14.2.6 Shell Environments, Bash Variables and Parameters Facts

To write helpful bash shell scripts, you need a good understanding of the environment that the script runs in including environment variables, shell variables, bash shell parameters, user variables, and expansions,

This lesson covers the following topics:

- Environment Variables, Shell Variables and Shell Environment Types
- The time command

Environment Variables, Shell Variables and Shell Environment Types

Linux is a multi-processing operating system, and each shell environment is spawned as a process.

- As a process is created, environment information is gathered from a variety of files and settings on the system, and made available to the shell process.
- This shell environment is implemented as key-value pairs or environment variables.
- Each environment variable has a name that's assigned a value or multiple values.
- Bash startup routines and bash shell startup scripts can add shell variables to the shell process.

The way that shell variables are added to a shell process depends on the shell environment type.

- An interactive shell reads from standard-input (the keyboard) and writes to standard-output (the display screen). There are two kinds of interactive shells.
    - A login shell requires a user to provide a user ID and password. If you logged in using the Linux console or through a remote SSH session, it's a login shell.
    - A non-login shell does not require a user ID and password. The terminal windows in the Gnome desktop environment is an example of a non-login shell.
    - Bash shell startup scripts that add shell variables are only run in interactive shells.
- A non-interactive shell does not interact with the user. Examples of a non-interactive shell are when a cron job starts or when a shell script is run.
    - Bash shell startup scripts do not run in non-interactive shells.

Scripts and Environment Variable Inheritance

A bash shell runs in a Linux process. Each Linux process has a parent process.

- To run a script from a parent bash shell:
  - A child process is created.
  - The child process opens a bash shell.
  - The script is run in the child bash shell.
- The child shell inherits environment variables from its parent process.
  - A copy of the parent shell's environment variables is given to the child shell.
  - Any environment variable added to the parent shell after the child shell is created won't be available to the child shell.
  - The child shell can manipulate these environment variables without affecting the parent's environment variables.

Environment Variables

Environment variables are mostly used by the shell. The following commands are used when working with environment variables.

Command	Description	Examples
printenv	When a environment variable name is added as an argument, **printenv** displays the environment variable's value. When no arguments are added, **printenv** displays a list of environment variables and their values.	**printenv PATH** This command displays the value of the PATH environment variable. **printenv** This command lists all environment variables and their values.
variable name=value	Creates a shell variable with the given name and value	**training=TestOut** This command creates the shell variable with the name of "training" that has the value "TestOut".
export *variable name*	Converts a shell variable into an inheritable environment variable.	**export training** This command converts the training variable into an environment variable.
delare -x *variable name=value*	Creates and exports an environment variable at the same time.	**declare -x training=TestOut** This command creates an environment variable named "training" that has the value "TestOut".

Shell Variables

Many people confuse shell variables with environment variables, since their names are in uppercase and they seem to be available in every shell session.

- Shell variables aren't inherited like environment variables.
- Most shell variables are created by shell startup scripts.

One way to see the difference between environment variables and shell variables is to run two commands, **printenv** and **set**, and compare the results.

- The **printenv** command lists only environment variables.
- The **set** command lists all variables, including environment variables.
- Any variable listed in the **set** command output that isn't in the **printenv** output is a shell variable.

Another way to see the difference between environment variables and shell variables is to add the **printenv** and **set** commands to a script.

- If **printenv** is run interactively, its results are identical to a script that runs the **printenv** command.
- The **set** command run interactively will give different results than the **set** command run in a script.
    - This is due to the shell variables that are added by the shell startup files that are only run in interactive shell environments.

Shell variables are created just like environment variables by entering the variable's name followed by the equal sign and then the value to be assigned to the name.

Positional Parameters

Positional parameters are a series of special variables that contain the arguments given when the shell is invoked. The names of these special variables are digits, so they're referenced by $1, $2, $3, $4, and so on.

