# Shell Programming for System Administrators

SA-245

# About This Course

# Course Goal

This course provides you with knowledge and skills to:

- Develop shell scripts
- Analyze and design procedures to successfully create shell scripts
- Automate the tasks that you would frequently perform on the job

# Course Map

## Shell Basics and Script Analysis and Design

| | | |
|---|---|---|
| UNIX® Shells and Shell Scripts | Writing and Debugging Scripts | The Shell Environment |

## Useful Utilities and Regular Expression Characters

| | | |
|---|---|---|
| Regular Expressions and `grep` | The sed Editor | The `nawk` Programming Language |

## Programming Constructs

| | | |
|---|---|---|
| Conditionals | Interactive Scripts | Loops |

## Advanced Shell Programming

| | | |
|---|---|---|
| Advanced Variables, Parameters, and Argument Lists | Functions | Traps |

# Course Overview

- Description
- Audience
- The Bourne and Korn shells

# Module-by-Module Overview

- Module 1 – "UNIX Shells and Shell Scripts"
- Module 2 – "Writing and Debugging Scripts"
- Module 3 – "The Shell Environment"
- Module 4 – "Regular Expressions and `grep`"
- Module 5 – "The `sed` Editor"
- Module 6 – "The `nawk` Programming Language"
- Module 7 – "Conditionals"
- Module 8 – "Interactive Scripts"
- Module 9 – "Loops"
- Module 10 – "Advanced Variables, Parameters, and Argument Lists"
- Module 11 – "Functions"
- Module 12 – "Traps"

# Course Objectives

- Develop and debug scripts
- Use local and environmental variables and shell metacharacters in scripts
- Customize system-wide shell initialization files
- Use regular expression characters with the `grep`, `sed`, and `nawk` utilities
- Write `sed` scripts to perform non-interactive editing tasks
- Write `nawk` scripts to manipulate individual fields within a record
- Write `nawk` scripts to write reports based upon an input file

# Course Objectives

- Use the exit status of a command to determine if the command succeeded or failed
- Access and process command-line arguments passed into a script
- Develop a USAGE message to display when a script is invoked incorrectly
- Use flow control constructs, such as branching and looping
- Develop interactive scripts
- Perform string manipulation and integer arithmetic on shell variables
- Write a script that uses functions
- Write a script that uses a trap to catch a signal

# Skills Gained by Module

## Meaning of:

- Black boxes
- Gray boxes.

| Skills Gained | Module 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Develop and debug scripts | ■ | ■ | ■ | | | | | | | | | |
| Use local and environmental variables and shell metacharacters in scripts | | | ■ | | | | | | | | | |
| Customize system-wide shell initialization files | | | ■ | | | | | | | | | |
| Use regular expression characters with the `grep`, `sed`, and `nawk` utilities | | | | ■ | ■ | ■ | | | | | | |
| Write `sed` scripts to perform non-interactive editing tasks | | | | | ■ | | | | | | | |
| Write `nawk` scripts to manipulate individual fields within a record | | | | | | ■ | | | | | | |
| Write `nawk` scripts to write reports based upon an input file | | | | | | ■ | | | | | | |
| Use the exit status of a command to determine if the command succeeded or failed | | | | | | | ■ | | | | | |
| Access and process command-line arguments passed into a script | | | | | | | | | | ■ | | |
| Develop a USAGE message to display when a script is invoked incorrectly | | | | | | | ■ | | | | | |
| Use flow control constructs, such as branching and looping | | | | | | | ■ | | ■ | | | |

# Skills Gained by Module

•

| Skills Gained | Module 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Develop interactive scripts | | | | | | | | ■ | | | | |
| Perform string manipulation and integer arithmetic on shell variables | | | | | | | | | | ■ | | |
| Write a script that uses functions | | | | | | | | | | | ■ | |
| Write a script that uses a trap to catch a signal | | | | | | | | | | | | ■ |

# Topics Not Covered

- Basic UNIX® commands – Covered in SA-118: *Fundamentals of Solaris™ 8 Operating Environment*
- System startup and shutdown procedures – Covered in SA-238: *Solaris™ 8 Operating Environment System Administration I*
- How to add a user to the system – Covered in SA-238: *Solaris™ 8 Operating Environment System Administration I*
- Logical device names for disks – Covered in SA-238: *Solaris™ 8 Operating Environment System Administration I*
- Mounting and unmounting file systems – Covered in SA-238: *Solaris™ 8 Operating Environment System Administration I*

# Topics Not Covered

- Adding and removing software on the system – Covered in SA-238: *Solaris™ 8 Operating Environment System Administration I*
- The JumpStart™ facility – Covered in SA-238: S*olaris™ 8 Operating Environment System Administration I* and SA-381: *Solaris™ JumpStart™*

# How Prepared Are You?

- Are you an experienced UNIX user who is familiar with basic commands, such as `rm`, `cp`, `man`, `more`, `mkdir`, `ps`, and `chmod`?
- Can you create and edit text files using `vi` or a text editor?
- Do you understand the system boot process and proper shutdown procedures?
- Can you create users and passwords and set file permissions?
- Do you understand device naming conventions to mount and unmount file systems?
- Do you know user software package commands, such as `pkgadd`, `pkgrm`, and `pkginfo`?

# Introductions

- Name
- Company affiliation
- Title, function, and job responsibility
- System administration experience
- Experience modifying or writing scripts
- Reasons for enrolling in this course
- Expectations for this course

# How to Use Course Materials

- Course map
- Objectives
- Lecture
- Exercise
- Check Your Progress

# Typographical Conventions and Symbols

- `Courier` is used for the names of commands, files, and directories, as well as on-screen computer output.
- **`Courier bold`** is used for characters and numbers that you type.
- *`Courier italics`* is used for variables and command-line placeholders that are replaced with a real name or value.
- *Palatino italics* is used for book titles, new words or terms, or words that are emphasized.

# Module 1

## UNIX® Shells and Shell Scripts

# Objectives

- Describe the role of shells in the UNIX environment
- Describe the standard shells
- Define the components of a shell script
- Write a simple shell script

# What Is a Shell?

- A command-line interpreter
- A utility program
- A program started for each user when they log in or open a command or tool window
- An interface between the user and the operating system (kernel)
- A way for the user to execute utilities and other programs

# What Are a Shell's Functions?

- Command-line interpreter
- Programming language
- User environment

# Available Shells

- Bourne shell (`sh`)
- C shell (`csh`)
- Korn shell (`ksh`)
- GNU Bourne-Again shell (`bash`)
- Desktop Korn shell (`dtksh`)
- Job Control shell (`jsh`)
- Restricted Shell Command Interpreter (`rsh`)
- Enhanced C shell (`tcsh`)
- Z shell (`zsh`)

