

GOALS — BASIC

Dare to be first.

- What is shell scripting?
- What makes shell scripting useful?
- What should you know already...

### GOALS - PART I

Dare to be first.

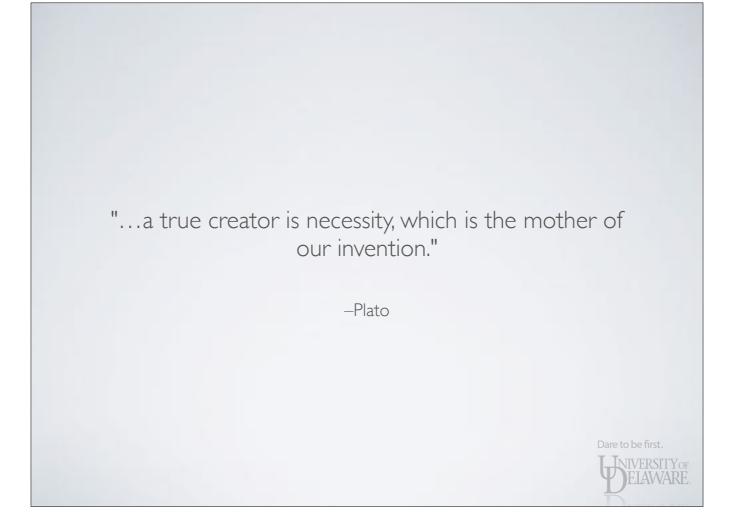
- Script layout/design
- Variables
  - environment vs. local
  - types: string, numeric
- Standard language constructs
  - conditionals
  - loops

### GOALS — PART 2

Dare to be first.

- Command substitution
  - backticks vs. dollar-parenthesis
- Advanced variables
  - advanced expansion
  - arrays
- Advanced language constructs
  - subroutines
  - case statements (branch tables)
  - and/or lists





- You've performed 1000 runs of a program and captured its output in files named out\_#.log
  - Each component of the program prints how long it took to execute (*walltime*): [program] walltime=#
  - The multiple walltimes in a file must be summed
  - You wish to calculate the minimum, maximum, and average walltime

- You've performed 1000 runs of a program and captured its output in files named out\_#.log
  - I.For each file, find all [program] walltime=# lines and extract the number of seconds. Add the number of seconds = total
    - I.If that total is the largest yet, remember it
    - 2.If that total is the smallest yet, remember it
    - 3.Add to the 'total total' walltime
  - 2.Show the minimum, maximum, and average walltime

• For each file, find all [program] walltime=# lines and extract the number of seconds, sum them

\$ total\_wall=\$(grep walltime= file\_1.log | sed 's/^.\*walltime=//' | \
> awk 'BEGIN{total=0;}{total+=\$1;}END{print total;}')

- grep extracts lines with 'walltime='
- sed removes program name and 'walltime='

INIVERSITY

• awk sums the walltimes

Dare to be first. **INIVERSITY**OF ELAWARE

#### • Numerical bits:

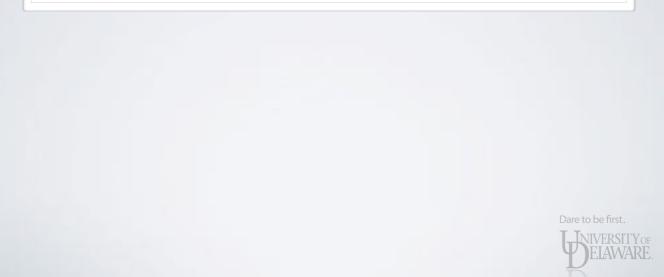
• Initialization:

\$ max\_wall=0; min\_wall=1000000; total\_total\_wall=0; num\_files=0;

- Per-file checks:
- \$ if [ \$total\_wall -gt \$max\_wall ]; then max\_wall=\$total\_wall; fi
  \$ if [ \$total\_wall -lt \$min\_wall ]; then min\_wall=\$total\_wall; fi
  \$ total\_total\_wall=\$((total\_total\_wall+total\_wall))
  \$ num\_files=\$((num\_files+1))

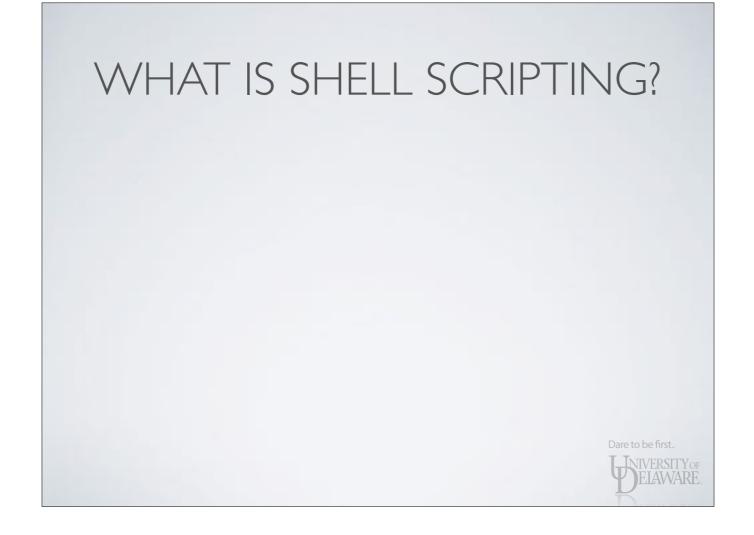
#### • Final analysis:

\$ printf "Total walltime (min/max/avg): "
\$ printf "%d/%d/\n" \$min\_wall \$max\_wall \$((total\_total\_wall/num\_files))



# WHAT IS SHELL SCRIPTING?

- All of the example shell code was entered at the prompt in *interactive* mode
- If you want to reissue a command, you must edit and retype it (or copy-and-paste, etc.)
  - A number of the lines of shell code in our example must be entered 1000 times!



#### WHAT IS SHELL SCRIPTING?

When promoting the Apple II at computer shows, Steve Wozniak would sit down before the show and manually enter the machine code for the BASIC interpreter — because they hadn't made a disk or tape drive for the computer yet.

