Unit 5
Shell Commands

Unit Objectives

After completing this unit, you should be able to:

- Use the **`print`** and **`echo`** command
- Use special printing characters
- Use the **`read`** command
- Understand option and argument processing with **`getopts`**
- Use history manipulations with **`fc`**
- Use the **`set`** command
- Use shell options with **`set`**
- Use shell invocation
- Use built-in commands
- Use shell commands provided by AIX

The print Command (ksh 88 and ksh93)

The **print** command is the Korn shell output mechanism:

`print argument ...`	prints arguments to standard output separated by spaces
`print - argument ...`	to print arguments that look like options
`print -r argument ...`	RAW mode – do not interpret print's special characters (listed on next page)
`print -R argument ...`	equivalent to "**-**" and "**-r**"
`print -uN argument ...`	output sent to file descriptor **N**
`print -s argument ...`	output to the shell history file only

Special print Characters

Backslash character sequences have special meaning (except in raw mode)

`\a`	Alarm - ring the terminal bell
`\b`	Backspace
`\c`	Print without trailing newline (same as `print -n`)
`\f`	Form feed
`\n`	Newline
`\r`	Return
`\t`	Tab
`\v`	Vertical tab
`\\`	Backslash
`\0xxx`	Character with octal code `xxx` (up to three octal digits)

The echo Command (Bash)

The **echo** command is the Bash shell output mechanism:

- The **echo** special characters in bash are the same as the **print** special characters in ksh (**\a**, **\b**, **\c**, and so forth)
- To use the **echo** special characters, you must use the **-e** option
- On some systems, **-e** is the default. In this case **-E** turns off the interpretation of special characters (similar to **-r** in **print**)

echo also has **-n** option to omit trailing newline after input

print Examples

When you use the **print** command, here's what you get.

```
$ print "Line 1\n\tLine2"
Line1
      Line 2
$ print 'One quarter = \0274'
One quarter = ¼
$ print 'Backslash = \0134'
Backslash = \
$ print -r 'hi\\\\there 1'
hi\\\\there 1
$ print -r hi\\\\there 2
hi\\there 2
$ print 'hi\\\\there 3'
hi\\there 3
$ print hi\\\\there 4
hi\there 4
$ _
```

with -r and quotes

with -r and no quotes

with no -r and quotes

with no -r and no quotes

The printf Command - An Advanced Print

- The **printf** command allows for more powerful formatting.

- The **printf** commands comes built-in with ksh93. However, AIX also has a version of **printf** (/usr/bin/printf) that can be used from bash and ksh88.

- Syntax:
  - **printf format-string [arguments ...]**

- Examples:

```
printf "#%10s#\n" title
printf "#%-10s#\n" title
printf "%.5f" 123456.789
```

Results:

```
#     Title#
#Title     #
123456.78900
```

The read Command

To get input while a shell script is running, use **read**:

```
read variable ...
```

The **read** command reads a line from its standard input

- Assigns input words to the variables
- Set remaining variables to null if too few words
- Set last variable to the remainder of the words if too few variables

For the Korn and bash shells, if no variables are specified, the **REPLY** variable is set to the whole input line

read Examples

We can use the **read** from the shell prompt as well:

```
$ read var1 var2
123 456 789
$ print "var1 = $var1 \tvar2 = $var2"
var1 = 123          var2 = 456 789
$ read var1 var2
abc
$ print "var1 = $var1 \tvar2 = $var2"
var1 = abc var2 =
$ read
hi there
$ print $REPLY
hi there
$ _
```

read Command Options

The Korn shell **read** command has some options:

```
read -r variable ...      raw mode – \ is not taken as
                          a line continuation character

read -uN variable ...     read from file descriptor N
```

You can specify a prompt for the command to display on standard error, add a "?prompt" to the first variable

```
read  variable?prompt  variable ...
```

For example, to request a user for a text string:

```
read string?'Please enter a text string'
```

read Options for ksh93

`read -A variable`	reads words into an indexed array named *variable*, starting at index 0
`read -d delimiter`	use *"delimiter"* instead of newline
`read -n number`	read, at most, *"number"* bytes
`read -t seconds`	wait *"seconds"* for input, else exit

read Options for bash

`read -a variable`	reads words into an indexed array named *variable*, starting at index 0
`read -p prompt variable`	similar to read *var?prompt* in ksh
`read -s`	silient mode (no echo)

read Options Examples (1 of 2)

```
#!/usr/bin/ksh
# usage: readrun
# prompt the user for their name
read first?"Enter your name: " last
print "firstname = $first\tlastname=$last"
```