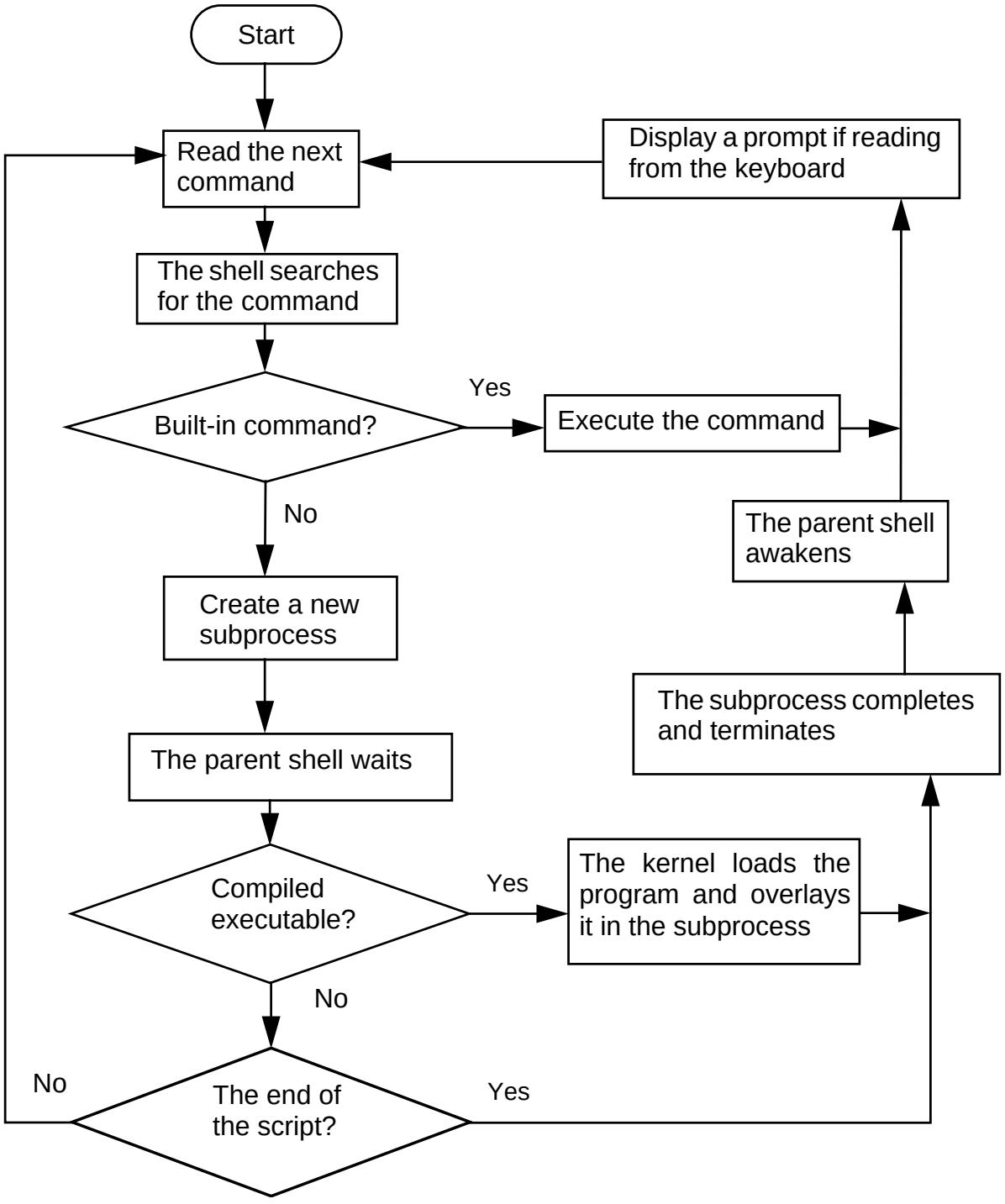
# Subshells – Child Processes

- Compiled executable script
  - Creates a child process
  - The parent waits
  - The child executes
- Shell script
  - Invoke with a script name
  - The child process is a shell
  - The parent waits
  - The child executes

# Subshells – Child Processes

- Invoke with dot command
  - The script runs in current shell

# Subshells



# What Is a Shell Script?

- Contains shell and UNIX commands
- Specific purpose
- Reusable
- Executed like any command

# Programming Terminology

- Logic flow
- Loop
- User input
- Conditional branch
- Command control

# Logic-Flow Design

1. Do you want to add a user?
   a. If Yes:
      1. Enter the user's name.
      2. Choose a shell for the user.
      3. Determine the user's home directory.
      4. Determine the group to which the user belongs.
   b. If No, go to Step 3.
2. Do you want to add another user?
   a. If Yes, go to Step 1.a.
   b. If No, go to Step 3.
3. Exit.

# Example Bourne Script: `echoscript1.sh`

```
# cat echoscript1.sh
#!/bin/sh

clear
echo "SCRIPT BEGINS"

echo "Hello $LOGNAME"
echo

echo "Todays date is: \c"
date '+%m/%d/%y'

echo "and the current time is: \c"
date '+%H:%M:%S%n'

echo "Now a list of the processes in the current shell"
ps

echo "SCRIPT FINISHED!!"
```

# Example Korn Script: `echoscript2.ksh`

```
$ cat echoscript2.ksh
#!/bin/ksh

clear
print "SCRIPT BEGINS"

print "Hello $LOGNAME"
print

print -n "Todays date is: "
date '+%m/%d/%y'

print -n "and the current time is: "
date '+%H:%M:%S%n'

print "Now a list of the processes in the current shell"
ps

print "SCRIPT FINISHED!!"
```

# Example Boot Script: `/etc/init.d/volmgt`

```
$ cat /etc/init.d/volmgt
#!/sbin/sh
#
# Copyright (c) 1997-1998 by Sun Microsystems, Inc.
# All rights reserved.
#
#ident  "@(#)volmgt     1.6     98/12/14 SMI"

case "$1" in
'start')
        if [ -f /etc/vold.conf -a -f /usr/sbin/vold ]; then
                echo 'volume management starting.'
                /usr/sbin/vold >/dev/msglog 2>&1 &
        fi
        ;;

'stop')
        /usr/bin/pkill -x -u 0 vold
        ;;

*)
        echo "Usage: $0 { start | stop }"
        exit 1
        ;;

esac
exit 0
```

# Exercise: UNIX Shells and Shell Scripts

- Objectives
- Tasks
- Discussion
- Solutions

# Check Your Progress

- Describe the role of shells in the UNIX environment
- Describe the standard shells
- Define the components of a shell script
- Write a simple shell script

# Module 2

## Writing Scripts

# Objectives

- Start a script with #!
- Put comments in a script
- Put commands in a script
- Change permissions on the script
- Execute a script
- Debug a script

# Creating Shell Scripts

- Create a file using any text editor
- Put UNIX commands, user programs, or the names of other scripts as commands in the file
- Save the file, and exit from the editor

# Creating Shell Scripts

```
$ vi firstscript.sh

<vi session>

$ cat firstscript.sh
#!/bin/sh

clear
echo "SCRIPT BEGINS"

echo "Hello $LOGNAME!"
echo

echo "Todays date and time: \c"
date
echo

mynum=21
myday="Monday"

echo "The value of mynum is $mynum"
echo "The value of myday is $myday"
echo

echo "SCRIPT FINISHED!!"
echo
```

# Executing a Shell Script

- Give the script execute permission
- Execute the script as a command
- Create a subshell in which to execute the script

# Executing `firstscript.sh`

```
$ cat firstscript.sh
#!/bin/sh

clear
echo "SCRIPT BEGINS"

echo "Hello $USER!"
echo

echo "Todays date and time: \c"
date
echo

mynum=21
myday="Monday"

echo "The value of mynum is $mynum"
echo "The value of myday is $myday"
echo

echo "SCRIPT FINISHED!!"
echo
```

# Starting a Script With #!

- For a system boot script, use Bourne shell (`/sbin/sh`)
- For a typical script use the shell that is most comfortable to you or the one that supports all the features you desire

```
#!/bin/sh
#!/bin/csh
#!/bin/ksh
```

# Putting Comments in a Script

```
$ cat scriptwithcomments.sh
#!/bin/sh

# This script clears the window, greets the user,
# and displays the current date and time.

clear                                   # Clear the window
echo "SCRIPT BEGINS"

echo "Hello $LOGNAME!"                 # Greet the user
echo

echo "Todays date and time: \c"
date                                    # Display current date and
time
echo

mynum=21                                # Set a local shell variable
myday="Monday"                          # Set a local shell variable

echo "The value of mynum is $mynum"
echo "The value of myday is $myday"
echo

echo "SCRIPT FINISHED!!"
echo
```

```
$ ./scriptwithcomments
SCRIPT BEGINS
Hello root!

Todays date and time: Fri May  5 13:44:48 MDT 2000

The value of mynum is 21
The value of myday is Monday

SCRIPT FINISHED!!
```

