Unit 7
Shell Types, Commands, and Functions

Unit Objectives

After completing this unit, you should be able to:

- Use shell arrays
- Define and call functions
- Use **`typeset`** command
- Use **`autoload`** functions
- Process command aliases
- Use preset aliases
- Use tracked aliases
- Use the **`whence`** command
- Understand command line processing
- Understand command line re-evaluation with **`eval`**

Defining Arrays

The Korn and Bash shells supports one-dimensional arrays:

- Arrays need not be "declared"
- Access an element of an array by a subscript to a variable name
- Any variable with a valid subscript becomes an array
- A subscript is an expression enclosed within `[ ]`
- Subscripts should lie in the range 0 to 4095 -- (ksh only)
- Variable attributes (for example, **`readonly`**) apply to all elements of the array

Caution: An entire array cannot be exported, only the 0th element

Assigning Array Elements

Just like ordinary variables, values can be assigned, and later referred to:

- Assign contents to an array element using
  ```
  array[N]=argument
  ```
- To **unset** an array and assign new values sequentially, use
  ```
  set -A array argument ...
  ```
- To simply replace existing array values with new ones, use
  ```
  set +A array argument ...
  ```

Associative Arrays in ksh93

- ksh93 allows associative arrays
- Associative arrays are indexed by string values
- Indicate an associate array with **`typeset -A`**
  - Examples:

```
$ typeset -A tax
$ tax[NJ]=6
$ tax[NM]=5
$ tax[NY]=4
```

Referencing Array Elements

The **$** notation is used to refer to the value in a variable:

- When referencing an array element use **${ }** notation
  print ${array[N]}
- To refer to all the elements of an array use an ***** or @ subscript (to give a space separated list)
  ${array[*]}** or **${array[@]}
- If you omit a subscript, it means the zeroth element
  ${array[0]}** == **$array
- To show how many elements exist within an array
  ${#array[@]}

Array Examples

```
$ list[0]="Line 0"
$ list[1]="Line 1"
$ list[3]="Line 3"
$ print $list
Line 0
$ print ${list[*]}
Line 0 Line 1 Line 3
$ print ${list[0]}
Line 0
$ print ${list[1]}
Line 1
$ print ${list[2]}

$ print ${list[3]}
Line 3
$ print $list[1]
Line 0[1]
$ _
```

fill the array list.

print the zeroth element.

print all elements.

print elements individually.

element `[2]` is null.

without `{ }` notation, we get "`$list`" + "`[1]`".

Another Array Example

Here we have the beginnings of a card game.

```
#!/usr/bin/ksh
# Usage: pickacard.ksh
# To choose a random card from a new deck
integer number=0
for suit in CLUBS DIAMONDS HEARTS SPADES
  do
  for n in ACE 2 3 4 5 6 7 8 9 10 JACK QUEEN KING
    do
    card[number]="$n of $suit"
    number=number+1
    done
  done
print  ${card[RANDOM%52]}

$ pickacard.ksh
QUEEN of DIAMONDS
$ _
```

Defining Functions

- Commands can be group together and named.
- The set of commands form the **`function`** body.
- **`function`** definitions look like:

Bourne, Korn, and Bash	***Korn and Bash***

`identifier()`
`{`
`  commands`
`}`
	`function identifier`
	`{`
	`  commands`
	`}`

- Functions:
  - Provide a means of breaking down programs into discrete units
  - Stored in memory for fast access
  - Executed, like new commands, in the current environment

Functions and Variables

Functions have different variables to the main script:

- Arguments
  - Taken as positional parameters to the function
  - Calling script **`$1-${n}`** parameters are reset on leaving the called function

- Variables
  - Declared with the **`typeset`** or **`integer`** commands (inside a Korn shell function) are "local" variables to the function
  - All other variables are "global" in the Script
  - The "scope" of a "local" variable includes all functions called from the current function

function Example

A useful function...

```
#  Handy for usage errors in Shell Scripts
#  Invoke function usage with arguments: script
#  followed by arglist.  Note exit status!
function usage
{
  prog="$1"; shift
  print -u2 "$prog: usage: $prog $@"
  exit  1
}
```