Script	Script Execution	Description
[auser@mylinux bin]$ cat testparams #!/bin/bashbr> echo "Positional Parameters" echo '$1 = ' $1 echo '$2 = ' $2	[auser@mylinux bin]$ testparams TestOut is Great Positional Parameters $1 = TestOut $2 = is $3 = Great	The positional parameters are set to the arguments given when the script is run.

echo '$3 = ‘ $3 exit 0		
[auser@mylinux bin]$ cat testparams2 #!/bin/bash echo "Positional Parameters" echo '$1 = ' $1 echo '$9 = ' $9 echo '$10 = ' ${10} echo '$11 = ' ${11} exit 0	[auser@mylinux bin]$ testparams2 one 2 3 4 5 6 7 8 nine ten eleven Positional Parameters $1 = one $9 = nine $10 = ten $11 = eleven	If you have more than nine arguments are given, the name of the positional parameter will use two digits, like $10, $11, etc. When $10 is used, the bash shell interprets this as the $1 parameter followed by a zero (0). When referenced, it must be enclose the parameter in braces: ${10}. This is a shell expansion

Special Parameters

The following is a list of special parameters that can only be referenced using the dollar sign ($*, $#, $?, etc.) Their values are set and maintained by the bash shell.

Special Parameter	Description
*	The positional parameters as a single text string.
@	The position parameters as an array.
#	The number of positional parameters.
?	The exit status of the last executed command.
-	The option flags from the last set command.
$	The process ID (PID) of the shell.
!	The process ID of the last background job.
0	The name of the shell or shell script.
_	The last argument of the previous command.

User Variables

In the bash shell, user variables are treated the same as environment variables and shell variables. Consider creating them with lowercase names to avoid environment and shell variables being overwritten with new values.

- In most cases, create user variables at the same time as you assign them a value.
    - Enter the name of the variable, the equal sign (=), and then the value to assign the variable.
- Reference the variable using the dollar sign ($).
    - When a dollar sign followed by a variable name is encountered in a bash command, the value of the variable is used instead of the variable name.
    - In bash, this is described as expanding the variable.
- There are a few bash commands, like the **read** command, that will create a variable and assign a value to it.
- A user variable can be created using the **declare** command.
    - The **declare** command can be used every time to create a variable.
    - The **declare** command is mostly used with shell arithmetic.
- A variable can be deleted using the **unset** command.

Integer Variables and Shell Arithmetic

All bash variables are stored as character strings. The **declare -i** command will treat the variable as an integer.

- An integer is a whole number, like the numbers you use to count.
    - An integer can be positive, negative, or zero.
- The declare -i command allows the variable to be assigned a value using shell arithmetic.
- Shell arithmetic uses:
    - The plus character (+) for addition
    - The minus or dash character (-) for subtraction
    - The asterisk character (*) for multiplication
    - The slash character (/) for division.
- Other operators are described on the bash man page.
- To use shell arithmetic:
    - Enter the variable and the equal sign (=), and then enter an arithmetic equation.
        - Equations can be made up of variables and operators with no spaces between them
        - Parenthesis can be used to specify the order that the arithmetic operations are performed.
    - Don’t use the dollar sign to use the value for a variable.

The following examples illustrate the use of shell arithmetic.

Example	Results	Description
x=4 declare -i y=x+4 echo '$y = ' $y	$y = 8	The variable y is assigned the value x+4 which is 4+4 or 8.
declare y=x+4 echo '$y = ' $y	$y = x+4	The variable y is assigned the literal characters "x+4".
a=badchars echo '$a = ' $a declare -i z=a+7 echo '$z = ' $z	$a = badchars $z = 7	The variable z will accept shell arithmetic, but the variable a is assigned character data. When a is used in an equation, the value of zero (0) is substituted. So the variable z is assigned the value a+7 which is interpreted as 0+7 or 7.
x=4 declare -i b="John" echo '$b = ' $b b=x+20 echo '$b = ' $b	$b = 0 $b = 24	The variable b will accept shell arithmetic, so the value of zero (0) is substituted for character data. So b="John" assigns a zero to the variable b. Once a variable is declared as an integer, it will always accept shell arithmetic. So b=x+20 assigns 4+20 or 24 to the variable b.

Storing Arrays in Variables

You can assign a variable multiple values by declaring it as an array. There are two types of arrays, indexed and associative. The following examples illustrate the use of arrays.