# Adding the Debugging Statement

- To run an entire script in debug mode, add `-x` after the `#!/bin/ksh` on the first line:

  ```
  #!/bin/ksh   -x
  ```

- To run an entire script in debug mode from the command line, add a `-x` to the `ksh` command used to execute the script:

  ```
  $ ksh   -x   script_name
  ```

- To run debug with options, use `-x`, `-v`, or `-f`

# Debug Mode Controls

- `-x` Displays the line after interpreting metacharacters and variables
- `-v` Displays the line before interpreting metacharacters and variables
- `-f` Disables file-name substitutions
- `set -`*`option`* Turns on the option
- `set +`*`option`* Turns off the option

# Example: Debug Mode Specified on the #! Line

```
$ cat debug1.sh
#!/bin/sh -x

echo "Your terminal type is set to: $TERM"
echo

echo "Your Timezone is set to: $TZ"
echo

echo "Now we will list all the nfs scripts in /etc/rc2.d"
ls /etc/rc2.d/*nfs*
echo

echo "Now we will list all the nfs scripts in /etc/rc3.d"
ls /etc/rc3.d/*nfs*
echo
```

# Results of debug1 With the -x Option

```
$ ./debug1.sh
+ echo Your terminal type is set to: dtterm
Your terminal type is set to: dtterm
+ echo

+ echo Your Timezone is set to: US/Mountain
Your Timezone is set to: US/Mountain
+ echo

+ echo Now we will list all the nfs scripts in /etc/rc2.d
Now we will list all the nfs scripts in /etc/rc2.d
+ ls /etc/rc2.d/K28nfs.server /etc/rc2.d/S73nfs.client
/etc/rc2.d/K28nfs.server  /etc/rc2.d/S73nfs.client
+ echo

+ echo Now we will list all the nfs scripts in /etc/rc3.d
Now we will list all the nfs scripts in /etc/rc3.d
+ ls /etc/rc3.d/S15nfs.server
/etc/rc3.d/S15nfs.server
+ echo
```

# Example: Debug Mode With `set -x`

```
$ cat debug2.sh
#!/bin/sh

set -x
echo "Your terminal type is set to: $TERM"
echo
set +x

echo "Your Timezone is set to: $TZ"
echo

echo "Now we will list all the nfs scripts in /etc/rc2.d"
ls /etc/rc2.d/*nfs*
echo

echo "Now we will list all the nfs scripts in /etc/rc3.d"
ls /etc/rc3.d/*nfs*
echo
Note the following lines in the script:
echo "Your terminal type is set to: $TERM"
echo

$ ./debug2.sh
+ echo Your terminal type is set to: dtterm
Your terminal type is set to: dtterm
+ echo

Your Timezone is set to: US/Mountain

Now we will list all the nfs scripts in /etc/rc2.d
/etc/rc2.d/K28nfs.server  /etc/rc2.d/S73nfs.client

Now we will list all the nfs scripts in /etc/rc3.d
/etc/rc3.d/S15nfs.server
```

# Example: Debug Mode With `set -v`

```
$ cat debug3.ksh
#!/bin/ksh
set -v

echo "Your terminal type is set to: $TERM"
echo

echo "Your Timezone is set to: $TZ"
echo

echo "Now we will list all the nfs scripts in /etc/rc2.d"
ls /etc/rc2.d/*nfs*
echo

echo "Now we will list all the nfs scripts in /etc/rc3.d"
ls /etc/rc3.d/*nfs*
echo
```

# Results of debug3.ksh

```
$ ./debug3.ksh
echo "Your terminal type is set to: $TERM"
Your terminal type is set to: dtterm
echo


echo "Your Timezone is set to: $TZ"
Your Timezone is set to: US/Mountain
echo


echo "Now we will list all the nfs scripts in /etc/rc2.d"
Now we will list all the nfs scripts in /etc/rc2.d
ls /etc/rc2.d/*nfs*
/etc/rc2.d/K28nfs.server  /etc/rc2.d/S73nfs.client
echo


echo "Now we will list all the nfs scripts in /etc/rc3.d"
Now we will list all the nfs scripts in /etc/rc3.d
ls /etc/rc3.d/*nfs*
/etc/rc3.d/S15nfs.server
echo
```

# Example: Debug Mode With `set -o noglob`

```
$ cat debug4.ksh

#!/bin/ksh

echo "Your terminal type is set to: $TERM"
echo

echo "Your Timezone is set to: $TZ"
echo

set -o noglob
echo "Now we will list all the nfs scripts in /etc/rc2.d"
ls /etc/rc2.d/*nfs*
echo
set +o noglob

echo "Now we will list all the nfs scripts in /etc/rc3.d"
ls /etc/rc3.d/*nfs*
echo

$./debug4.ksh
Your terminal type is set to: dtterm

Your Timezone is set to: US/Mountain

Now we will list all the nfs scripts in /etc/rc2.d
/etc/rc2.d/*nfs*: No such file or directory

Now we will list all the nfs scripts in /etc/rc3.d
/etc/rc3.d/S15nfs.server
$
```

# Exercise: Writing Shell Scripts

- Objectives
- Tasks
- Discussion
- Solutions

# Check Your Progress

- Start a script with #!
- Put comments in a script
- Put commands in a script
- Change permissions on the script
- Execute a script
- Debug a script

# Module 3

## The Shell Environment

# Objectives

- Use Bourne and Korn shell variables
- Assign values to shell variables
- Display the value of shell variables
- Make variables available to subprocesses using `export`
- Display the value of environment variables
- Unset shell and environment variables
- Customize the user environment using the `.profile` file
- Perform arithmetic operations

# Objectives

- Create and use aliases
- Display aliases and the values assigned to them
- Define the built-in aliases
- Customize the Bourne and Korn shell environments
- Use the tilde expansion and command substitution features of the Korn shell

# Reviewing User Startup Scripts

- `/etc/profile` runs first when a user logs in.
- `$HOME/.profile` runs second when a user logs in.
- `$HOME/.kshrc` runs third if the `ENV` variable is set.

# Shell User Environment

- Example `.profile` script

```
$ cat /.profile
# This file initially did NOT exist for root
MANPATH=$MANPATH:/usr/share/man:/usr/dt/share/man:/usr/java1.2/man
ENV=$HOME/.kshrc
EDITOR=vi
export MANPATH ENV EDITOR
```

- To change to the Korn shell, add these lines:

```
SHELL=/usr/bin/ksh  # This variable determines the default shell
                    # for subshells windows
export SHELL
/usr/bin/ksh        # Invokes a Korn shell as a child of the login
shell
```

- Example `.kshrc` script

```
$ cat .kshrc
PS1="HOSTNAME ! $"
set -o trackall
alias l='ls -laF'
alias ls='ls -aF'
alias hi='fc -l'
alias c=clear
```

# A Review of Variables

```
$ set
AB2_DEFAULTSERVER=http://docs.sun.com/
CUE_HOSTNAME=sunray10
DISPLAY=:46.0
DOMAIN=renegades.Central.Sun.COM
DOMAIN_COUNT=1
EDITOR=vi
ERRNO=25
FCEDIT=/bin/ed
LANG=C
LOGNAME=milner
LPDEST=hutchence
MAIL=/var/mail/milner
MAILCHECK=600
MANPATH=/usr/dt/man:/usr/man:/usr/openwin/share/man:/usr/man:/usr/
openwin/share/man:/usr/dt/man:/usr/dist/local/man/5.7
PRINTER=hutchence
PS1='$ '
PS2='> '
PS3='#? '
PS4='+ '
PWD=/home/milner/K-Shell-Course/InstructorGuide/Examples
SHELL=/bin/ksh
TZ=US/Mountain

$ env
DTSOURCEPROFILE=true
DOMAIN=renegades.Central.Sun.COM
DTUSERSESSION=milner-sunray10-46
EDITOR=vi
LOGNAME=milner
MAIL=/var/mail/milner
CUE_HOSTNAME=sunray10
PRINTER=hutchence
DISPLAY=:46.0
TERM=dtterm
TZ=US/Mountain
LPDEST=hutchence
DOMAIN_COUNT=1
```

# Special Shell Variables

- `$` Contains the process identification number of the current process
- `?` Contains the exit status of the most recent foreground process
- `!` Contains the process ID of the last background job started

# Creating Variables in the Shell

- Create the variable, and give it a value with `var=`*`value`*
- Unset the value, and release the variable with `unset`
- Make the variable known to subshells with `export`
- Display the value of a variable with `echo $var`

# Exporting Variables to Subshells

- Created variables are local unless exported
- Environment variables are passed to subprocesses

# Reserved Variables

- Be careful about changing the values of these variables.
- The shell uses these variables.
- For a complete list of reserved variables, read the man page for `sh` or `ksh`.