Imagine his time savings when a tape drive became available!

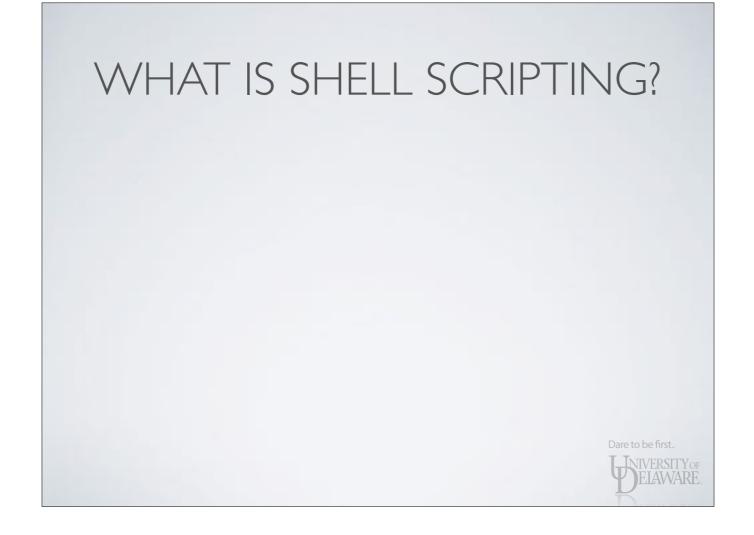
### WHAT IS SHELL SCRIPTING?

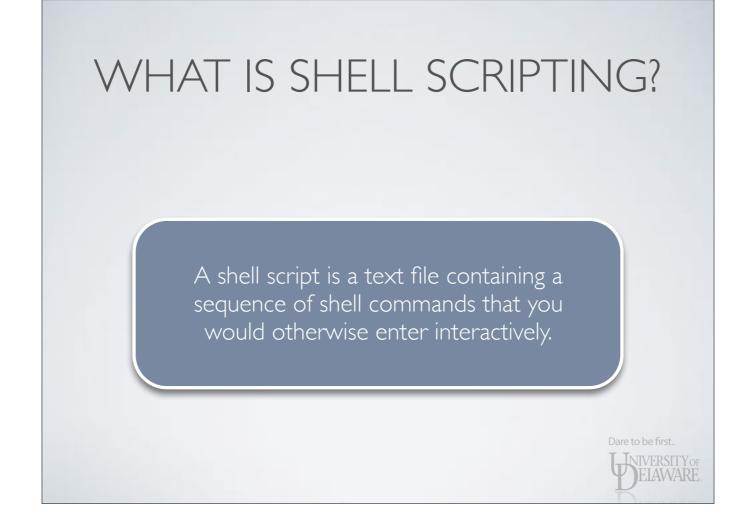
When promoting the Apple II at computer shows, Steve Wozniak would sit down before the show and manually enter the machine code for the BASIC interpreter — because they hadn't made a disk or tape drive for the computer yet.

Imagine his time savings when a tape drive became available!

• Hence the quote from Plato: monotonous, repetitious tasks beg for invention of shortcuts.







# WHAT MAKES SHELL SCRIPTING USEFUL?

- Several obvious reasons:
  - Massive time-savings versus working interactively
  - Easily create your own 'commands'
  - Provides a record of the commands necessary to repeat a task.
- Far easier to debug a sequence of commands

   rerun script vs. reenter every command
- The script runs in its own environment



# WHAT MAKES SHELL SCRIPTING USEFUL?

- Several obvious reasons:
  - Massive time-savings versus working interactively

Shell scripts are computer programs written in a highlevel language that extend the functionality of the shell and simplify your workflow.

rerun script vs. reenter every command

• The script runs in its own environment



- Your script skills are proportional to your proficiency with the interactive shell environment.
  - The more you know about the shell's language, the more you can do in scripts...
  - ...and the more sophisticated you seek to make your scripts, the more you tend to learn about the shell's language.

Dare to be first.

- Your script skills are proportional to your proficiency with the interactive shell environment.
- Understand the Unix filesystem, privileges
  - A script is a file: has a user and group owner and user-group-other permissions
  - "Execute" bit(s) must be set for script to be directly executable

• Your script skills are proportional to your proficiency with the interactive shell environment.

Dare to be first.

- Understand the Unix filesystem, privileges
- Proficiency in a text editor
  - From the shell: vi, vim, emacs, pico, nano, ...
  - On your PC

You must be aware of line encoding when editing shell scripts for Unix/Linux on Windows! Windows text files demarcate end-of-line differently than Unix/Linux, and many Unix/Linux programs will not function properly when given a Windows text file.

•

The *dos2unix* command can convert the file to Unix/Linux line encoding. A better option is to use a Windows text editor that can save files with Unix/Linux line encoding.



Dare to be first.

- Your script skills are proportional to your proficiency with the interactive shell environment.
- Understand the Unix filesystem, privileges
- Proficiency in a text editor



- Your script skills are proportional to your proficiency with the interactive shell environment.
- Understand the Unix filesystem, privileges
- Prc "Script" is a generic term that typically denotes an interpreted (not compiled) programming language. Scripts can be written for shells, Perl, Python, PHP, AppleScript, JavaScript, Lua, TCL, et al. **This tutorial focuses on Bash shell scripting.**

Dare to be first.

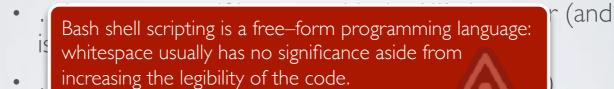
- Each line of text in the script...
  - ...can have leading and trailing whitespace (which is ignored when executed)
  - ... is a comment if it starts with the '#' character (and is ignored when executed)
  - ...may be blank (and is ignored when executed)
  - ... contains <u>all of</u> or <u>a portion of</u> a shell command
  - ... can be continued on the next line if it ends with the '\' character



• Each line of text in the script...