What would the result be for the following?

```
# readrun
Enter your name: Lee
  firstname = _____     lastname = _____

  Enter your name: Lee Lynn Llewellyn
  firstname = _____     lastname = _____
```

read Options Examples (2 of 2)

```
#!/usr/bin/ksh
# Usage: readpwd
# Read & print parts of /etc/passwd.
IFS=:
while read name pwd uid guid gecos home shell
do
        print "$name" "$uid" "$guid" "$shell"
done  <  /etc/passwd
```

Here's what happens:

```
$ readpwd
root 0 0 /bin/bash
bin 1 1 /sbin/nologin
daemon 2 2 /sbin/nologin
adm 3 4 /sbin/nologin
...
$ _
```

Processing Options

Parameters on a script command line are of two types:

- Arguments – used in script
- Options – used to tell the script what to process

General parameter/argument processing is difficult

Consider

```
$ myscript -a -f optionfile argfile
$ myscript -foptionfile -va argfile
```

Shell provides **`getopts`** as a solution

The getopts Command

- The **getopts** command processes options and associated arguments from a parameter list

```
getopts optionstring variable parameter...
```

- Each invocation of **getopts** processes the next option in the **parameter** list (parameter list usually comes from the command line, but can come from within a script)
  - Usually called within a loop
- The **optionstring** lists expected option identifiers
  - If an option identifier requires an associated argument, add a colon (**:**)
- A leading colon in the list suppresses "invalid option" messages by **getopts**

getopts Syntax Example

How are options processed when passed to a script?
Assumptions:

- The possible options are **a**, **b** and **c**
- Option **b** is to have an associated argument
- Suppress normal OpSys error messages

Inside the script **getopts** will be used early on:

```
while  getopts  ':ab:c'  flag
do
        identify the values set by getopts
done
```

A correct command line to the script might be

```
$ prog.ksh  +c  -ab  barg  --  arg1  arg2
```

What about?

```
$ prog.ksh   -c -b -a -- arg1 arg2
```

getopts Example

```
#!/usr/bin/ksh
#  Example of getopts
USAGE="usage:  example.getopts.ksh [+-c] [+-v] [-a argument]"

while getopts :a:cv varflag
do
case $varflag in
     a)    argument=$OPTARG ;;
     c)    compile=on ;;
    +c)    compile=off ;;
     v)    verbose=on ;;
    +v)    verbose=off ;;
     :)    print "You forgot an argument for the switch called a.";
           exit ;;
    \?)    print "$OPTARG is not a valid switch" ; print "$USAGE" ;
           exit ;;
    esac
done
print -c "compile is $compile; verbose is $verbose;
print "argument is $argument "
#END
```

getopts Notes

- **`getopts`** does not support options that start with a "**`+`**" in bash

- **`getopts`** supports putting a "**`#`**" after an option letter (in the valid option list) instead of a "**`:`**" to specify the option's argument must be a number in ksh93
  - Example:
    `:ab#c` **`b`** takes an argument, which must be a number

The fc Command

The Shell fc command interactively edits and then re-executes portions of your command history file:

`fc start end`	edits and executes a command range -- **`start`** defaults to the last command -- **`end`** defaults to the value of start
`-e editor`	to specify an editor other than **`$FCEDIT`** -- WARNING! The shell default is **`/bin/ed`**

To re-execute a single command with automatic editing:

```
fc -e - old=new command
```

`old=new`	to swap string **`old`** with string **`new`**
`command`	to specify a **`command`** -- default command is the last command

fc Examples - Edit and Execute, List

Ranges may be strings, absolute or relative numbers...

`$ fc`	edit the last command with the `$FCEDIT` editor, and then re-execute
`$ fc cc`	edit the previous command beginning with `cc`
`$ fc -e vi 10 20`	use `vi` to edit history lines `10` to `20`

Automatic editing can specify a command in a similar way

`$ fc -e -`	re-execute last command as it was
`$ fc -e - 2=3 10`	swap `3` for `2` in command number `10`

The ksh fc command lists portions of your command history file:

`$ fc -l start end`	list the specified command range -- the default is the last 16 commands

For example...