# Review of Quoting Characters

- Backslash – Alters the special meaning of the following character
- Single quotes – Turns off the special meaning of all characters
- Double quotes – Turns off the special meaning of characters except $, ‘, ", and \

# Shell Command Substitution

- The Bourne shell uses ‘‘ (backquotes)
- The Korn shell supports the older Bourne shell syntax
- The Korn shell uses the `$(`*`command`*`)` syntax

# Korn Shell Tilde Expansion

- `~/` The full path name of user's home directory
- `~`*`username`* The full path name of *`username`*'s home directory
- `~+` The full path name of the working directory
- `~-` The previous working directory
- `--` The previous working directory

# Arithmetic Operations on Bourne Shell Variables

- The Bourne shell only assigns string values to variables
- The Bourne shell has no built-in arithmetic
- The external statement `expr` treats the variables as numbers and performs arithmetic operations

# Arithmetic Operations on Korn Shell Variables

- Arithmetic evaluation is invoked by placing an integer expression within two pairs of parentheses.

  `(( ... ))`

- All calculations are performed using integer arithmetic.

# Arithmetic Precedence

1. Expressions within parentheses are evaluated first.
2. `*`, `%`, and `/` have greater precedence than `+` and `-`.
3. Everything else is evaluated left-to-right.

# The Korn Shell `let` Statement

- The `let` statement is an alternative to the `((...))` statement.
- Type the arithmetic formula with no spaces unless the formula is enclosed in double-quote (") characters.
- It is more common to use the Korn shell's `((...))` syntax instead of the `let` statement.

# Math in Scripts

```
$ cat math.ksh
#!/bin/ksh

# Script name: math.ksh

# This script finds the cube of a number, and the
# quotient and remainder of the number divided by 4.

y=99

(( cube = y * y * y ))
(( quotient = y / 4 ))
(( rmdr = y % 4 ))

print "The cube of $y is $cube."
print "The quotient of $y divided by 4 is $quotient."
print "The remainder of $y divided by 4 is $rmdr."

# Notice the use of parenthesis to
# control the order of evaluating.
(( z = 2 * (quotient * 4 + rmdr) ))
print "Two times $y is $z.""

$ ./math.ksh
The cube of 99 is 970299.
The quotient of 99 divided by 4 is 24.
The remainder of 99 divided by 4 is 3.
Two times 99 is 198.
```

# Korn Shell Aliases

- An alias is a way of assigning a simple name to what might be a long or complicated command or series of commands.
- Variables hold data; aliases hold commands.
- Aliases can specify a version of a command when more than one version of the command is on the system.

# Built-in Aliases

- `functions='typeset -f'`
- `history='fc -l'`
- `integer='typeset -i'`
- `nohup='nohup'`
- `r='fc -e -'`
- `suspend='kill -STOP $$'`

# Shell Parse Order

1. Read command
2. Keyword
3. Alias
4. Built-in commands
5. Functions
6. Tilde expansion
7. Command substitution
8. Arithmetic expression substitution
9. Metacharacters for expansion of file names
10. Command or script lookup
11. Execute the command

# Exercise: Shell Environment

- Objectives
- Tasks
- Discussion
- Solutions

# Check Your Progress

- Use Bourne and Korn shell variables
- Assign values to shell variables
- Display the value of shell variables
- Make variables available to subprocesses using `export`
- Display the value of environment variables
- Unset shell and environment variables
- Customize the user environment using the `.profile` file
- Perform arithmetic operations

# Check Your Progress

- Create and use aliases
- Display aliases and the values assigned to them
- Define the built-in aliases
- Customize the Bourne and Korn shell environments
- Use the tilde expansion and command substitution features of the Korn shell

# Module 4

## Regular Expressions and `grep`

# Objectives

- Use and describe regular expressions
- Describe the `grep` command
- Use the `grep` command to find patterns in a file
- Use regular expression characters with the `grep` command

# What Is grep?

- Globally searches for a regular expression, and prints the results
- Searches text files for a specific pattern
- When pattern is found, the entire line is printed
- Regular expression characters are permitted in the search pattern

```
$ ps -ef | grep msxyz
$ ps -ef | grep dtterm
    352 ??       0:00 dtterm
    353 ??       0:13 dtterm
    354 ??       0:11 dtterm
   1766 pts/5    0:00 dtterm
```

# grep Options

| Option | Meaning |
|---|---|
| `-i` | Ignores uppercase and lowercase |
| `-c` | Prints a count of lines that match |
| `-l` | Prints the name of the files in which the lines match |
| `-v` | Prints the lines that do not contain the search pattern |
| `-n` | Prints the line numbers |

# Regular Expression Metacharacters

| Metacharacter | Function |
|---|---|
| \ | Escapes the special meaning of an RE character |
| ^ | Matches the beginning of the line |
| $ | Matches the end of the line |
| \< | Matches beginning of word anchor |
| \> | Matches end of word anchor |
| [] | Matches any one character from the specified set |
| [-] | Matches any one character in the specified range |
| * | Matches zero of more of the preceding character |
| . | Matches any single character |

# Regular Expressions

```
$ ps -ef | grep '[A-Z]'
     UID   PID  PPID  C    STIME TTY      TIME CMD
    root   259   254  1 08:31:41 ?       0:52 /usr/openwin/bin/Xsun :0
-nobanner -auth /var/dt/A:0-o9aWFa
    root   248     1  0 08:31:34 console  0:00 /usr/lib/saf/ttymon -g -h -p ultrabear console
login:  -T sun -d /dev/console -
    root   278   260  0 08:32:13 ?       0:00 /bin/ksh /usr/dt/bin/Xsession
    root   349   341  0 08:32:26 ?       0:01 dtfile -session dtreaWVa
    root   327   324  0 08:32:15 pts/3    0:00 -sh -c       unset DT;       DISPLAY=:0;
/usr/dt/bin/dtsession_res -merge
    root   324   278  0 08:32:15 pts/3    0:00 /usr/dt/bin/sdt_shell -c           unset DT;
DISPLAY=:0;            /usr/dt/bin/dt
<output truncated>

$ ps -ef | grep 'A'
    root   259   254  1 08:31:41 ?       0:52 /usr/openwin/bin/Xsun :0 -nobanner -auth /var/dt/
A:0-o9aWFa
    root   327   324  0 08:32:15 pts/3    0:00 -sh -c       unset DT;       DISPLAY=:0;
/usr/dt/bin/dtsession_res -merge
    root   324   278  0 08:32:15 pts/3    0:00 /usr/dt/bin/sdt_shell -c           unset DT;
DISPLAY=:0;            /usr/dt/bin/dt
```

# Escaping a Regular Expression

- A \ (backslash) character escapes the RE characters.
- It interprets the next character literally, not as a metacharacter.
- A backslash gives special meaning to the next character, such as \<, \>, \(, and \).

