**VARIABLES**

Used to store a value by name

**To Create**

VARIABLE\_NAME=VALUE

* x=10 if x already exist it is assigned new value
* x=”10 11 22” value containg space use quotes
* don’t use white space around =

**To get the value**

echo ${variable name}

**Varaible Names**

* Only letter, number and underscore are allowed
* 1st character shld be a letter or an underscore
* Variable names are case-sensitive
* Use lowercase variable name as best practise

An uninitialized variable has a "null" value -- no assigned value at all (*not* zero!).

${Variable\_name)

$(command)

Example:

hello="A B C D"

echo $hello # A B C D

echo "$hello" # A B C D

As you see, echo $hello and echo "$hello" give different results.

Why?

# =======================================

# Quoting a variable preserves whitespace.

other\_numbers="1 2 3"

If there is whitespace embedded within a variable, 83 #+ then quotes are necessary.

**Setting up Null variable**

hello=

Note that setting a variable to a null value is not the same as unsetting it, although the end result is the same.

**Uninitialized value**

echo "uninitialized\_variable = $uninitialized\_variable"

Uninitialized variable has null value (no value at all!). however it evaluates as 0 in an arithmetic operation

**Unset the variable**

Var=”cricket”

Unset Var

As it is uninitialized it contains null value, however it evaluates as 0 in an arithmetic operation

**Bash Variables Are Untyped**

Unlike many other programming languages, Bash does not segregate its variables by "type." Essentially, Bash variables are character strings, but, depending on context, Bash permits arithmetic operations and comparisons on variables. The determining factor is whether the value of a variable contains only digits.

Null variable transformed into an integer.

Undeclared variable transformed into an integer

**Special variable types**

**Local variable**

* Variables visible only within a code block or function

**Environmental Variable**

* Variables that affect the behavior of the shell and user interface.
* Every time a shell starts, it creates shell variables that correspond to its own environmental variables. Updating or adding new environmental variables causes the shell to update its environment, and all the shell's child processes (the commands it executes) inherit this environment.
* If a script sets environmental variables, they need to be "exported," that is, reported to the environment local to the script
* A script can export variables only to child processes, that is, only to commands or processes which that particular script initiates. A script invoked from the command-line cannot export variables back to the command-line environment. Child processes cannot export variables back to the parent processes that spawned them.

**Positional Parameter**

* Arguments passed to the scripts from the command line $0 $1 $2 $3…..
* $0 Name of the scripts
* From the command-line, however, $0 is the name of the shell
* $1 $2 $3 …First Second and n Arguments
* $\* & $@ Displays All arguments passed to the scripts
* $# Total number of arguments
* $\_ last argument of previous command
* $! Process id of last background cmd
* $$ process id of current shell
* If a script expects a command-line parameter but is invoked without one, this may cause a *null variable assignment*, generally an undesirable result. One way to prevent this is to append an extra character to both sides of the assignment statement using the expected positional parameter. variable1\_=$1\_ # Rather than variable1=$1
* # This will prevent an error, even if positional parameter is absent
* After $9, the Arguments must be enclosed in brackets, for example, ${10}, ${11}, ${12}.

**Quoting Variables**

When referencing a variable, it is generally advisable to enclose its name in double quotes. This prevents reinterpretation of all special characters within the quoted string -- except $, ` (backquote), and \ (escape).

List="one two three"

* Double quotes following an echo sometimes escape \. Moreover, the -e option to echo causes the "\t" to be interpreted as a tab.
* echo "hello\\abcd" >>>> hello\abcd

echo -e "x\ty" >>>>> x y

**Escape Variables**

* Escaping is a method of quoting single characters. The escape (\) preceding a character tells the shell to interpret that character literally **used with echo and sed**

echo "abcd\

> lfcd"

Abcdlfcd #This will print as one line.

echo "\v\v\v\v" # Prints \v\v\v\v literally

Use the -e option with 'echo' to print escaped characters.

echo –e “\v\v\v\v” Prints 4 vertical tabs

* \n means newline echo –e “my \nname is venkat \ndegree is BE”
* \t means horizontall tab
* \v means vertical tab
* \b means backspace
* \a means alert (beep or flash)
* \0xx translates to the octal ASCII equivalent of 0nn, where nn is a string of digits

**Example**

* echo "Hello" # Hello
* echo "\"Hello\" ... he said." # "Hello" ... he said.
* echo "\$variable01" # $variable01
* echo "The book cost \$7.98." # The book cost $7.98.
* echo "\\" # Results in \ & echo "\" gives syntax error
* variable=\\ echo "$variable" # \
* echo foo\

bar # Newline escaped. #foobar

**Typing variables: declare or typeset**

The declare or typeset builtins, which are exact synonyms, permit modifying the properties of variables

declare/typeset options

-r readonly [declare -r var1=1] constant variable

-i integer [declare -i number]

-a array [declare -a indices] The variable indices will be treated as an array.

-f function(s) [declare –f function name]

-x export [declare -x var3] This declares a variable as available for exporting outside the environment of the script itself. -x var=$value

**Exit and Exit Status**

* The exit command terminates a script, just as in a C program. It can also return a value, which is available to the script's parent process.
* Every command returns an exit status (sometimes referred to as a return status or exit code). A successful command returns a 0, while an unsuccessful one returns a non-zero value that usually can be interpreted as an error code.
* Likewise, functions within a script and the script itself return an exit status. The last command executed in the function or script determines the exit status.

When a script ends with an exit that has no parameter, the exit status of the script is the exit status of the last command executed in the script (previous to the exit).

#!/bin/bash

COMMAND\_1

# Will exit with status of last command.

exit

echo exit $?

$? reads the exit status of the last command executed

The equivalent of a bare exit is exit $? or even just omitting the exit.

**TEST**

**IF/ELSE CONDITIONS**

**Helo.sh**

**#!/bin/bash**

**Echo “hello world” ……………………………. ./hello.sh**

**Read inputs**

**Read –p “enter username: ” user\_var**

**Read –sp “enter password:” pwd**

**Echo**

**As array**

**Read –a names**

**Echo “names: ${name[0]} ${name[1]} ”**

**To check the status of the server**

**#!/bin/bash**

**Ping -c 2 192.22.215.18**

**If [$! –eq 0]**

**Then**

**Echo “”server ios alive”**

**Else “server is down”**

**Fi**

**#!/bin/bash**

**HOSTNAME=$(hostname)**

**CRITICAL=98**

**WARNING=90**

**CRITICAL\_MAIL=**[**FVENKAT007@GMAIL.COM**](mailto:FVENKAT007@GMAIL.COM)

**Mkdir –p /var/log/cpu**

**LOG\_FILE= /var/log/cpu/cpu\_usage- “date +%h ”.log**

**Touch $LOG\_FILE**

**For path in /**

**Do**