Array Type	Example	Results	Description
indexed	[auser@mylinux bin]$ cat arrays1 #!/bin/bash declare -a namearray=(Bob Sue Joe Jane) namearray[7]=George	[auser@mylinux bin]$ arrays1 $namearray[0] = Bob $namearray[1]	Indexed arrays are create **declare -a** command. The named namearray is initia elements beginning with t of zero (0). Index numbers consecutive. To reference

	echo '$namearray[0] = ' ${namearray[0]} echo '$namearray[1] = ' ${namearray[1]} echo '$namearray[2] = ' ${namearray[2]} echo '$namearray[3] = ' ${namearray[3]} echo '$namearray[7] = ' ${namearray[7]} echo "This won't work" echo '$namearray[2] = ' $namearray[2] exit 0	= Sue $namearray[2] = Joe $namearray[3] = Jane $namearray[7] = George This won't work $namearray[2] = Bob[2]	array, add the index numb the variable name within b enclose the whole express
associative	[auser@mylinux bin]$ cat arrays2 #!/bin/bash declare -iA agearray=([Bob]=21 [Sue]=19 [Joe]=22 [Jane]=25) echo '$agearray[Bob] = ' ${agearray[Bob]} echo '$agearray[Sue] = ' ${agearray[Sue]} echo '$agearray[Joe] = ' ${agearray[Joe]} echo '$agearray[Jane] = ' ${agearray[Jane]} agearray[Bob]=agearray[Sue]+agearray[Joe] echo '$agearray[Bob] = ' ${agearray[Bob]} echo '$agearray[Sue] = ' ${agearray[Sue]} echo '$agearray[Joe] = ' ${agearray[Joe]} echo '$agearray[Jane] = ' ${agearray[Jane]} exit 0	[auser@mylinux bin]$ arrays2 $agearray[Bob] = 21 $agearray[Sue] = 19 $agearray[Joe] = 22 $agearray[Jane] = 25 $agearray[Bob] = 41 $agearray[Sue] = 19 $agearray[Joe] = 22 $agearray[Jane] = 25	Associative arrays are crea **declare -A** command. In t associative array named a declared with the -i option arithmetic. An element of referenced using the varia bracketed element name, Array elements can be use arithmetic. The command 'agearray[Bob]=agearray[S assigns the value of 41 (Su age) to the Bob element.

Shell Expansions

There are may types of shell expansions. Two important expansions are variable expansion and command expansion.

Variable expansion is known as parameter expansion and uses the ${} construct. The following examples illustrate parameter expansion.

Example	Results	Description
echo '$namearray[2] = ' ${namearray[2]}	namearray[2] = Joe	Parameter expansion is required to get the proper results when an array element is referenced.

location="NY 10014" echo ${location:3}	10014	An offset operator is added to the parameter expansion. This is a sub-string expansion that starts at the offset that follows the colon, in this case, 3, and continues to the end of the value. The offset begins counting at 0. Beginning with an offset of 3 and continuing to the end, this gives “10014.”
location="NY 10014" echo ${location/10014/10032}	NY 10032	Pattern substitution is added to the parameter expansion. The command looks for the pattern “10014” in the location value and substitutes “10032.”

Command expansion is also known as command substitution and uses the $() construct. Command substitution uses the output of a command to replace the command itself. The command within the parentheses is run, and the output is substituted. Often, command substitution is used as an argument for another command or to set a variable. The following examples illustrate command substitution.

Example	Results	Description
find -maxdepth 1 -mtime +30	./a.dat ./b.dat	The find command can be used in a command expansion. In this case the a.dat and b.dat files are the only two files in a directory that are over 30 days old.
mv –t ./old $(find -maxdepth 1 -mtime +30)	No output unless an error is generated	This command would be translated to **mv –t ./old ./a.dat ./b.dat** which would move two files to the 'old' directory.
lastfriday=$(date --date="last Friday") echo "The file was backed up on $lastfriday"	The file was backed up on Fri Feb 22 00:00:00 PST 2019	The first command in this example assigns the output of the date command to the lastfriday variable.
lastfriday=`date --date="last Friday` echo "The file was backed up on $lastfriday"	The file was backed up on Fri Feb 22 00:00:00 PST 2019	The backquote (`) can be used for command substitution. The backquote on a keyboard is usually to the left of the 1 key. Don't confuse the backquote with the single quote (') that is next to the Enter key.