# Escaping a Regular Expression

```
$ grep '$' /etc/init.d/nfs.server
#!/sbin/sh
#
# Copyright (c) 1997-1999 by Sun Microsystems, Inc.
# All rights reserved.
#
#pragma ident   "@(#)nfs.server 1.30    99/06/10 SMI"

[ ! -d /usr/bin ] && exit

# Start/stop processes required for server NFS
<output truncated>

$ grep '$' /etc/init.d/nfs.server | wc -l
     128

$ wc -l /etc/init.d/nfs.server
     128 /etc/init.d/nfs.server

$ grep '\$' /etc/init.d/nfs.server
case "$1" in
    >/tmp/sharetab.$$
       [ "x$fstype" != xnfs ] && \
       echo "$path\t$res\t$fstype\t$opts\t$desc" \
       >>/tmp/sharetab.$$
       /usr/bin/touch -r /etc/dfs/sharetab /tmp/sharetab.$$
       /usr/bin/mv -f /tmp/sharetab.$$ /etc/dfs/sharetab
<output truncated>


$ grep '\$' /etc/init.d/nfs.server | wc -l
    15
```

# Line Anchors

- Use ^ for the beginning of the line
- Use $ for the end of the line

# Line Anchors

```
$ grep 'root' /etc/group
root::0:root
bin::2:root,bin,daemon
sys::3:root,bin,sys,adm
adm::4:root,adm,daemon
uucp::5:root,uucp
<output truncated>

$ grep '^root' /etc/group
root::0:root

$ grep 'mount$' /etc/vfstab
#device      device       mount        FS      fsck    mount   mount
```

# Word Anchors

- Use \< for the beginning of the word
- Use \> for the end of the word

# Word Anchors

```
$ grep '\<uucp'  /etc/group
uucp::5:root,uucp

$ grep 'the' /etc/init.d/nfs.server
if [ -f /etc/dfs/sharetab ] ; then
 # Retain the last modification time so that it can be truncated
 # by the share command when it is called first time after boot.
 # lines, then run shareall to export them, and then start up mountd
<output truncated>

$ grep '\<the\>' /etc/init.d/nfs.server
 # Retain the last modification time so that it can be truncated
 # by the share command when it is called first time after boot.
 # logging enabled, or they were shared in the previous session
 # When the system comes up umask is not set; so set the mode now
 # the grace period
```

# Character Classes

- `[abc]` Finds a single character in the class
- `[a-c]` Finds a single character in the range
- `[^a-c]` Finds a single character not in the range

# Character Classes

```
$ grep '[iu]' /etc/group
bin::2:root,bin,daemon
sys::3:root,bin,sys,adm
uucp::5:root,uucp
mail::6:root
nuucp::9:root,nuucp
sysadmin::14:
nogroup::65534:

$ grep '[u-y]' /etc/group
sys::3:root,bin,sys,adm
uucp::5:root,uucp
tty::7:root,tty,adm
nuucp::9:root,nuucp
sysadmin::14:
nobody::60001:
nogroup::65534:

$ grep '\<[Tt]he\>' teams
The teams are chosen randomly.
```

# Single Character Match

- Matches any single character with “ **.** ”

```
$ grep 'c...h' /usr/dict/words
$ grep '^c...h' /usr/dict/words
$ grep '^c...h$' /usr/dict/words
```

# Closure (*)

- Matches the preceding character zero or more times

```
$ grep 'Team*' teams
Team one consists of
Team two consists of
Tea for two and Dom
Tea for two and Tom

$ grep '\<T.*m\>' teams
Team one consists of
Tom
Team two consists of
Tea for two and Dom
Tea for two and Tom

$ grep '*' teams

$ grep 'abc' *
data1:abcd
```

# Exercise: Regular Expressions and `grep`

- Objectives
- Tasks
- Discussion
- Solutions

# Check Your Progress

- Use and describe regular expression
- Describe the `grep` command
- Use the `grep` command to find patterns in a file
- Use regular expression characters with the `grep` command

# Module 5

## The sed Editor

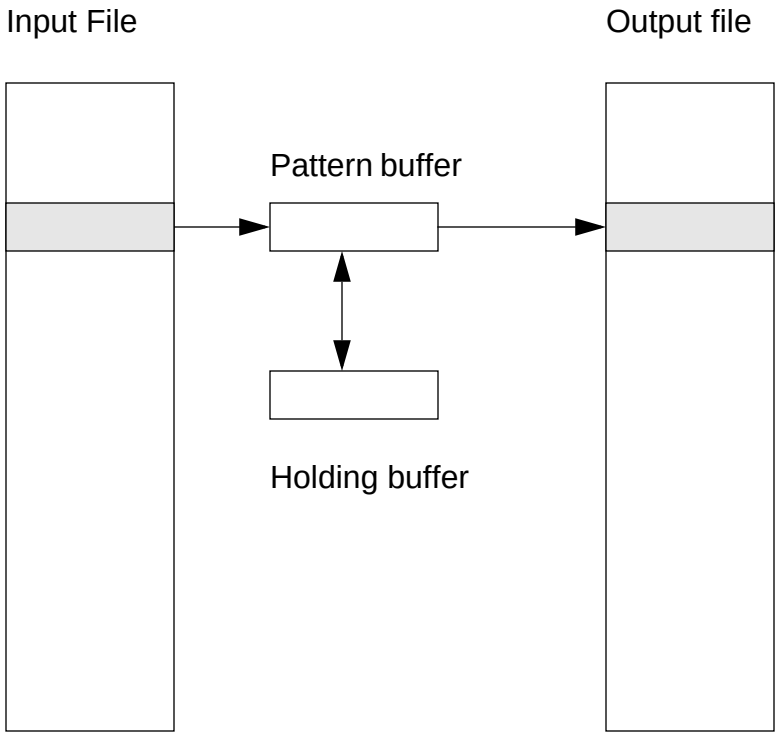
# Objectives

- Use the `sed` editor to perform noninteractive editing tasks
- Use regular expression characters with the `sed` command

# The sed Editor

- A stream editor
- Nondestructive
- Noninteractive
- Uses regular expressions

# The sed Editor



# Command Format

```
sed [options] '[address(s)] action [args]'
      file(s) [ > outfile]
```

# Editing Commands

| Command | Function |
|---|---|
| d | Deletes a line or lines |
| p | Prints a line or lines |
| r | Reads a file |
| s | Substitutes one string for another |

## sed Options

| Option | Function |
|---|---|
| -n | Suppresses the default output |
| -f | Reads sed commands from a script file |

# Addressing

- Specifies a single line number or range of line numbers
- Uses $ for last line of file
- Searches for a regular expression
- Delimits a regular expression with a forward slash

# Using sed to Print Text

```
$ sed '3,5p' data.file
northwest       NW      Joel Craig      3.0 .98 3       4
western         WE      Sharon  Kelly   5.3 .97 5       23
southwest       SW      Chris Foster    2.7 .8  2       18
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 4       15
southern        SO      May Chin        5.1 .95 4       15
southeast       SE      Derek Johnson   5.0 .70 4       17
southeast       SE      Derek Johnson   5.0 .70 4       17
eastern         EA      Susan Beal      4.4 .8  5       20
northeast       NE      TJ Nichols      5.1 .94 3       13
north           NO      Val Shultz      4.5 .89 5       9
central         CT      Sheri Watson    5.7 .94 5       13

$ sed -n '3,5p' data.file
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 4       15
southeast       SE      Derek Johnson   5.0 .70 4       17

$ sed -n '/west/p' data.file
northwest       NW      Joel Craig      3.0 .98 3       4
western         WE      Sharon  Kelly   5.3 .97 5       23
southwest       SW      Chris Foster    2.7 .8  2       18

$ sed -n '/west/,/southern/p' data.file
northwest       NW      Joel Craig      3.0 .98 3       4
western         WE      Sharon  Kelly   5.3 .97 5       23
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 4       15

$ sed -n '/Chris/,$p' data.file
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 4       15
southeast       SE      Derek Johnson   5.0 .70 4       17
eastern         EA      Susan Beal      4.4 .8  5       20
northeast       NE      TJ Nichols      5.1 .94 3       13
north           NO      Val Shultz      4.5 .89 5       9
central         CT      Sheri Watson    5.7 .94 5       13
```

