbers, but strings (text) can be compared as well.

Description	Number Operator	String Operator
Greater Than	-gt	>
Less Than	-lt	<
Equal To	-eq	==
Not Equal To	-ne	!=
Less or Equal	-le	
Greater or Equal	-ge	



The map of output for a logical AND given inputs of A and B is:

A	B	Output
True	True	True
True	False	False
False	True	False
False	False	False

The map of output for a logical OR given inputs of A and B is:

A	B	Output
True	True	True
True	False	True
False	True	True
False	False	False

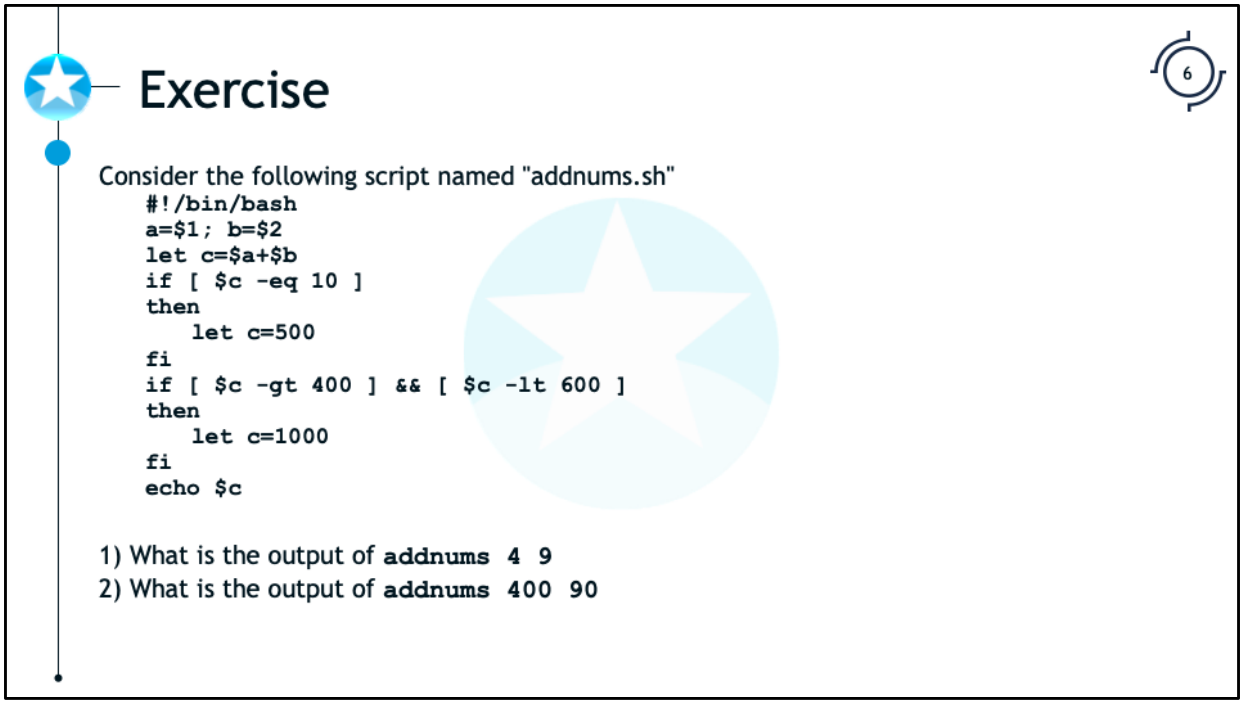
The conditional operators "&&" and "||" allow you to evaluate multiple logic tests surrounded by square brackets. "&&" is a logical "AND" while "||" is a logical "OR". For example, the following section of pseudo-code will only evaluate if both tests are true:

```
if [ $today == "Sunday" ] && [ $month == "April" ]
then
     echo "It is a Sunday in April!!"
fi
```

This section of pseudo-code will execute if either test evaluates to true:

```
if [ $today == "Sunday" ] || [ $month == "April" ]
then
     echo "It is a a Sunday in any month OR it is
     ANY day in the Month of April!!"
```

```
fi
```