Ending Functions

A function completes after executing the last command:

- The exit code is normally that of the last command
- **`return`** can be used to specify an exit code **`N`**, or just end the function at that point

```
return N
```

- **`exit`** will terminate the current function and current shell

```
exit N
```

- Errors within a Korn shell function cause it to return control and the error exit code to the calling Script

Functions may be deleted from memory using...

```
unset -f functionname
```

Functions and Traps

The behavior of **`trap`** with functions is determined by the shell type:

Bourne:	a **`trap`** is "global" – the same in and out of a **`function`**
Korn:88	a **`trap`** is "local" to a **`function`** and is reset on completion
	a main program **`trap`** is shared with **`function`**s, but can be overriden inside **`function`**
	a **`signal`** that is not caught or ignored, may cause the script to terminate
	a **`signal`** that is ignored by the shell, is also ignored by **`function`**s called from it

Functions in ksh93

- Function's characteristics change in ksh93 depending on which syntax was used to set up the **`function`**

`identifier ( )`
`{ ...`
`}`

- All variables are global

- **`$0`** always scriptname

- A main program trap is shared with **`functions`**
- A **`trap`** inside a **`function`** overrides a main program **`trap`**, and is passed out

`function identifier`
`{ ...`
`}`

- Variables are made local with **`"typeset"`**
- **`$0`** reflects **`function`** name while inside the **`function`**
- A main program **`trap`** is shared with **`functions`**
- A **`trap`** inside a **`function`** overrides a main program **`trap`**, but only while inside the **`function`**

Functions in bash

- **`$0`** will always be the scriptname, whether inside or outside **`function`**
- Prefers "**`declare`**" or "**`local`**" over **`typeset`**
- A main program **`trap`** is shared with **`function`**
- A **`trap`** within a **`function`** overrides the main program **`trap`** while inside the **`function`**, and is passed out to the main program

The typeset/declare Commands

The Korn shell **typeset** and Bash shell **declare** commands define or list variables and their attributes:

```
typeset ±LN variable1=value1 variable2=value2 ...
```

Omitting variables lists variables with specified attributes

- sets attributes, or lists names and values

+ unsets attributes, or lists just names

Where **L** is any of ...

r	the **readonly** attribute – no modification of variables' value
i	sets the **integer** attribute – use with **N** to set number *base*
x	the **export** attribute – the variable will be exported

The preferred method in bash is the "**declare**" command

typeset Examples

Declare arrays to specify size and/or attributes:

```
$ typeset -xi8 a2[1]
$ a2=52
$ a2[1]=25
$ ksh
$ print $a2 ${a2[1]}
8#64
$ _
```

exported & octal integer

only element 0 was exported

Inside a Korn Shell function, **typeset** creates a "local" variable...

```
# Function to convert numbers into binary
function  binary_convert
{
      typeset -i2 binary=$1
      print "$1 = $binary"
}
```

typeset with Functions

Other uses of **typeset** are:

- Display functions
- Set function attributes
- Unset function attributes

```
typeset ±fL function1 function2 ...
```

- To list functions with specified attributes, omit function list
- **-f** sets attributes, or displays function names and definitions
- **+f** unsets attributes, or displays only function names

Where **L** is any of...

x the **export** attribute – the function will be available to implicit shells invoked from the current one

t the shell **xtrace** option for a function

typeset with Functions Examples

```
$ typeset -f
function list
{
      while [[ "$1" != "X" ]]
      do
                  print $1
                  shift  1
      done
}
$ typeset -fx list
$ typeset +f  or typeset -F
list
$ unset -f list
$ typeset -f
$ _
```

shows functions in full

export the list function

(bash) lists function names

list doesn't exist anymore

autoload Functions

A shell function that is defined only when it is first called, is an **autoload** function:

- Using **autoload** functions improves performance
- The shell searches directories listed in the **FPATH** variable for a file with the name of the called function
- Call the **autoload** from within your .profile (or .bash_profile)
- The contents of that file then defines the function
- Existing function definitions are not unset

Another way is to:

- Place functions into a separate directory
- Set **$FPATH** equal to the full pathname of that directory
- Make sure the function name and file name is the same

Aliases

The Korn shell **`alias`** facility provides:

- A way of creating new commands
- A means of renaming existing commands

Creation: **`alias name=definition`**

Deletion: **`unalias name`**

An **`alias`** definition may contain any valid shell script or metacharacters

Processing Aliases

Command lines are split into words by the shell:

- Check the first word of each command line for a defined **`alias`**
- A backslash in front of a command name prevents **`alias`** expansion if the **`alias`** exists
- If the definition ends in a **`space`** or **`tab`**, the next command word will also be processed for **`alias`** expansion
- Resolve **`alias`** names within a **`function`** when function definitions are read, not at execution

Preset Aliases

- Korn shell uses the following exported aliases
  - May be unaliased or redefined

```
alias functions='typeset -f'
alias hash='alias -t'
alias history='fc -l'
alias integer='typeset -i'
alias nohup='nohup '
alias r='fc -e -'
alias stop='kill -STOP'
alias suspend='kill -STOP $$'
alias type='whence -v'
```

with trailing space

The alias Command

The **alias** command has some options:

```
alias -L name=definition
```

Where **L** is any mix of...

x	to set, or display exported aliases
t	to set, or list tracked aliases

If ***definition*** is quoted...

"definition**"**	doubles are interpreted when entered
'definition**'**	singles are interpreted when executed

alias Examples

```
$ x=10
$ alias px="print $x" rx='print $x'
$ x=100
$ px                              prints $x as it was
10
$ rx                              prints the current $x
100

$ alias -x ls='ls -a'             ls is set and exported

$ rm /tmp/*                       you want to remove all /tmp
rm: remove '/tmp/atestfile'? _
<Ctrl-c>                          you realize the list is too long
$ \rm /tmp/*                      you escape the alias for rm
...                               you cross your fingers
$ ls /tmp                         you hope you did it correctly
```

Tracked Aliases

A **tracked alias** reduces the search time for a future use of a command

```
set -o trackall    or    set -h
```

Turns on Shell **trackall** option

First use of a command creates **tracked alias**

Force creation with

```
alias -t name
```

List all **tracked alias**es

```
alias -t
```

NOTE: The value of a **tracked alias** becomes undefined when the **PATH** variable is reset

Hashing in bash

A **hash** reduces the search time for a future use of a command.

All commands are remembered in a **hash** table by bash. Disable this facility by:

set -d or **set -o nohash**

The built-in **hash** lists the table

Add an explicit entry by

hash command (must be in PATH)

To delete the **hash** table:

hash -r

The whence Command

whence reports how a command will be carried out by the shell

```
whence -pv command
```

- **-v** for a verbose report
- **-p** to force a PATH search even if the command is an **alias** or **function** (AIX only option)

```
$ whence vi
/usr/bin/vi
$ whence -v vi                                  executable program
vi is a tracked alias for /usr/bin/vi
$ whence -v print
print is a shell builtin
$ whence type                                   so type is an alias
whence -v
$ type for
for is a reserved word
$ _
```

* when in bash, use **type** instead of **whence** (**type** is built-in in bash)

The eval Command

The shell processes each command line read before invoking the relevant commands.