# Using `sed` to Substitute Text

- Substitute a new string for an old string

  ```
  sed 's/oldstring/newstring/' file
  ```

- Use `g` to substitute globally
- Use `&` to include the `oldstring` in the `newstring`

# Using sed to Substitute Text

```
$ sed  's/3/X/' data.file
northwest       NW      Joel Craig      X.0 .98 3       4
western         WE      Sharon  Kelly   5.X .97 5       23
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 4       15
southeast       SE      Derek Johnson   5.0 .70 4       17
eastern         EA      Susan Beal      4.4 .8  5       20
northeast       NE      TJ Nichols      5.1 .94 X       13
north           NO      Val Shultz      4.5 .89 5       9
central         CT      Sheri Watson    5.7 .94 5       1X
$ sed  's/3/X/g' data.file
northwest       NW      Joel Craig      X.0 .98 X       4
western         WE      Sharon  Kelly   5.X .97 5       2X
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 4       15
southeast       SE      Derek Johnson   5.0 .70 4       17
eastern         EA      Susan Beal      4.4 .8  5       20
northeast       NE      TJ Nichols      5.1 .94 X       1X
north           NO      Val Shultz      4.5 .89 5       9
central         CT      Sheri Watson    5.7 .94 5       1X

$ sed  -n '/ [0-9]$/p' data.file
northwest       NW      Joel Craig      3.0 .98 3       4
north           NO      Val Shultz      4.5 .89 5       9

$ sed 's/ [0-9]$/& Single Digit/' data.file
northwest       NW      Joel Craig      3.0 .98 3       4 Single Digit
western         WE      Sharon  Kelly   5.3 .97 5       23
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 4       15
southeast       SE      Derek Johnson   5.0 .70 4       17
eastern         EA      Susan Beal      4.4 .8  5       20
northeast       NE      TJ Nichols      5.1 .94 3       13
north           NO      Val Shultz      4.5 .89 5       9 Single Digit
central         CT      Sheri Watson    5.7 .94 5       13
```

# Reading From a File of New Text

- Read in a file after the line containing the search expression.
- The `r` (read) command is followed by the path name to the file.

# Reading From a File of New Text

```
$ cat northmesg
*** The northern regions are the newest in the company ***
*** and the people are still being trained. **************


$ sed '/north/r northmesg' data.file
northwest       NW      Joel Craig      3.0 .98 3       4
*** The northern regions are the newest in the company ***
*** and the people are still being trained. **************
western         WE      Sharon  Kelly   5.3 .97 5       23
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 4       15
southeast       SE      Derek Johnson   5.0 .70 4       17
eastern         EA      Susan Beal      4.4 .8  5       20
northeast       NE      TJ Nichols      5.1 .94 3       13
*** The northern regions are the newest in the company ***
*** and the people are still being trained. **************
north           NO      Val Shultz      4.5 .89 5       9
*** The northern regions are the newest in the company ***
*** and the people are still being trained. **************
central         CT      Sheri Watson    5.7 .94 5       13
```

# Using sed to Delete Text

- Delete lines containing the search expression
- Delete lines in the address range
- Do not delete lines using !

# Using sed to Delete Text

```
$ sed '4,8d' data.file
northwest       NW      Joel Craig      3.0 .98 3       4
western         WE      Sharon  Kelly   5.3 .97 5       23
southwest       SW      Chris Foster    2.7 .8  2       18
central         CT      Sheri Watson    5.7 .94 5       13

$ sed '/west/d' data.file
southern        SO      May Chin        5.1 .95 4       15
southeast       SE      Derek Johnson   5.0 .70 4       17
eastern         EA      Susan Beal      4.4 .8  5       20
northeast       NE      TJ Nichols      5.1 .94 3       13
north           NO      Val Shultz      4.5 .89 5       9
central         CT      Sheri Watson    5.7 .94 5       13

$ sed '/^west/d' data.file
northwest       NW      Joel Craig      3.0 .98 3       4
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 4       15
southeast       SE      Derek Johnson   5.0 .70 4       17
eastern         EA      Susan Beal      4.4 .8  5       20
northeast       NE      TJ Nichols      5.1 .94 3       13
north           NO      Val Shultz      4.5 .89 5       9
central         CT      Sheri Watson    5.7 .94 5       13

$ sed '/south/,/north/d' data.file
northwest       NW      Joel Craig      3.0 .98 3       4
western         WE      Sharon  Kelly   5.3 .97 5       23
north           NO      Val Shultz      4.5 .89 5       9
central         CT      Sheri Watson    5.7 .94 5       13
```

# Reading `sed` Commands From a File

- Place the commands in a file
- Use `-f` to tell `sed` to read the file

# Reading sed Commands From a File

```
$ cat script1.sed
1,4d
s/north/North/
s/^east/East/

$ sed -f script1.sed data.file
southeast       SE      Derek Johnson   5.0 .70 4       17
Eastern         EA      Susan Beal      4.4 .8  5       20
Northeast       NE      TJ Nichols      5.1 .94 3       13
North           NO      Val Shultz      4.5 .89 5       9
central         CT      Sheri Watson    5.7 .94 5       13
```

# Using sed to Write Output Files

- Writes the specified records to the named file
- The `w` (write) command is followed by the path name of the file

# Using sed to Write Output Files

```
$ cat script5.sed
/north/w northregions
s/9[0-9]/& Great job!/w topperformers

$ sed -n -f script5.sed data.file

$ more northregions topperformers
::::::::::::::
northregions
::::::::::::::
northwest     NW      Joel Craig      3.0 .98 3       4
northeast     NE      TJ Nichols      5.1 .94 3       13
north         NO      Val Shultz      4.5 .89 5       9
::::::::::::::
topperformers
::::::::::::::
northwest     NW      Joel Craig      3.0 .98 Great job!      3       4
western       WE      Sharon  Kelly   5.3 .97 Great job!      5       23
southern      SO      May Chin        5.1 .95 Great job!      4       15
northeast     NE      TJ Nichols      5.1 .94 Great job!      3       13
central       CT      Sheri Watson    5.7 .94 Great job!      5       13
```

# Exercise: The sed Editor

- Objectives
- Tasks
- Discussion
- Solutions

# Check Your Progress

- Use the sed editor to perform noninteractive editing tasks
- Use regular expression characters with the sed command

# Module 6

## The nawk Programming Language

# Objectives

- Use nawk commands from the command line
- Write simple nawk programs to generate data reports from text files
- Write simple nawk programs to generate numeric and text reports from text files

# Introduction to nawk

- Looks at data by records and fields
- Uses regular expressions
- Uses numeric and text variables and functions
- Uses command-line arguments

# nawk Format

- Commands have the form:

  nawk *'statement' input.file*

- Scripts are executed with:

  nawk -f *scriptfile input.file*

# Using nawk to Print Selected Fields

- Command conventions:
  - Enclose the command in single quotes
  - Enclose the command in braces `{}`
- Specify individual records with `$0`
- Specify individual fields with `$1`, `$2`, `$3`, and so on