٠

• ...can have leading and trailing whitespace (which is ignored when executed)





Id

#### • Example:

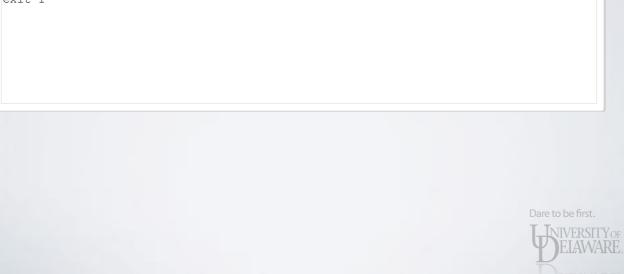
# My first script
#
# All lines above (and this one) are ignored on execution.
egrep -r \
 '^2015-01-14' \
 /var/log/messages\* \
 /var/adm/syslog\*
 # All done
 echo "All done."
 exit 1

Dare to be first.



#### • What the shell sees when you execute this script:

egrep -r '^2015-01-14' /var/log/messages\* /var/adm/syslog\* echo "All done." exit 1



- Note that this shell script is truly just a sequence of commands.
  - No variables, loops, conditionals, or other programming constructs
  - This script would be functional in most any Unix/ Linux shell — Bash, Csh, Tcsh, Zsh!

• Note that this shell script is truly just a sequence of commands.

\$ bash my\_first\_script egrep: /var/adm/syslog\*: No such file or directory "All done."

\$ csh my\_first\_script
"All done."

\$ tcsh my\_first\_script
"All done."

\$ zsh my\_first\_script
my\_first\_script:7: no matches found: /var/adm/syslog\*
"All done."

Dare to be first.

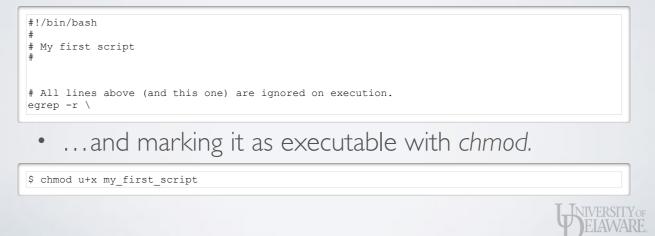
- Executing the script
  - In this example, a shell was started and asked to execute the script:

\$ bash my\_first\_script

• You need to know in which shell the script should be executed!

Dare to be first.

- Executing the script
  - A script can be made to behave like any other executable by adding a *hash-bang* (a.k.a. *shebang*)...



#### • Executing the script

\$ ls -1 my\_first\_script -rwxr--r- 1 user group 227 Jan 15 11:00 my\_first\_script

\$ ./my\_first\_script egrep: /var/adm/syslog\*: No such file or directory "All done."

Dare to be first.

#### • Executing the script

\$ ls -1 my\_first\_script -rwxr--r-- 1 user group 227 Jan 15 11:00 my\_first\_script

\$ ./my\_first\_script
egr: /var/adm/syslog\*: No such file or directory

To execute this script I prefixed its name with './' — why did I do this?

Dare to be first.

ELAWARE.

- Executing the script
  - Another often-used hash-bang variant:



Most often seen in Python scripts.

.



• Most often seen in Python scripts.

## SCRIPT LAYOUT/DESIGN

- Executing the script
  - Script can also be executed in the current shell without starting a new process with its own environment.
  - Usually used to setup path environment variables.

\$ echo \$INTEL_PATH	
\$ echo \$MANPATH	
/usr/share/man:/usr/local/share/man	
\$ source /opt/intel/setup_fortran.sh \$ echo \$INTEL PATH	
/opt/intel/2013/sp1	
\$ echo \$MANPATH	
/opt/intel/2013/sp1/man:/usr/share/man:/usr/local/share/man	
opt/intel/2013/sp1/man:/usr/share/man:/usr/local/share/man	
	NIVERS
	JELAV

Most often seen in Python scripts.

٠

- Where would mathematics be without variables to represent unspecified values?
  - The same goes for computer programming
  - Variables help to generalize program code
    - Rather than entering all data when the program is written, variables defer value specification to the time of execution

Dare to be first.

**INIVERSITY** OF

- Command line arguments behave like variables
- Rather than editing and recompiling the 'ls' command each time I use it, I provide one or more paths:

	1 root root	11528 Nov 11 2010 4	11toppm	
	1 root root	112224 Nov 22 2013 a		
-rwxr-xr-x.	1 root root	93848 Feb 15 2012 a	1 5	
-rwxr-xr-x 1 :	1 root root	50248 Jul 23 10:18 a	ıb	
			Dare to be	TITSL.

• As we will see, command line arguments actually ARE variables within a shell script

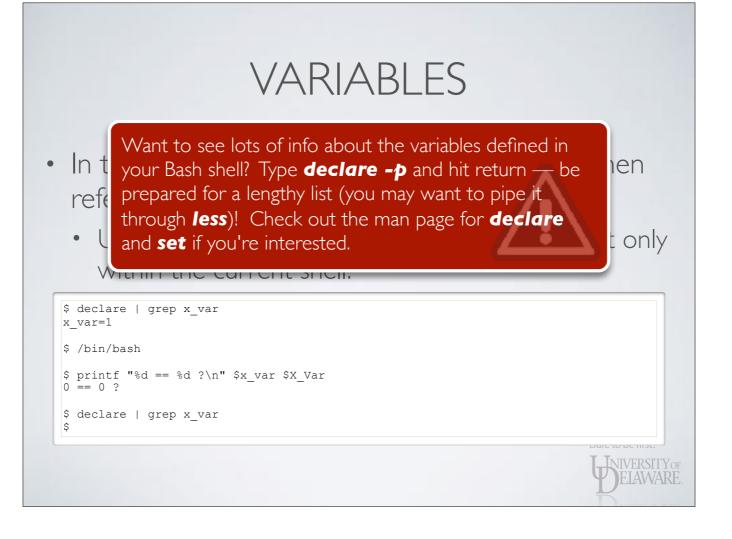


- In the Bash shell, variables use a leading '\$' when referenced but are declared without it
  - Names are case sensitive



