Shell QuickSheet

Version: 1.0.0
Date: 10/30/11

Note: This QuickSheet is relevant to Bourne derived (Bourne, Korn, Bash) *language* issues only. Unix command line utilities are *not* covered here. Compatibility varies by implementation and version - many Bourne implementations are simply links to Korn or Bash. The *generic* identifier "Korn" assumes Korn88 unless otherwise specified.

Variables

- Explicit declaration and typing is done with `typeset` in Korn and `declare` (or `typeset`) in Bash. Explicit declaration is not required, and is not used in Bourne.
- Bash and Korn support function local variables, but have different scoping rules.

Typed variables

typeset	declare	Description
-a	-a	(Normal / indexed) array
-A	-A	Associative array [Bash,Korn93]
-F *n*		Floating point with optional *n* percision [Korn93]
-i *b*	-i	Integer (w/optional base argument *b* [Korn])
-r	-r	Make variable read only
-n		Reference variable ("pointer" to another var.) [Korn93]
-u	-u	Convert on assignment to uppercase
-l	-l	Convert on assignment to lowercase
-T		Declare a (compound variable) type [Korn93]

Integer base conversion

```
HEX=ff ⇐ $HEX is a string containing ff
typeset -i 8 OCT=16#$HEX ⇐ $OCT now holds "8#377"
typeset -i 10 DEC=$OCT ⇐ Leading "8#" is within $OCT, ∴ not required
printf "%x\n" $DEC ⇐ $DEC holds 255, printf prints "ff"
```

Typed variable example

```
typeset -r MY_CONST_VAR=100
```

Find length of `$myvar`

```
length=${#myvar}
```

Arrays [Bash,Korn]

Declare & fill array

```
set -A pepboys manny moe jack [Korn]
pepboys=(manny moe jack) [Bash,Korn]
```

Declaring an array

```
typeset -a myarray [Bash,Korn] ⇐or⇒ declare -a myarray [Bash]
```

Access 4th member of array

```
GETVAL=${myarray[3]} ⇐ Indexes are 0 based
```

Print out all members of the array

```
echo ${pepboys[*]}
my_cmd "${pepboys[@]}" ⇐ Preserves whitespace
```

Count the number of members in an array

```
count=${#myarray[*]}
```

Append newvalue to an array

```
myarray=( ${myarray[*]} newvalue ) ⇐ Specialized indexing will be lost
 ↪ Use "${myarray[@]}" to preserve whitespace in array members.
myarray+=( newvalue ) [Korn93]
```

Associative Arrays [Bash,Korn93]

Declare associative array

```
typeset -A famous_people
```

Fill associative array

```
aarray=( [one]=uno [two]=dos [three]=tres )
```

Add item to array

```
famous_people[Socrates]=Philosopher
```

Access item from array

```
famous_skill=${famous_people[Hannibal]}
```

Print out all keys of the array

```
echo ${!famous_people[*]} ⇐or⇒ echo "${!famous_people[@]}"
```

Compound Variables [Korn93]

Declare compound variable with three *members*: `a`, `b`, & `c`. Explicitly type `c`.

```
myvar=( a= b= typeset -i c= )
myvar.c=4 ⇐ Set member c from previous example to 4
B=${myvar.b} ⇐ Access member b from previous example
```

if / test

↪ Note: The `then` and `fi` clauses in the following examples are omitted for space.

Numeric compare ⇐ (`-lt` < | `-gt` > | `-ne` != | `-eq` ==)

```
if (( $N > 1 )) ⇐or⇒ if (( $N == 1 )) [Bash,Korn]
if [ $N -gt 1 ] ⇐or⇒ if [ $N -eq 1 ] [Bourne]
```

String compare ⇐ (!= | = | < | >) ⇐ < and > are for sort order compare

```
if [[ $X = $Y ]] [Bash,Korn] ⇐ Use single =, but many shells allow ==
if [[ $X == $Y ]] [Korn93] ⇐ Preferred method for Korn 93, = is allowed
if [ $X = $Y ] [Bourne] ⇐ [ is a builtin or binary, [[ is a language construct
```

Ands / Ors ⇐ (`-a` && | `-o` ||)

```
if [[ $A = $B || $C = $D ]] [Bash,Korn]
if [ $A = $B -o $C = $D ] [Bourne]
```

Test for first parameter (test for potentially empty string)

```
if [[ -z $1 ]] [Bash,Korn]
if [ -z "$1" ] ⇐or⇒ if [ "$1" = "" ] [Bourne]
```

Check return value from `mycmd`

```
if mycmd > /dev/null 2>&1 [Bash,Korn] ⇐ Use $? for Bourne.
```

Extracting Substrings

```
${astrvar:offset:length} ⇐ length chars of $astrvar starting at offset
${astrvar:offset} ⇐ Remainder of the chars of $astrvar starting at offset
```