# Using nawk to Print Selected Fields

```
$ cat data.file
northwest       NW      Joel Craig      3.0 .98 3       4
western         WE      Sharon Kelly    5.3 .97 5       23
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 4       15
southeast       SE      Derek Johnson   5.0 .70 4       17
eastern         EA      Susan Beal      4.4 .8  5       20
northeast       NE      TJ Nichols      5.1 .94 3       13
north           NO      Val Shultz      4.5 .89 5       9
central         CT      Sheri Watson    5.7 .94 5       13

$ nawk '{ print $3, $4, $2 }' data.file
Joel Craig NW
Sharon Kelly WE
Chris Foster SW
May Chin SO
Derek Johnson SE
Susan Beal EA
TJ Nichols NE
Val Shultz NO
Sheri Watson CT
```

# Formatting With `print`

- `\t` Tab
- `\n` Newline
- `\007` Bell
- `\011` Tab
- `\012` Newline
- `\042` "
- `\045` %

# Formatting With `print`

```
$ nawk '{ print $3, $4 "\t" $2 }' data.file
Joel Craig       NW
Sharon Kelly     WE
Chris Foster     SW
May Chin         SO
Derek Johnson    SE
Susan Beal       EA
TJ Nichols       NE
Val Shultz       NO
Sheri Watson     CT
```

# Using Regular Expressions

- Use the same regular expression characters as in `egrep`
- Special patterns – `BEGIN` and `END`

# Using Regular Expressions

```
$ nawk '/east/' data.file
southeast       SE      Derek Johnson   5.0 .70 4       17
eastern         EA      Susan Beal      4.4 .8  5       20
northeast       NE      TJ Nichols      5.1 .94 3       13

$ nawk '/east/ { print $1, $5, $4 }' data.file
southeast 5.0 Johnson
eastern 4.4 Beal
northeast 5.1 Nichols

$ nawk '/east/ { print $1, $5 "\t" $4 }' data.file
southeast 5.0   Johnson
eastern 4.4     Beal
northeast 5.1   Nichols

$ nawk '/^east/' data.file
eastern         EA      Susan Beal      4.4 .8  5       20

$ nawk '/.9/' data.file
northwest       NW      Joel Craig      3.0 .98 3       4
western         WE      Sharon  Kelly   5.3 .97 5       23
southern        SO      May Chin        5.1 .95 4       15
northeast       NE      TJ Nichols      5.1 .94 3       13
north           NO      Val Shultz      4.5 .89 5       9
central         CT      Sheri Watson    5.7 .94 5       13

$ nawk '/\.9/' data.file
northwest       NW      Joel Craig      3.0 .98 3       4
western         WE      Sharon  Kelly   5.3 .97 5       23
southern        SO      May Chin        5.1 .95 4       15
northeast       NE      TJ Nichols      5.1 .94 3       13
central         CT      Sheri Watson    5.7 .94 5       13
```

# Special Patterns – `BEGIN` and `END`

- `BEGIN` An action to take before reading any lines
- `END` An action to take after all lines are read and processed

# Special Patterns – BEGIN and END

```
$ nawk 'BEGIN { print "Eastern Regions\n" }; /east/ { print $5,
$4 }' data.file
Eastern Regions

5.0 Johnson
4.4 Beal
5.1 Nichol

$ nawk 'BEGIN {
> print "Eastern Regions\n"};  /east/ {print $5, $4}' data.file
Eastern Regions

5.0 Johnson
4.4 Beal
5.1 Nichols

$ nawk 'BEGIN
> { print "Eastern Regions\n" }; /east/ { print $5, $4 }'
data.file
nawk: syntax error at source line 2
 context is
        BEGIN >>>
 <<<
nawk: bailing out at source line 2

$ nawk 'BEGIN { print "Eastern Regions\n"};  /east/ {print $5,
$4}
> END {print "Eastern Region Monthly Report"}' data.file
Eastern Regions

5.0 Johnson
4.4 Beal
5.1 Nichols
Eastern Region Monthly Report
```

# Programming With nawk

```
$ cat report
BEGIN {print "Eastern Regions\n"}
/east/ {print $5, $4}
END {print "Eastern Region Monthly Report"}

$ nawk -f report  data.file
Eastern Regions

5.0 Johnson
4.4 Beal
5.1 Nichols
Eastern Region Monthly Report

$ cat report2
BEGIN {print "** Acme Enterprises **"}
BEGIN {print "Eastern Regions\n"}
/east/ {print $5, $4}
END {print "Eastern Region Monthly Report"}
$ nawk -f report2  data.file
** Acme Enterprises **
Eastern Regions

5.0 Johnson
4.4 Beal
5.1 Nichols
Eastern Region Monthly Report
```

# Using Built-in Variables

| Name | Default Value | Description |
|---|---|---|
| OFS | Space | The output field separator |
| FS | Space or tab | The input field separator |
| NR | | The number of records from the beginning of the first input file |

# Working With Variables

- Input field separator

```
nawk -F:
nawk 'BEGIN {FS=":"}
```

- Output field separator

```
nawk '{print $3, $4 "\t" $2}' data.file
nawk 'BEGIN {OFS="\t"}
```

# Exercise: nawk and Regular Expressions

- Objectives
- Tasks
- Discussion
- Solutions

# User-Defined Variables

- Variable names should not conflict with the `nawk` variable or function names
- Variables are automatically initialized to a null string
- Variables used in arithmetic statements or functions are initialized to 0

# User-Defined Variables

```
$ cat numexample.nawk
{ counter = counter + 1 }
{ print $0 }
END { print "*** The number of records is " counter }

$ nawk -f numexample.nawk data.file
northwest       NW      Joel Craig      3.0 .98 3       4
western         WE      Sharon  Kelly   5.3 .97 5       23
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 4       15
southeast       SE      Derek Johnson   5.0 .70 4       17
eastern         EA      Susan Beal      4.4 .8  5       20
northeast       NE      TJ Nichols      5.1 .94 3       13
north           NO      Val Shultz      4.5 .89 5       9
central         CT      Sheri Watson    5.7 .94 5       13
*** The number of records is 9

$ cat numexample2.nawk
{ total = total + $8 }
{ print "Field 8 = " $8 }
END { print "Total = " total }

$ nawk -f numexample2.nawk data.file
Field 8 = 4
Field 8 = 23
Field 8 = 18
Field 8 = 15
Field 8 = 17
Field 8 = 20
Field 8 = 13
Field 8 = 9
Field 8 = 13
Total = 132
```

# User-Defined Variables

```
$ cat numexample3.nawk
{ total = total + $8 }
{ print $0 }
END { print "The total of field 8 is " total }
$ nawk -f numexample2.nawk data.file
northwest       NW      Joel Craig      3.0 .98 3       4
western         WE      Sharon  Kelly   5.3 .97 5       23
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 4       15
southeast       SE      Derek Johnson   5.0 .70 4       17
eastern         EA      Susan Beal      4.4 .8  5       20
northeast       NE      TJ Nichols      5.1 .94 3       13
north           NO      Val Shultz      4.5 .89 5       9
central         CT      Sheri Watson    5.7 .94 5       13
The total of field 8 is 132
```

# Variable Examples

```
$ nawk '/N/ { print NR, $0 }' data.file
1 northwest     NW      Joel Craig      3.0 .98 3       4
7 northeast     NE      TJ Nichols      5.1 .94 3       13
8 north         NO      Val Shultz      4.5 .89 5       9

$ nawk 'BEGIN { count = 0 }
>  /N[EOW]/ { print NR, $0; count = count + 1 }
>  END { print "count of North regions is", count }' data.file
1 northwest     NW      Joel Craig      3.0 .98 3       4
7 northeast     NE      TJ Nichols      5.1 .94 3       13
8 north         NO      Val Shultz      4.5 .89 5       9
count of North regions is 3

$ nawk '{ print "Record:", NR, $NF }' data.file
Record: 1 4
Record: 2 23
Record: 3 18
Record: 4 15
Record: 5 17
Record: 6 20
Record: 7 13
Record: 8 9
Record: 9 13

$ cat raggeddata.file
northwest       NW      Joel Craig      3.0 .98 3       4
WE      Sharon  Kelly   5.3 .97 23
southwest       SW      Chris Foster    2.7 .8  2       18
southern        SO      May Chin        5.1 .95 15
southeast       SE      Derek   5.0     17
eastern         Susan Beal      4.4 .8          20
NE      TJ Nichols      5.1 .94 3       13
Val Shultz      4.5     5       9
central         CT      Sheri Watson    .94     5
```