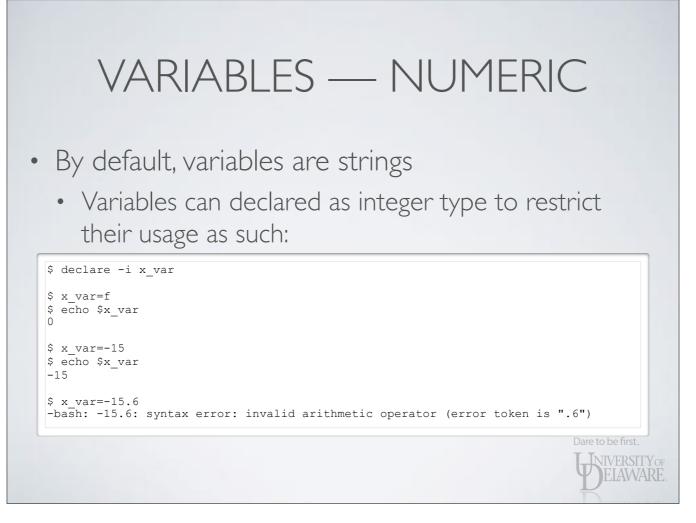
- In the Bash shell, variables use a leading '\$' when referenced but are declared without it
  - Until they are explicitly exported, variables exist only within the current shell:

<pre>\$ declare   grep x_var x_var=1</pre>	
\$ /bin/bash	
<pre>\$ printf "%d == %d ?\n" \$x_var \$X_Var 0 == 0 ?</pre>	
\$ declare   grep x_var \$	
	ELAWARE
	P

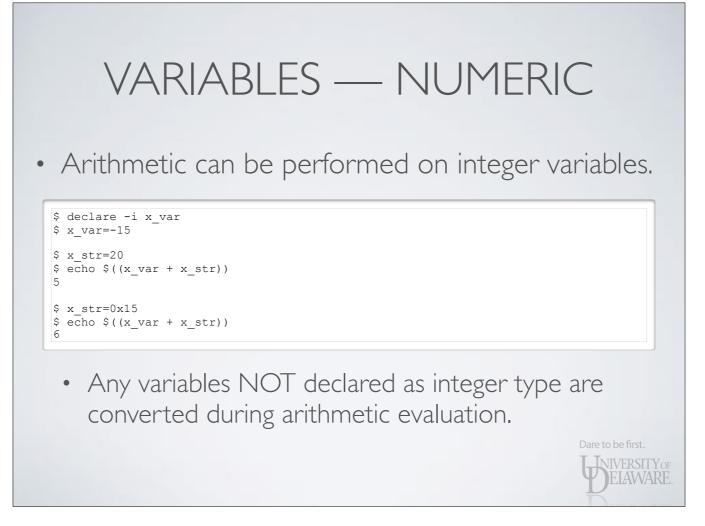


- When your script executes, it inherits any environment variables that were defined in the parent process.
  - The script is free to change their values and reexport them.
  - This influences child processes of the script, but NOT the script's parent process.

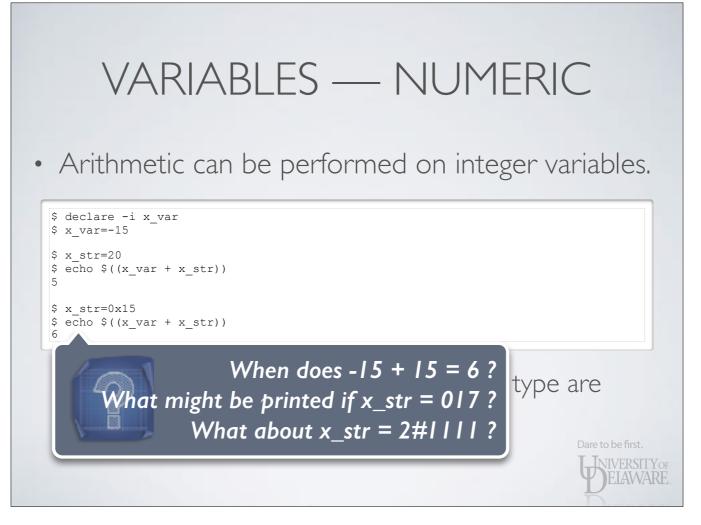
- · As a parent, I determine the rules by which I raise my children.
- $\cdot\,$  When they become parents, they are free to reuse my rules or make their own.
- · BUT, I cannot travel back in time and change the rules my parents used to raise me!



• Declaring a variable as integer restricts the values that can be assigned to it.



Bash will convert strings containing hexadecimal and octal values.



· Bash will convert strings containing hexadecimal and octal values.

Ari	thmetic	Bit	twise
+	addition	&	and
-	subtraction		or
*	multiplication	۸	xor
/	division	~	negation
%	modulo	<<	shift left
**	exponentiation	>>	shift right
	Ot	her	
expr	? expr : expr	ternary o	operator
(expr)		sub-expression	(precedence)

## **SPECIAL VARIABLES**

\$?	exit status of last program executed	
\$\$	pid of this program pid of last-started background job	
\$!		
\$#	number of command line aguments	
\$*	command line arguments as string <sup>†</sup>	
\$@	command line arguments as string <sup>†</sup>	
\$0	name of this program	
\$I <b></b> \$9	command line argument 1, 2,, 9	
\$_	starts as full path to script, changes to last command's argument list	

<sup>†</sup>The \$\* and \$@ behave differently when inside double quotes.