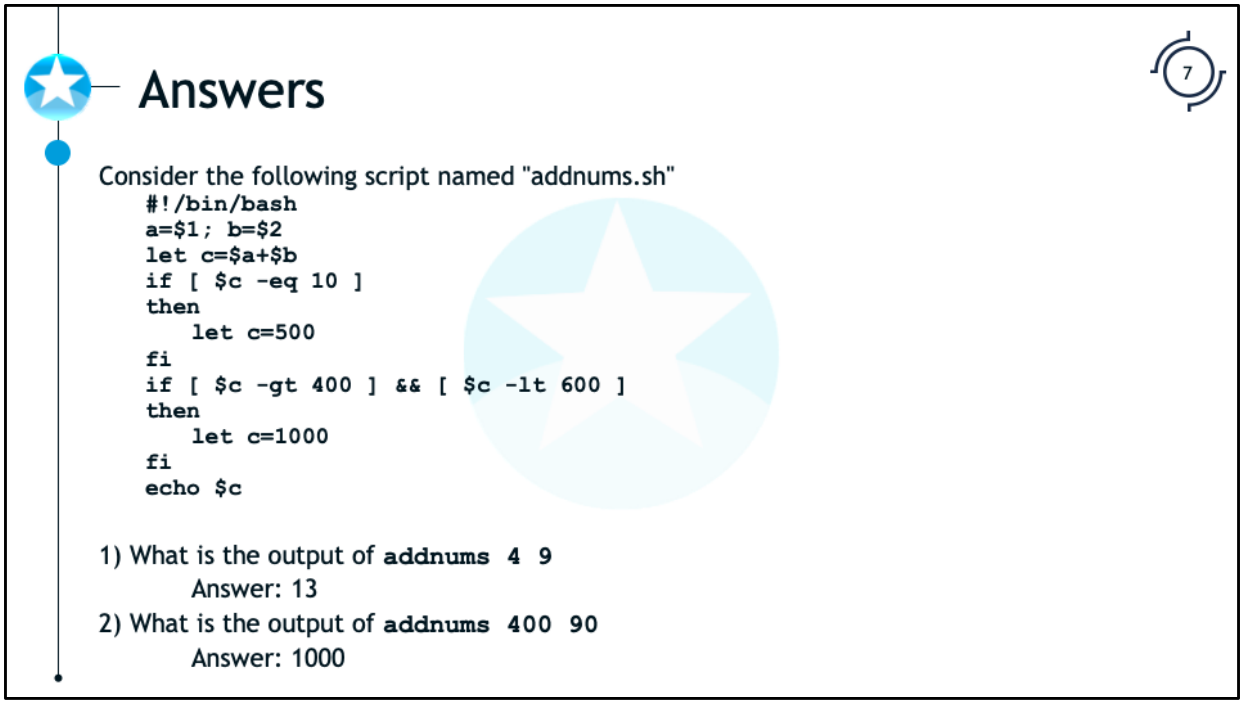


Consider the following script named "addnums.sh"

```
#!/bin/bash
a=$1; b=$2
let c=$a+$b
if [ $c -eq 10 ]
then
  let c=500
fi
if [ $c -gt 400 ] && [ $c -lt 600 ]
then
  let c=1000
fi
echo $c
```

1) What is the output of **`addnums 4 9`**

2) What is the output of **`addnums 400 90`**



Consider the following script named "addnums.sh"