Shell / set options

-a	allexport	Export variables on creation or modification
-e	errexit	Exit script on non-zero return value, throw ERR
-x	xtrace	Print commands as run with variable expansion
-v	verbose	Print lines as read from file without variable expansion
-u	nounset	Check for unset variables
-n	noexec	Do not execute read commands (Can be used for trigger-lock)

Shell math

Add 1 to variable `$VAL`, place result in `$VAL`

```
VAL=$((VAL + 1)) [Bash,Korn] ⇐ Leading $ not required inside $(( ... ))
VAL=`expr $VAL + 1` [Bourne] ⇐ $(( ... )) works in most implementations
(( VAL++ ))⇐or⇒ (( VAL += 1 ))⇐or⇒ (( VAL = VAL + 1 )) [Bash,Korn]
```

Test conditions

`-d` *file*	*file* exists as a directory
`-e` *file*	*file* exists
`-f` *file*	*file* exists as a file
`-s` *file*	*file* exists and is larger than 0
`-r` *file*	*file* exists and is readable
`-w` *file*	*file* exists and is writeable
`-x` *file*	*file* exists and is executable
`-z` *string*	*string* is empty

↪ Additional examples are available from the `man` page for `test`.

```
if [[ -e /path/to/myfile ]] ; then echo "myfile exists." ; fi
if [[ -z $1 ]] ; then echo "Parameter missing." ; fi
```

↪ These examples are [Bash,Korn] , while "[" and "test" are [Bourne] .

Command Substitution

```
VAL=$(mycmd 2> /dev/null) ⇐ Newer version, tends to work in Bourne
VAL=`mycmd 2> /dev/null` ⇐ Older version, works in all
VAL=${ mycmd 2> /dev/null } [Korn93] ⇐ No sub-shell, allows for side effects
```

Special Variables

`$$`	-	PID of shell (frequently used in temp file naming)
`$?`	-	Last return value
`$0`	-	The current shell ⇐ Don't use `$SHELL`
`$SECONDS`	-	Seconds since shell was started
`$RANDOM`	-	A random number ⇐ Use modulus (%) to limit to a range
`$@`	-	All arguments (Also `$*` - different in seperator)
`$LINENO`	-	Current line number of script

Functions

```
function bash_korn_func [Bash,Korn]
{ echo "First parameter is $1." ; }
bourne_func () [Bourne] ⇐ Supported in all later shells
{ echo "First parameter is $1." ; }
```

Conditional commands

```
true && echo "Always print"  | true || echo "Never print"
false && echo "Never print"  | false || echo "Always print"
[ -e afile ] && echo "afile exists."
```

Pattern matching

`?(`*pattern*`)`	-	Zero or one instances of *pattern*
`*(`*pattern*`)`	-	Zero or more instances of *pattern*
`+(`*pattern*`)`	-	One or more instances of *pattern*
`@(`*pattern*`)`	-	Exactly one instance of *pattern*
`!(`*pattern*`)`	-	Anything not matching *pattern*
`~(E)`*pattern*	-	*pattern* is an extended regular expression (egrep)
`~(G)`*pattern*	-	*pattern* is an basic regular expression (grep)

```
if [[ ${STRING} = A@(da|to)m ]] ⇐ Match Adam or Atom
```

Substring pattern extraction / substitution

`${var#`*pattern*`}`	-	Delete *first* match from left, return rest
`${var##`*pattern*`}`	-	Delete *all* matches from left, return rest
`${var%`*pattern*`}`	-	Delete *last* match from right, return rest
`${var%%`*pattern*`}`	-	Delete *all* matches from right, return rest
`${var/`*pattern*`/`*string*`}`	-	Replace longest match of first occurrence
`${var//`*pattern*`/`*string*`}`	-	Replace longest match of all occurrences
`${var/#`*pattern*`/`*string*`}`	-	Replace longest match from beginning
`${var/%`*pattern*`/`*string*`}`	-	Replace longest match from end

```
theaddr=192.168.1.25 ⇐ Assign an address (example assumes class C)
network=${theaddr%.*} ⇐ Delete dot and last octet
thehost=${theaddr##*.} ⇐ Delete all octets followed by dots
echo ${password//~(E)./X} ⇐ Substitute X for every character in $password
```

Variable substitution

Expression	VAR defined return	VAR undefined return	VAR undefined set VAR to
`${VAR:-`*string*`}`	`$VAR`	*string*	
`${VAR:=`*string*`}`	`$VAR`	*string*	*string*
`${VAR:?`*string*`}`	`$VAR`	*string* to stderr, exit	
`${VAR:+`*string*`}`	*string*	NULL	

Other

```
my_cmd <<EOF
This is text that my_cmd will read from stdin as a "here document"
EOF
```

Call `cmd_failed` function when command fails (by trapping `ERR` signal) [Bash,Korn]

```
trap cmd_failed ERR ⇐ errexit option is not required, but may be appropriate
```

Pass a variable (compound, array, or other) by reference [Korn93]

```
my_function my_var ⇐ Note that my_var does not have leading $
```