## **SPECIAL VARIABLES**

many programs accept shift command discarc	sitional variables (\$1 \$9) but more than nine arguments. The ds \$1 and moves all positional ndex — including moving the
\$@	command line arguments as string <sup>†</sup>
\$0	name of this program
\$I \$ <b>9</b>	command line argument 1, 2,, 9
	starts as full path to script, changes

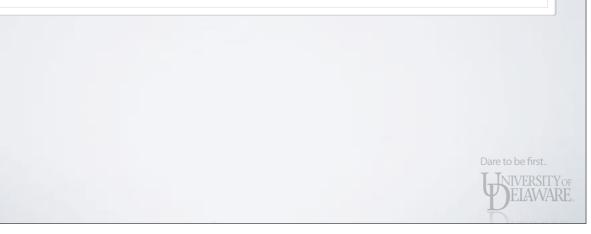
UNIVERSITY OF ELAWARE

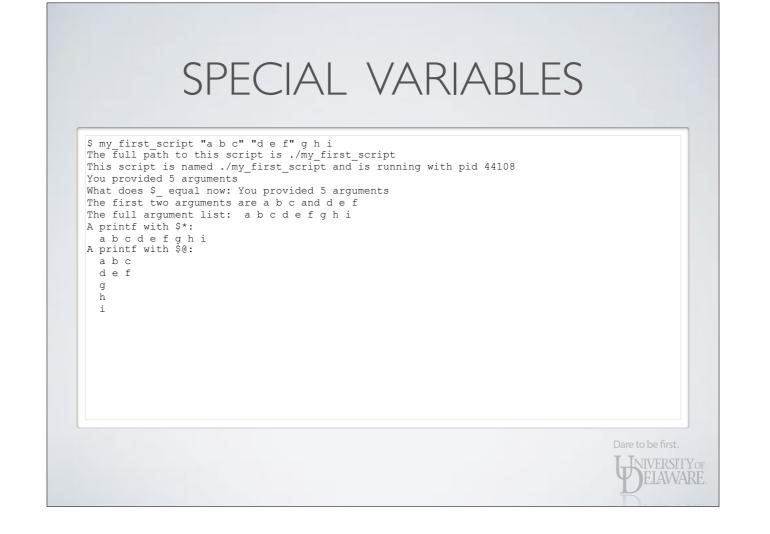
<sup>†</sup>The \$\* and \$@ behave differently when inside double quotes.

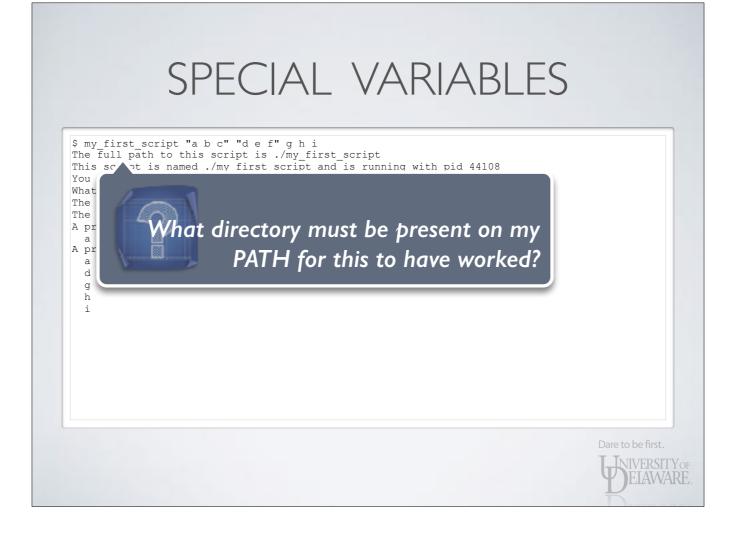
## SPECIAL VARIABLES

#### #!/bin/bash

echo "The full path to this script is \$\_"
echo "This script is named \$0 and is running with pid \$\$"
echo "You provided \$# arguments"
echo "What does \\$\_ equal now: \$\_"
echo "The first two arguments are \$1 and \$2"
echo "The full argument list: \$0"
echo "The full argument list: \$0"
echo "A printf with \\$\*:"
printf " \$s\n" "\$0"







- If variables can take on arbitrary values, then it is important to be able to test their value.
  - A conditional evaluates an expression and executes a different set of statements based upon its value.

Dare to be first.

FILAWARE

• Implies *branching*: out-of-sequence execution of program code

- Integral to using conditionals is making use of logical expressions.
  - Test the value of an integer variable
  - Test the value of a string variable
    - Treat the string as a filepath and test file metadata
  - Create complex expressions using logic operators

Dare to be first.

**INIVERSITY** OF

• Logical expression review:

#### **Integer Comparison**

a -eq b	equal	
a -ne b	not equal	
a -gt b	greater than	
a -ge b	greater than or equal to	
a -lt b	less than	
a -le b	less than or equal to	

• Logical expression review:

#### **String Comparison**

a = b	equal
a != b	not equal
-za	variable exists and is empty
-n <i>a</i>	variable exists and is non-empty
а	equivalent to '-n a'

Dare to be first.



## • Logical expression review:

Filepath '	Tests
------------	-------

-ra	path is readable by user
-wa	path is writable by user
-x a	path is executable by user
-f a	path exists and is a file
-d a	path exists and is a directory
-ea	path exists

UNIVERSITY OF ELAWARE.

<sup>†</sup> See the "man test" for additional filepath tests.

### • Logical expression review:

#### Logic Operators

<b>expr -a expr</b> both expressions evaluate to true		
expr -o expr	either expression evaluates to true	
! expr	logical negation	
(expr)	compound grouping	

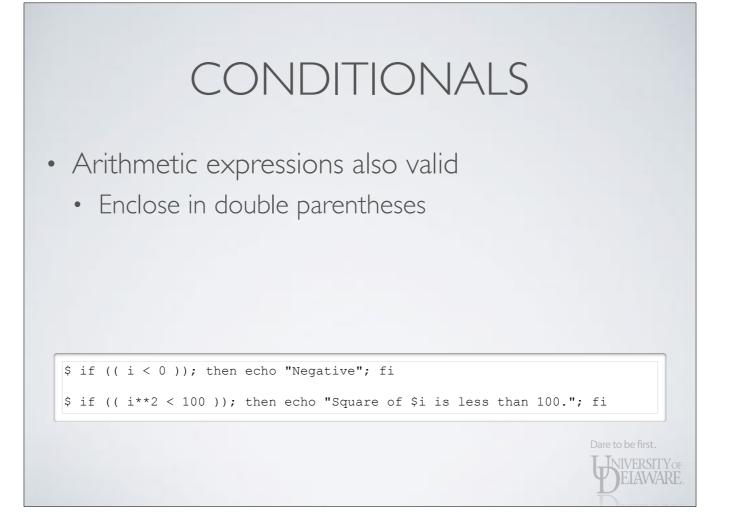
! expr	logical negation	
( expr )	compound grouping	
		Dare to be first.
		<b>UNIVERSITY</b> OF ELAWARE

FILAWARE

- Newer versions of Bash have extended tests
  - Double-bracket syntax
  - Less dependent on proper quoting
  - Accepts && and || operators
  - Regular expression matching

\$ if [[ ( -n \$str1 && -n \$str2 ) || \$int1 -gt 4 ]]; then ...