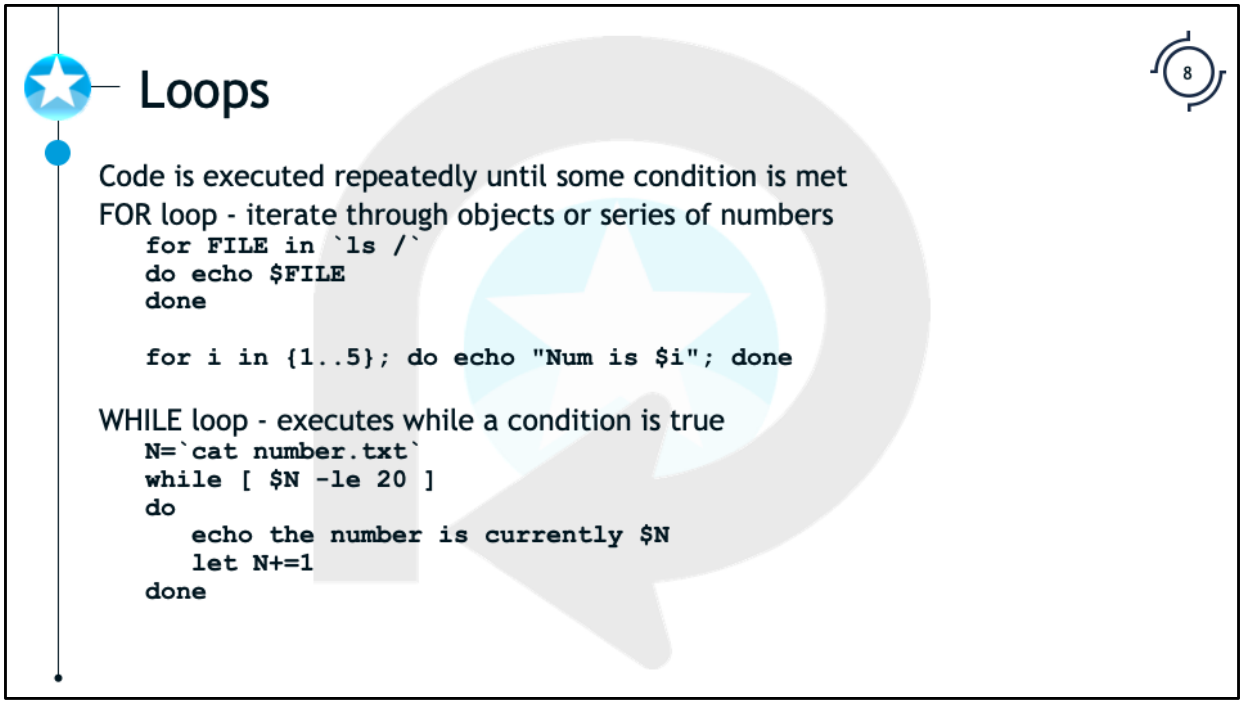
```
#!/bin/bash
a=$1; b=$2
let c=$a+$b
if [ $c -eq 10 ]
then
  let c=500
fi
if [ $c -gt 400 ] && [ $c -lt 600 ]
then
  let c=1000
fi
echo $c
```