Recieve a variable by reference (inside of previously named function) [Korn93]

```
typeset -n local_var=$1 ⇐ Now access $my_var as $local_var
```

Output redirection

```
echo "ERROR: Message." >&2 ⇐ Send output to stderr
acmd 2> /dev/null 1| newcmd ⇐ Capture stdout, ignore stderr
```

Universal EOL suppression

- Use the more expensive printf until EOL suppression method is determined.
- Can use if-then block instead of || and anonymous function.

```
printf "Determining method of EOL suppression..."
N=
C=
if ‘echo "X\c" | grep c > /dev/null 2>&1‘ ⇐ Bourne compatible
then    ↪ Using $(...) instead of ‘...‘ may break in Bourne
   N=-n
   C=
else
   N=
   C='\c'
fi
printf "Done.\n"

echo $N "Running my_cmd...$C" ⇐ EOL suppressed
true || { echo "Failed." ; exit ; }
echo "Done." ⇐ Normal EOL
```

Trap ERR

- If/when my_cmd returns an non-zero exit value, the script will execute the error_handler function and exit.

```
function error_handler
{
   printf "Failed.\n"
   echo "ERROR: Command failed.  Exiting now." >&2
}

trap error_handler ERR ⇐ register error handling function for ERR signal

set -e ⇐ Tell shell to exit on failure

printf "Running my_cmd..."
my_cmd
printf "Done.\n"
```

Timer with visual

- This example runs a command multiple times, while displaying a visual indicator, and then calculates the average time for each run.
- When running a single lengthy process, an alternative is to background the spinner and have it stop on a flag file.

```
typeset -i start_time=$SECONDS
typeset -i end_time=0
typeset -i total_time=0
typeset -i count=0
typeset -ir ITERATIONS=25 ⇐ This will be a read-only integer
typeset -F 3 average_time ⇐ Will print to 3 decimal places
typeset -F ftemp

printf "." ⇐ printf may not be a builtin! (Use “type” to find out.)
while (( count < ITERATIONS ))
do
   case $(( count % 4 )) in
      0) printf "\b|" ;;
      1) printf "\b/" ;;
      2) printf "\b-" ;;
      3) printf "\b\\" ;;
   esac

   my_timed_cmd > /dev/null 2>&1 ⇐ The timed command
   (( count++ ))
done

printf "\b" ⇐ Clean up the spinner

total_time=$(( SECONDS - start_time ))
ftemp=$total_time ⇐ $ftemp is used as a float “cast” here
average_time=$(( ftemp / ITERATIONS ))

echo Iterations:  $ITERATIONS
echo Total time:  $total_time seconds
echo Average time:  $average_time seconds
```

Capture more than one variable of output

- A, B, & C will capture first three space separated items, REST will capture all that remains. stderr will be ignored.

```
my_cmd 2> /dev/null | read A B C REST
echo "Third item is $C"
```

Capture more than one variable in loop

From a file

```
while read A B C REST
do
   echo $C
done < afile
```

From a command

```
my_cmd | while read A B C REST
do
   echo $C
done
```

Compound variable passed by reference

- The compound variable allows us to pass a complex set of parameters as a single option. This code is Korn 93 only.

```
function file_op
{                        ⇐ Parameter checking would be appropriate
   echo Running $1 on $2 ⇐ $1 is a string / name of a variable

   typeset -n operation=$1 ⇐ operation is a reference to compound var
   ${operation.command} ${operation.args} $2 ⇐ Run the command
   operation.last_result=$? ⇐ Save the result
   return ${operation.last_result} ⇐ Return the result
}

ALLREAD=( command=chmod
          args=664
          last_result= )

ALL_RUN=( command=chmod
          args=775
          last_result= )

WFAVOWN=( command=chown
          args=wfavorit:wfavorit
          last_result= )

file_op ALLREAD myfile            ⇐ Return value can be used here
echo Result:  ${ALLREAD.last_result} ⇐ or here
file_op WFAVOWN myfile
```

Flow Control

if-then-else block

```
if true ⇐ See the if / test section for condition examples.
then
   echo "Always"
else ⇐or⇒ elif condition ; then
   echo "Never"
fi
```

Switch statement

```
case $GRADE in
   A|B) echo "Good grade" ;& ⇐ “Fall through” to next item [Korn]
   C|D) echo "Pass" ;;
   "F") echo "Fail" ;;
   *) echo "Not Recgonized" ;;
esac
```

select loop

```
select CHOICE in Work Sleep Eat Exit
do
   echo "${CHOICE}ing."
   if [ "$CHOICE" = "Exit" ] ; then break ; fi ⇐ Leave select loop
done
```

while loop

```
while true ⇐or⇒ until false
do
   echo "Infinite loop."
   if true ; then continue ; fi ⇐ Goto the top of the loop
   echo "Never reachable."
done
```

Iterate over list

```
for X in 1 2 3
do
   echo $X
done
```

About this QuickSheet