\$ if [[ \$str1 =~ ^hpc ]]; then ...



- Simple conditional
  - If a logical expression is true, perform the following commands...
  - ...otherwise, continue executing after those commands.

<pre># When expression is true, do everything up to the ' : fi</pre>	'fi"
# Continue here	
	Dare to be first.

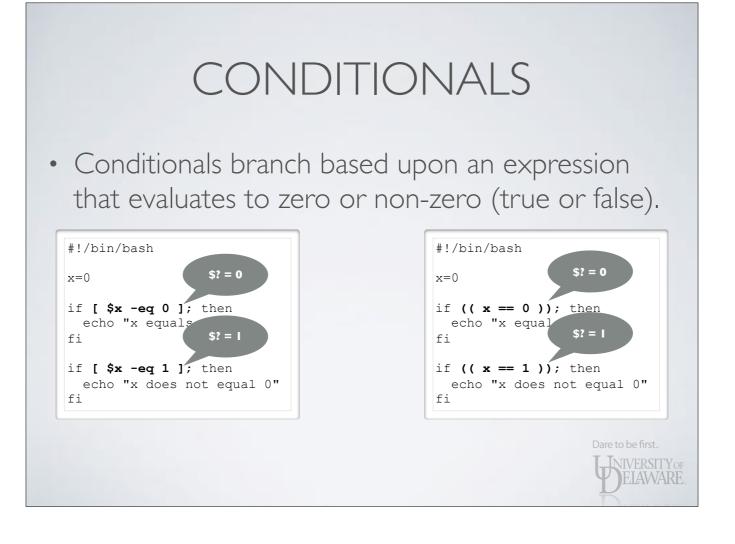
- Conditionals branch based upon an expression that evaluates to zero = true, non-zero = false
  - Most programming languages equate zero with false
  - Shell's primary job is executing programs
    - Unix programs return zero on success, non-zero on error
    - Behavior of conditionals follows this interpretation of success/failure

Dare to be first.

**INIVERSITY** OF

• Conditionals branch based upon an expression that evaluates to zero or non-zero (true or false).

x=0	x=0
if <b>[ \$x -eq 0 ];</b> then echo "x equals 0" fi	<pre>if (( x == 0 )); then     echo "x equals 0" fi</pre>
if <b>[ \$x -eq 1 ];</b> then echo "x does not equal 0" fi	<pre>if (( x == 1 )); then     echo "x does not equal 0" fi</pre>



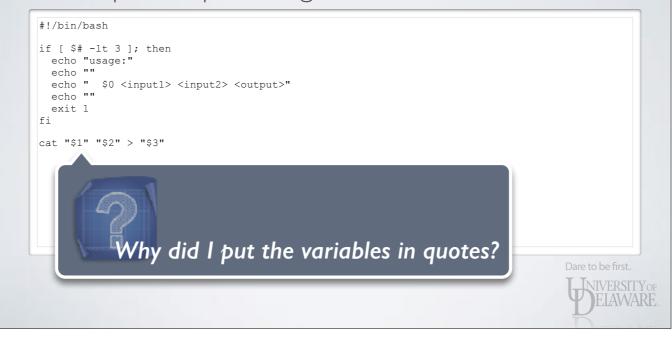
• Conditionals branch based upon an expression that evaluates to zero or non-zero (true or false).

\$ [ \$x -eq 0 ] \$ echo \$? 0	
\$ [ \$x -eq 1 ] \$ echo \$? 1	

• Example: required argument count

# #!/bin/bash if [ \$# -lt 3 ]; then echo "usage:" echo "" echo " fit cat "\$1" "\$2" > "\$3" Dare to be first. UVERSITION

• Example: required argument count



• Example: required argument count, files exist

#### #!/bin/bash if [ \$# -lt 3 ]; then echo "usage:" echo "" echo " \$0 <input1> <input2> <output>" echo "" exit 1 fi if [ ! -f "\$1" -o ! -r "\$1" ]; then echo "ERROR: invalid file specified: \$1" exit 1 fi if [ ! -f "\$2" -o ! -r "\$2" ]; then echo "ERROR: invalid file specified: \$2" exit 1 fi cat "\$1" "\$2" > "\$3" ELAWARE.

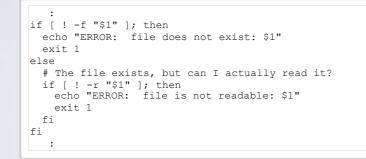
• else : ... execute this code if expression is false...

<pre># When expression is false, do everything up to the "fi" : fi # Continue here</pre>	: else	
fi # Continue here…		
# Continue here…	•	
	1	
Dare to be firs	+ Continue here	
Dare to be first	# Continue here	
Dare to be first	ŧ Continue here…	
	ŧ Continue here…	
	# Continue here…	Dare to be first

• else : Executing code when conditional evaluates false.

Dare to be first.