1) What is the output of **`addnums 4 9`**

Answer: 13

2) What is the output of **`addnums 400 90`**

Answer: 1000

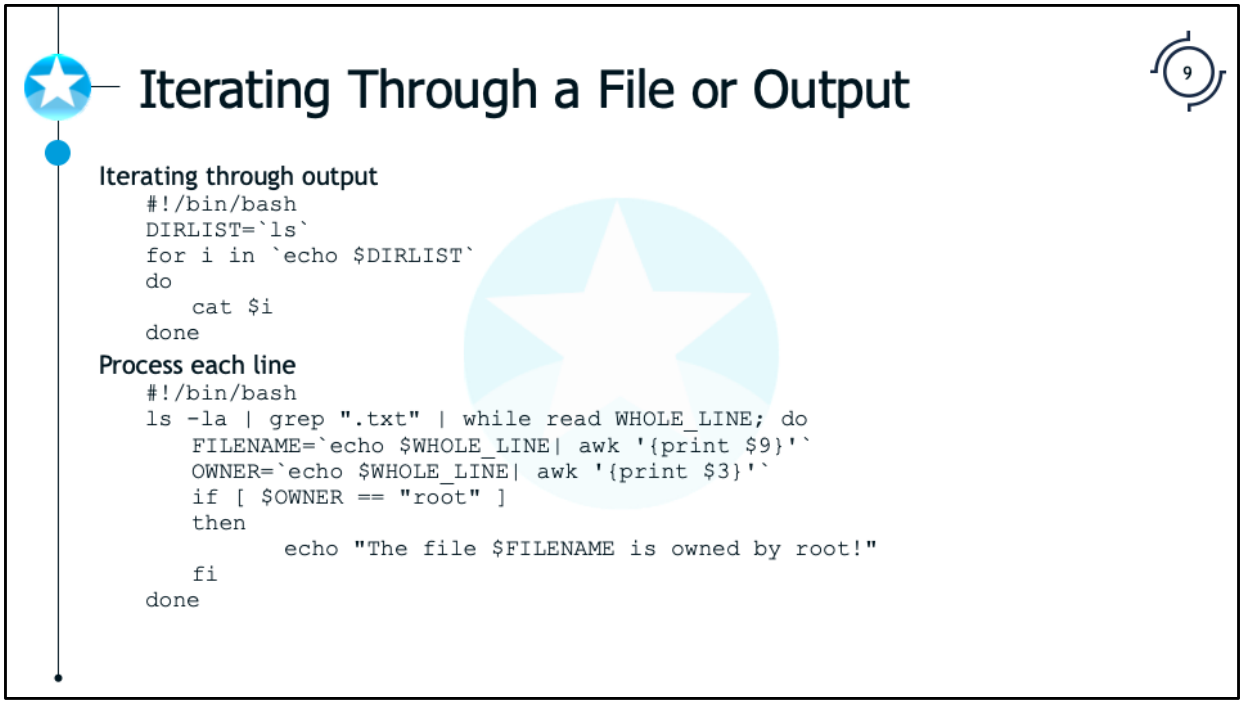


The example For loop will read each word of command output or a series/sequence of numbers. In the example above, the file names are consumed by the For loop. The variable $FILE will contain each filename, one at a time. This loop simply prints the file names.

The curly braces ({}) are used to create a sequence. The second loop will count from 1 to 5 by 1. The sequence operator will also take a "step" option, so to count from 0 to 10 by 2 the proper input would be: {0..10..2}.

The two For loops are represented in two different ways. The first has each command on a separate line. The second options allows all the commands to be entered on the same line where each command is separated by a semicolon (;).

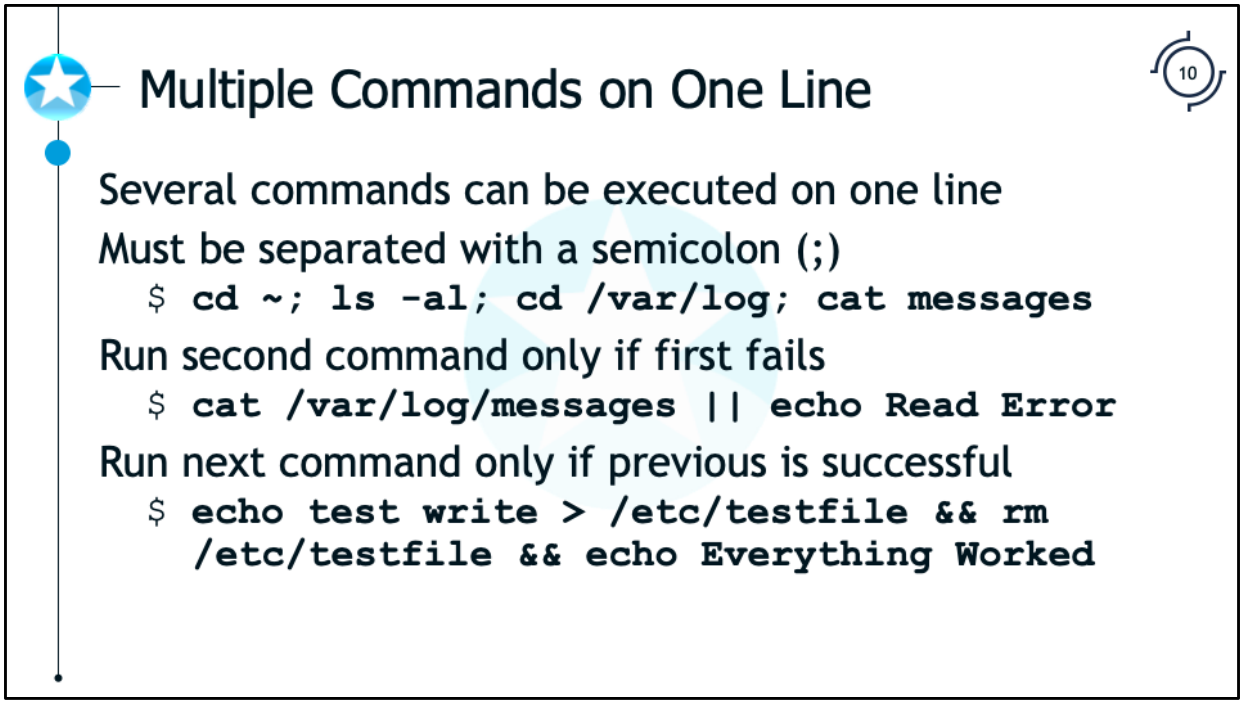
The example While loop will read a number from a file and increment it until it equals 20. In each iteration of the loop it prints the number on the screen. If the initial number read from the file is greater than 20 then the loop will never be entered and there will be no output.