`$ fc -l pg grep`	lists commands from the last `pg` to a `grep`
`$ fc -l 15 20`	lists commands `15` to `20`
`$ fc -l -5 -1`	lists the last `5` commands

The set Command

We have seen three functions performed by the **set** command:

set	lists set variables with their values
set value ...	resets the positional parameters
set -o vi	enables line recall and editing

This last form sets a shell option. There are several more options to **set**:

- Shell options and settings are listed by **set –o**
- Turn option on using **set -o option** or **set -L**
  (where **L** is an option identifier)
- Turn option off using **set +o option** or **set +L**

Korn Shell Options with Set (1 of 2)

Option:	*L*	*Description:*
allexport	**a**	automatically export each variable set
bgnice		run all background jobs at a lower priority – this is on by default for interactive shells
ignoreeof		stops an interactive shell exiting on **<Ctrl-d>** – you must use the **exit** command
errexit	**e**	exits if any command returns a non-zero return code
noclobber	**C**	stops the shell overwriting existing files with > redirection (**>|** works instead)
noexec	**n**	for a non-interactive shell to check syntax without executing commands

Korn Shell Options with Set (2 of 2)

Option	*L*	*Description*
noglob	**f**	to disable metacharacter expansion
notify	**b**	to notify asynchronously of background job completions
nounset	**u**	displays an error message when an unset variable is used
	s	to sort positional parameters -- ksh only
trackall	**h**	set-up a tracked alias for each new command —- on for non-interactive shells
verbose	**v**	to display input on standard error as it is read
vi		turns on history line recall and **vi** editing
xtrace	**x**	the debug option – the shell displays PS4 with each processed command line

Additional ksh93 Shell Options

`set -o pipefail`

Usually the exit status is of the last command in a pipeline.

`set -o pipefail` changes this behavior.

The exit status of a pipeline is changed to that of the last command to fail

`set -o viraw`

Allows for **`set -o vi`** plus allows **`<Tab>`** for file name completion

Bash Shell Options with Set

- The bash shell options are the same as the Korn shell unless noted:
  - The **`set -h`** (**`set -o hashall`**) disables hashing of commands
  - There is no **`set -o bgnice`** or **`set -o trackall`**
  - Bash users traditionally use **`set -o emacs`**

- Use "**`set -o`**" to list all of bash's options

Set Quiz

1. What command would you use to re-set the positional parameters to "one" "two" "three"?

2. What lists the shell options with settings?

3. Which **`set`** option ensures that each variable assignment will be inherited by a subshell?

4. What would stop **`<Ctrl-d>`** from logging me out?

5. How can I use **`set`** to protect my files from being overwritten by output redirection?

Shell Built-in Commands

We have seen the following built-in shell commands:

.	:	bg	break
cd	continue	echo	eval
exec	exit	export	fc
fg	getopts	jobs	kill
print	pwd	read	readonly
set	shift	test	[ ]
trap	typeset	unset	wait

In the later units we will see:

alias	command	let or (( ))	return
times	ulimit	unalias	whence

All built-in commands can run in the current environment

Special built-in commands may terminate the shell if an error occurs

AIX Shell Commands

Some built-in Korn shell commands are also provided as AIX commands, accessible from all shells:

`alias`	`bg`	`cd`	`command`
`echo`	`fc`	`fg`	`getopt`
`jobs`	`kill`	`newgrp`	`read`
`umask`	`unalias`	`wait`	

AIX commands are also provided for the logical words:

`false` `true`

Most of these commands are shell scripts in /usr/bin – they are provided for POSIX compliance

Checkpoint

1. Without using redirection, how could we print information to file descriptor 2?
2. What is wrong with the following command?
   ```
   read speed?"mph" distance?"miles"
   ```
3. What **`getopts`** statement would allow you to process options **`p`**, and **`a`**, with option **`t`** expecting an associated value?
4. In the bash shell, **`print`** is not built-in. What is the built-in command in Bash that performs similarly to Korn's **`print`**?
5. Which **`set`** option disables metacharacter pathname expansion?
6. Which **`set`** options would be most useful in helping to debug a shell script?

Unit Summary

- The Korn shell **`print`** command
- The Bash shell **`echo`** command
- Special printing characters
- The **`read`** command
- Option and argument processing with **`getopts`**
- History manipulations with **`fc`**
- The **`set`** command
- Shell options with **`set`**
- Shell invocation
- Built-in commands
- Shell commands provided by AIX