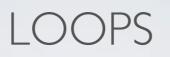
FILAWARE



• Conditionals can be nested

• *elif* : Multi-level conditional — sequence of tests versus nesting of tests

<pre>if [ ! -f "\$1" ]; then     echo "ERROR: file does not exist: \$1" </pre>	
exit 1 elif [ ! -r "\$1" ]; then	
echo "ERROR: file is not readable: \$1"	
exit 1	
elif [ ! -s "\$1" ]; then echo "WARNING: file is empty: \$1"	
fi	
:	
	Dare to be fi
	Dare to be f



- Scripting is useful because it minimizes one's retyping of oft-used shell code
- In like fashion, loops allow a sequence of statements to be executed zero or more times
  - Iterate over a set of items
  - Iterate a fixed number of times
  - Iterate until a conditional is satisfied
    - Iterate indefinitely

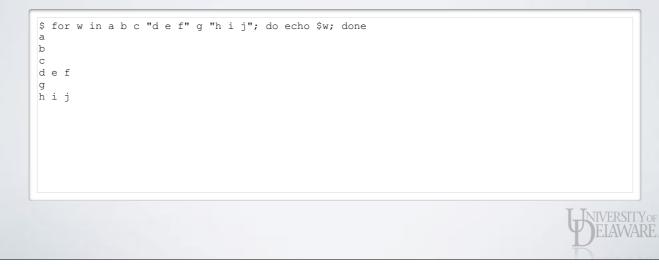
## LOOPS — SET OF ITEMS

• Given a string containing *words* separated by whitespace, perform a sequence of statements for each *word* 

; done		
# Continue here…		
	r	

# LOOPS — SET OF ITEMS

• Given a string containing words separated by whitespace, perform a sequence of statements for each word



#### LOOPS — SET OF ITEMS • Given a string containing words separated by whitespace, perform a sequence of statements for each word #!/bin/bash #!/bin/bash for w in "\$0"; do for w in \$0; do echo \$w echo \$w done done \$ ./echo\_args a b "c d e" \$ ./echo\_args a b "c d e" а а b b c d e С d е Dare to be first. **UNIVERSITY** OF ELAWARE

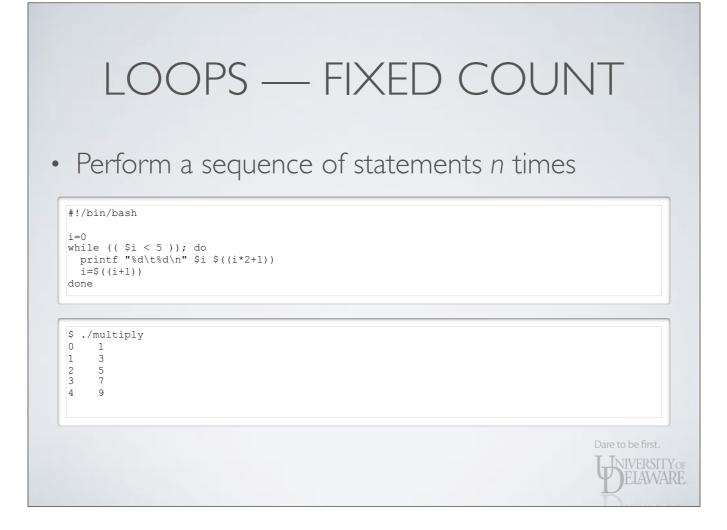
Iterate over the arguments to the script

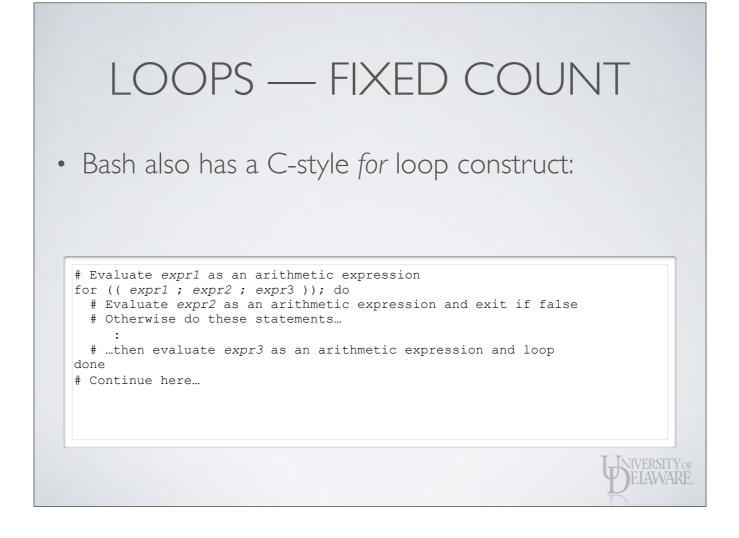
Illustrates the difference between quoted and unquoted use of \$@

# LOOPS — FIXED COUNT

• Perform a sequence of statements, exit loop when conditional expression is **false** 

:	uated to false,	, do these sta	cementos	
#then re-test	t expr			
done				
# Continue here…				





LOOPS — FIX	ED COUNT
• Bash also has a C-style for	loop construct:
<pre>#!/bin/bash for (( i=0 ; i &lt; 5 ; i++ )); do     printf "%d\t%d\n" \$i \$((i*2+1)) done</pre>	
<pre>\$ ./multiply 0 1 1 3 2 5 3 7 4 9</pre>	
	Dare to be first.

## LOOPS — INDEFINITE

Dare to be first.

- Loops may also run indefinitely (usually until a condition is met)
  - Exit via one or more conditionals inside loop

#### #!/bin/bash

```
while (( 0 == 0 )); do
  echo "Trying to ping hostname.domain.net..."
  ping -c 1 -t 1 hostname.domain.net > /dev/null 2>&1
  if [ $? -eq 0 ]; then
    break
  fi
  sleep 5
  done
  echo "I was able to ping hostname.domain.net."
```

## LOOPS — INDEFINITE

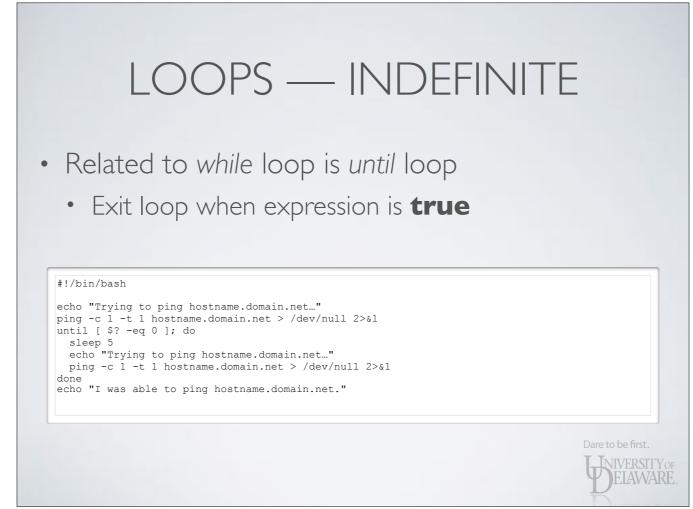
Dare to be first.