There are several ways to capture and process the output of a command or the contents of a file. We will talk about two methods here. The first method allows you to process one word at a time separated by spaces. The second method will process an entire line at a time marked by a line feed. Before we can process the output, we need to capture the output. We already talked about one of these methods when we discussed the use of variables. We can capture the output of one command in a variable with inline process execution. Let's suppose that we want to capture a listing of all the files in the current directory and then process that listing repeatedly. We can capture a directory listing to a variable then process those results several times. Consider the following bash script that will "cat" every file in the directory, printing its contents to the screen:

```
#!/bin/bash
# Above is the 1st line of all bash scripts and specifies the interpreter
# Executes ls using the backtick. Results stored in DIRLIST variable
DIRLIST=`ls`
# The echo command prints the value of DIRLIST for the For loop
# Each time through the loop $i will contain one of the lines from the
# directory listing
for i in `echo $DIRLIST`
# 'do' marks the beginning of the block of code to execute in our FOR loop
do
     #Block contains one line that prints the contents of each file
     cat i
#done marks the end of our FOR loop
done
```

But this processes each word separated by spaces in the output. If you want to parse an entire line, you can use the while loop with the "read" command, which will assign a full line to a variable as demonstrated in the second script above.



You can run several commands on the same line by separating them with a semicolon (";"). For example:

```
$ cd ~; ls -la ; cd /var/log; cat messages
```

This would change to your home directory, list the files, change to the "/var/log" directory and print the contents of the "messages" file to the screen.

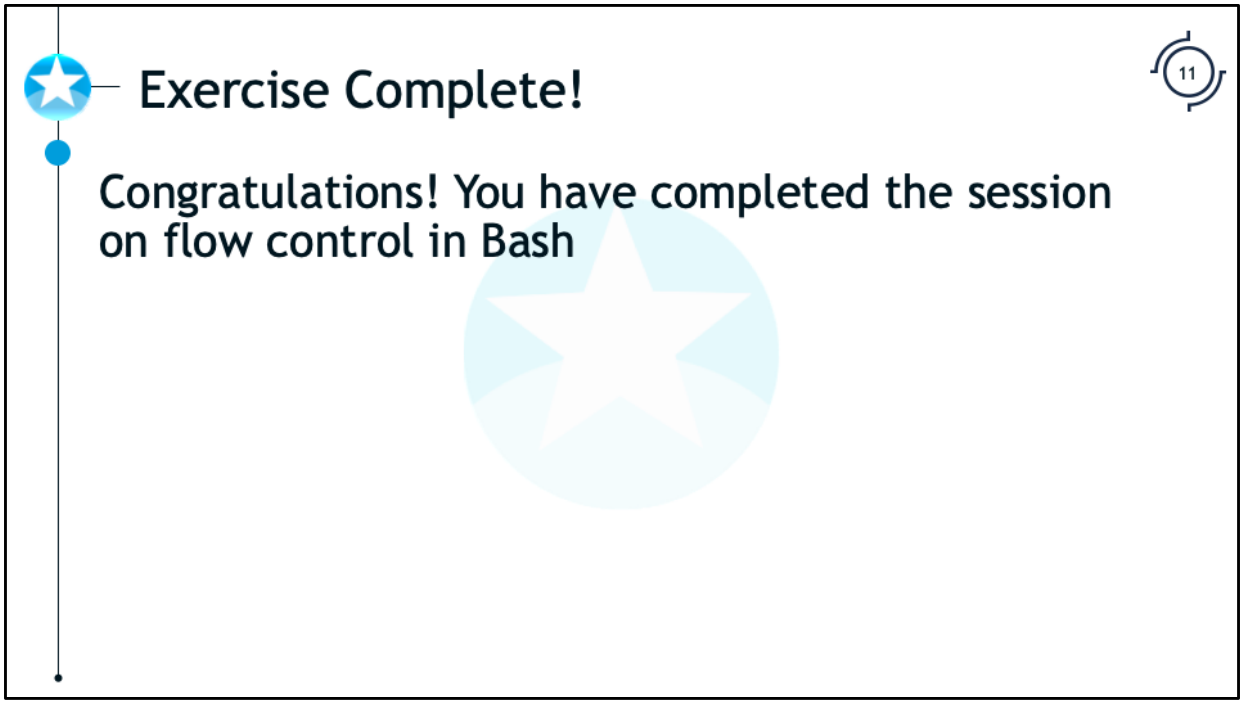
As mentioned earlier, conditional operators can also be used between commands on the command line. With "&&", the second command will only be run if the first command succeeds, and with "||" the second command will only be run if the first command fails. For example, the following line will print an error if there is a problem reading the "messages" log:

```
$ cat /var/log/messages || echo "Error reading messages"
```