- If you want to reread and process a command line, use **`eval`**:
- **`eval`** processes its arguments as normal
- The arguments are formed into a space separated string
- The shell then executes that string as a command line
- The return value is that of the executed command line

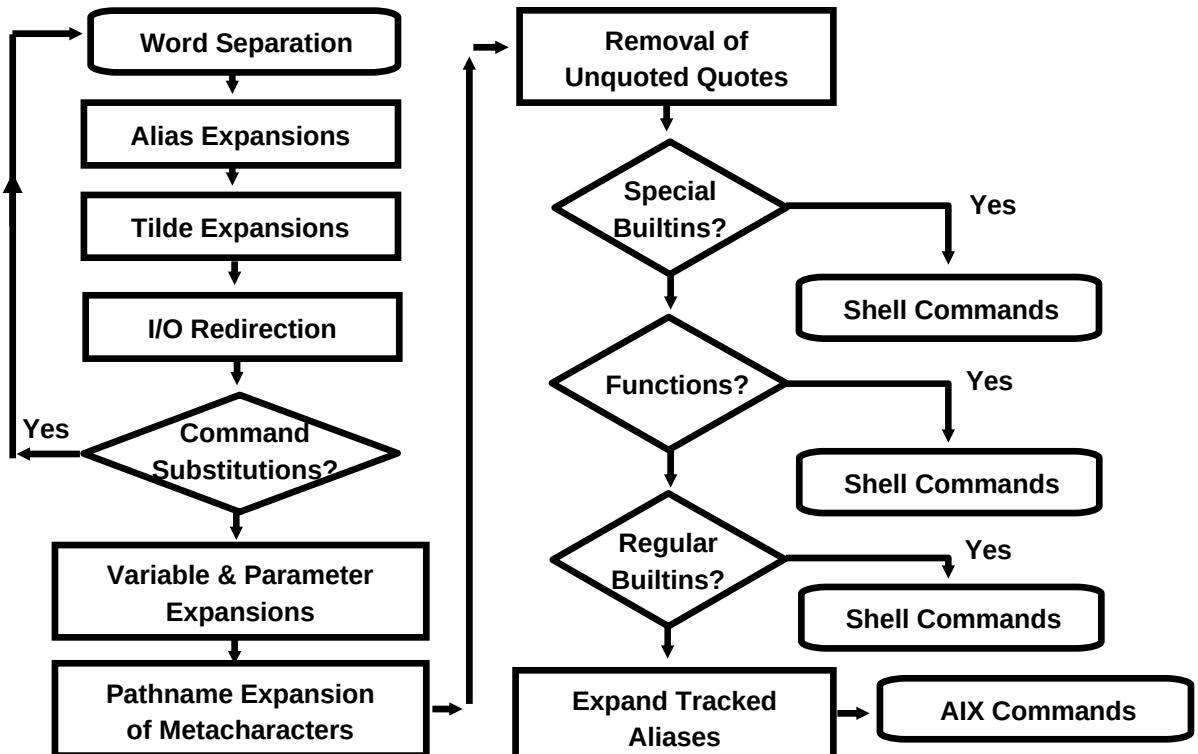
eval Examples

Here are some eval command lines...

```
$ eval print '*sh'
getopts.example.ksh      eval.ksh try.sh

$ message10=Hello

$ variable=message10                    print the message
$ eval print '$'$variable               named by $variable
Hello

$ cmd='ps -ef | grep marty'             run a string command
$ eval $cmd                             to list marty's processes
...
$ _
```

Command Line Processing

Each command line is processed in the following way by the shell:



Checkpoint

1. How is an array defined?
2. How do we refer to array elements?
3. How could we set a variable **users**, to contain the number of users logged onto the system?
4. How would we write a function to check the readability of a file?
5. How do we print out the first and last positional parameter?
6. How do we define local variables within a function?
7. How can we list which functions are defined?
8. Which command would allow you to load a library of functions?
9. How could we create an alias to show how many minutes have elapsed since the current shell began?

Unit Summary

- Shell arrays – defining and referencing
- Functions
- **`typeset`** command
- **`autoload`** functions
- Command aliases
- Preset aliases
- Tracked aliases
- The **`whence`** command
- Command line processing
- Command line re-evaluation with **`eval`**