FIVERSITY OF ELAWARE

- Loops may also run indefinitely (or until a condition is met)
  - Exit via loop's logical expression

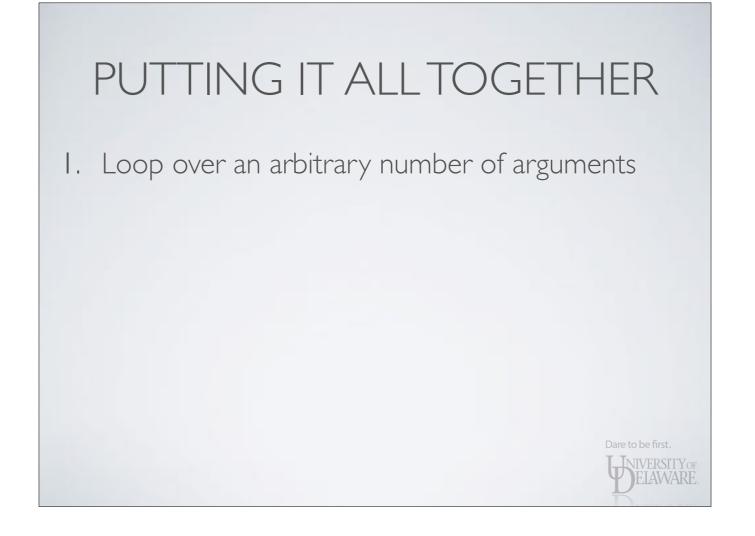
#### #!/bin/bash

echo "Trying to ping hostname.domain.net.."
ping -c 1 -t 1 hostname.domain.net > /dev/null 2>&1
while [ \$? -ne 0 ]; do
 sleep 5
 echo "Trying to ping hostname.domain.net.."
 ping -c 1 -t 1 hostname.domain.net > /dev/null 2>&1
done
 echo "I was able to ping hostname.domain.net."



- Write a script that will accept any number of file paths.
  - For each path, determine if the path exists or not
  - If it exists, determine whether it's a directory, regular file, or other type of entity





# So long as we have at one or more arguments, do the following loop: while [ \$# -gt 0 ]; do # The path we're interested in is \$1:

# Discard \$1 and move all other arguments down one index: shift

done



Dare to be first.

**INIVERSITY** OF

- I. Loop over an arbitrary number of arguments
- 2. For each argument, does path exist?

# So long as we have at one or more arguments, do the following loop: while [ \$# -gt 0 ]; do # The path we're interested in is \$1: if [ -e "\$1" ]; then

else echo "ERROR: path does not exist: \$1" fi

# Discard \$1 and move all other arguments down one index: shift done

> Dare to be first. **UNIVERSITY** OF

ELAWARE

Dare to be first.

INIVERSITY

- I. Loop over an arbitrary number of arguments
- 2. For each argument, does path exist?
- 3. If path exists, what kind is it?

#### #!/bin/bash

# So long as we have at one or more arguments, do the following loop: while [ \$# -gt 0 ]; do # The path we're interested in is \$1: if [ -e "\$1" ]; then # What is it? if [ -d "\$1" ]; then file\_kind='directory'
elif [ -f "\$1" ]; then file kind='regular file' else file kind='other' fi printf "%-20s %s\n" "\$file kind" "\$1" else echo "ERROR: path does not exist: \$1" fi # Discard \$1 and move all other arguments down one index: shift done **UNIVERSITY** OF ELAWARE

\$ ls -l ./file\_tests
-rw-r--r- 1 frey everyone 483 Jan 21 21:23 ./file\_tests

\$ chmod u+x ./file\_tests

\$ ls -l ./file\_tests
-rwxr--r- 1 frey everyone 483 Jan 21 21:23 ./file\_tests

\$ ./file\_tests /opt /opt/icu /opt/icu/attic /opt/icu/attic/icu4c-4\_0-src.tgz \
> /opt/non-exist /dev
directory /opt
directory /opt/icu
directory /opt/icu/attic
regular file /opt/icu/attic/icu4c-4\_0-src.tgz
ERROR: path does not exist: /opt/non-exist
directory /dev

UNIVERSITY OF ELAWARE

- Additional pieces
  - If no arguments provided, show help text
  - Options:
    - --keep-running : path doesn't exist, show error but don't exit

Dare to be first.

FILAWARE.

#!/	bin/	bash
-----	------	------

<pre># No arguments? User doesn't know what he's doing! if [ \$# -eq 0 ]; then    cat &lt;<eot pre="" usage:<=""></eot></pre>	
<pre>\$0 {options} [path1] {[path2] [pathn]}</pre>	
options:	
keep-running do not exit on error	
EOT exit 1 fi # Exit on error: keep_running=0	
	Dare to be first.

	<pre>elif [ "\$1" == '' ]; then    shift    break else    echo "ERROR: unknown option: \$1"    exit 1    fi else    break fi shift done</pre>	
--	---	--

# So long as we have one or more arguments, do the following loop: while [ \$# -gt 0 ]; do # The path we're interested in is \$1: if [ -e "\$1" ]; then # What is it? if [ -d "\$1" ]; then file kind='directory' elif [ -f "\$1" ]; then file kind='regular file' else file kind='other' fi printf "%-20s %s\n" "\$file\_kind" "\$1" else echo "ERROR: path does not exist: \$1" if [ \$keep running -eq 0 ]; then exit 1 fi fi # Discard \$1 and move all other arguments down one index: shift done