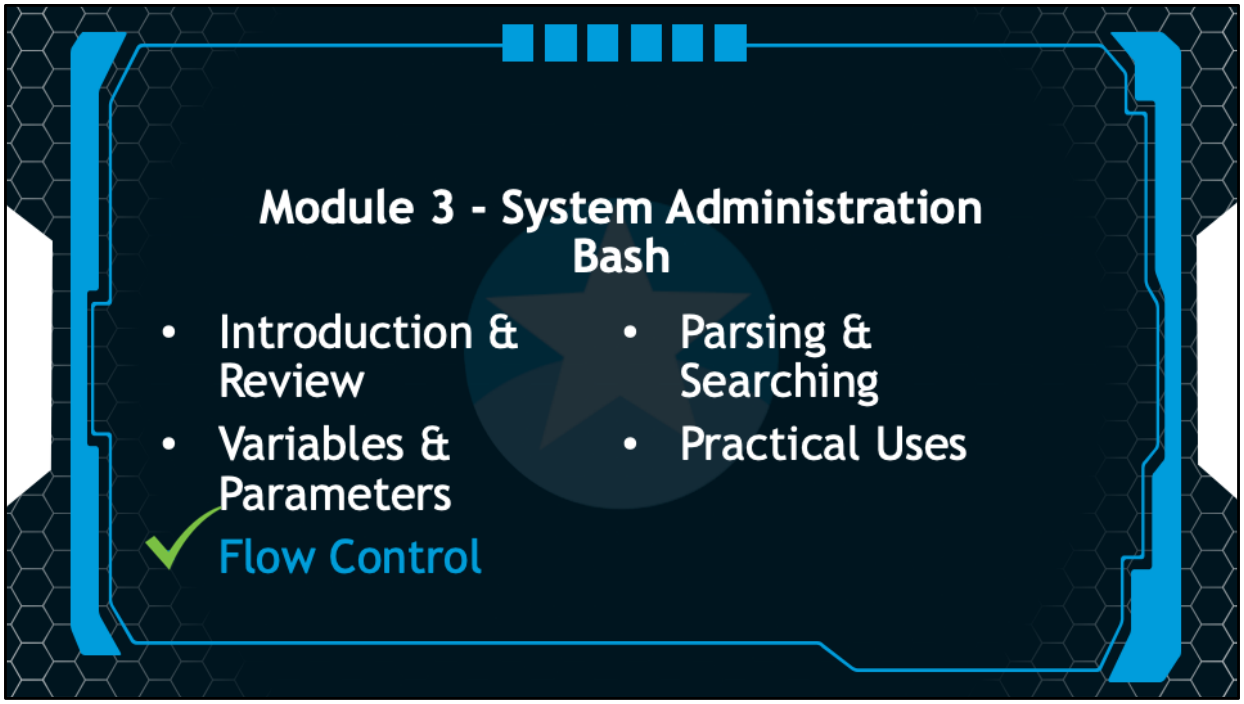
This functionality is due to the "short-circuit" nature of Linux logic. For example, with the Logical AND the result will be False if any of the input is False, so it will stop evaluating input once it reaches the first False result. Similarly, a Logical OR will return True if any input evaluates to True. Once it encounters the first True input it can stop evaluating the inputs.

```
$ echo test write > /etc/testfile && rm /etc/testfile && echo Everything Worked
```

This command has three parts and each piece will execute if the previous was successful. If the creation of /etc/testfile works then, and only then, will the file be deleted. Only if the deletion was successful (the first command would have to have been successful as well), would the words "Everything Worked" be output.



Congratulations! You have completed the session on flow control in Bash.



In the next session, we will discuss parsing and searching using Bash.