# Variable Examples

```
$ nawk '{ print "Record:", NR, "has", NF, "fields." }'
raggeddata.file
Record: 1 has 8 fields.
Record: 2 has 6 fields.
Record: 3 has 8 fields.
Record: 4 has 7 fields.
Record: 5 has 5 fields.
Record: 6 has 6 fields.
Record: 7 has 7 fields.
Record: 8 has 5 fields.
Record: 9 has 6 fields.

$ nawk '{ print "Field 1 has", length($1), "letters." }'
raggeddata.file
Field 1 has 9 letters.
Field 1 has 2 letters.
Field 1 has 9 letters.
Field 1 has 8 letters.
Field 1 has 9 letters.
Field 1 has 7 letters.
Field 1 has 2 letters.
Field 1 has 3 letters.
Field 1 has 7 letters.
```

# Writing Output to Files

- Use the redirection symbol, >, to send data to a file.

```
$ nawk '{ print $2, $1 > "textfile" }' data.file
$ cat textfile
NW northwest
WE western
SW southwest
SO southern
SE southeast
EA eastern
NE northeast
NO north
CT central
```

- Use two redirection symbols, >>, to append to a file.

# `printf()` Statement

```
printf( ”Hello World\n” )
printf( ”The value is %d\n”, num )
printf( ”The result is %10.2f\n”, num / 6 * 23 )
printf( ”My name is %-10s %20s\n”, $3, $4 )
```

```
$ nawk ’{ printf "%10s %3d \n", $4, $7 }’ data.file
     Craig     3
     Kelly     5
    Foster     2
      Chin     4
   Johnson     4
      Beal     5
   Nichols     3
    Shultz     5
    Watson     5
```

# Guided nawk Script

1. `quot /`
2. `quot / | grep -v "/dev" > users1`
3. `head -3 users1`
4. `head -3 users1 > users2`
5. `nawk` report on disk usage
6. Shell script to automate the steps

# Exercise: nawk Scripts

- Objectives
- Tasks
- Discussion
- Solutions

# Check Your Progress

- Use nawk commands from the command line
- Write simple nawk programs to generate data reports from text files
- Write simple nawk programs to generate numeric and text reports from text files

# Module 7

## Conditionals

# Objectives

- Use the exit status of a command as a conditional control
- Use the `if` statement to test a condition
- Pass values using command-line arguments (positional parameters) into a script
- Create USAGE messages
- Place parameters on the command line
- Use conditional constructs `if`, `then`, `elif`, `else`, `fi`
- Use `exit`, `let` and test statements ( `[[ ]]`, `(( ))` )

# Objectives

- Apply Boolean logic &&, ||, and !
- Use the `case` statement

# The `if` Statement

```
if command
then
    block of statements
fi
```

```
$ cat snoopy.sh
#!/bin/sh

# Script name: snoopy.sh

name=snoopy

if [ "$name" = "snoopy" ]
then
        echo "It was a dark and stormy night."
fi
```

# Parts of the `if` Statement

- Command
- Block of statements
- End if

# Exit Status

- Every command, program, and shell statement executed has an exit status associated with it
- The exit status is an integer variable and is saved in the shell reserved variable `?`
- A value of `0` for the exit status indicates the command ran successfully (no errors occurred)
- A nonzero exit status indicates failure (one or more errors occurred or the command could not accomplish what was asked)

# Exit Status

```
$ grep root /etc/passwd
root:x:0:1:Super-User:/:/sbin/sh
$ echo $?
0

$ grep root /etc/passwd > /dev/null
$ echo $?
0
$ grep root /etc/passwd
root:x:0:1:Super-User:/:/sbin/sh
$ if [ $? -eq 0 ]
> then
>       echo "Found root!"
> fi
Found root!

$ if grep root /etc/passwd
> then
>       echo "Found root!"
> fi
root:x:0:1:Super-User:/:/sbin/sh
Found root!

$ if grep root /etc/passwd > /dev/null
> then
>       echo "Found root!"
> fi
Found root!
```

# Numeric and String Comparison

- Number comparison:
  - Bourne and Korn: Use `[]` and spaces
  - Korn: Use `(( ))` and spaces are optional
- String comparison:
  - Bourne and Korn: Use `[ ]` and spaces
  - Korn: Use `[[ ]]` and spaces

# `if/then/else` Syntax

```
if command
then
      block of statements
else
      block of statements
fi
```

```
$ cat snoopynap.ksh
#!/bin/ksh

# Script name: snoopynap.ksh

name=snoopy

if [[ "$name" == "snoopy" ]]
then
        echo "It was a dark and stormy night."
else
        echo "Snoopy is napping."
fi
```

# if/then/elif/else Syntax

```
if command1
then
       block of statements
elif command2
then
       block of statements
else
       block of statements
fi
```

For example:

```
$ cat snoopy2.ksh
#!/bin/ksh

# Script name: snoopy2.ksh

name=snoopy

if [[ "$name" == "snoopy" ]]
then
        echo "It was a dark and stormy night."
elif [[ "$name" == "charlie" ]]
then
        echo "You're a good man Charlie Brown."
elif [[ "$name" == "lucy" ]]
then
        echo "The doctor is in."
elif [[ "$name" == "schroder" ]]
then
        echo "In concert."
else
        echo "Not a Snoopy character."
fi
```

# Using `if` to Check Command-Line Arguments

| Positional Parameter Name | Description of Command Line Argument |
|---|---|
| `$0` | The script name |
| `$1` | The value of the first argument to the script |
| `$2` | The value of the second argument to the script |
| `$9` | The value of the ninth argument to the script |
| `${10}` | The value of the tenth argument to the script—Korn shell only; for Bourne shell, use the `shift` statement |
| `${11}`, `${12}`, ... | The value of the eleventh, twelfth, and so on arguments to the script—for Korn shell only |
| `$#` | The number of arguments passed to the script |
| `$*` | The value of all command-line arguments |

# Creating the USAGE Message

- The script should verify the type of input and number of values
- The script can print an error message if the input is not correct

```
if (( $# != 2 )
then
     print “USAGE: $0 arg1 arg2 “
     exit
fi
```

# Using `if` to Check Leap Years

```
$ cat monthcheck
#!/bin/ksh

# Script name: monthcheck

mth=$(date +%m)

if (( mth == 2 ))
then
        echo "February usually has 28 days."
        echo "If it is a leap year, it has 29 days."
elif [[ $mth = @(04|06|09|11) ]]
then
        echo "The current month has 30 days."
else
        echo "The current month has 31 days."
fi

$
$ date
Fri Mar 13 13:28:24 GMT 2000
$
$ ./monthcheck
The current month has 31 days.
$
```

# Nested `if` Statements

- Any command within an `if` statement can be another `if` statement
- Each `if` statement requires its own `fi` statement

# Nested `if` Statements

```
$ cat leap.ksh
#!/bin/ksh

# Script name: leap.ksh

# Assume the user enters the year on the command line
# when they execute the script.

year=$1

if (( (year % 400) == 0 ))
then
        print "$year is a leap year!"
elif (( (year % 4) == 0 ))
then
        if (( (year % 100) != 0 ))
        then
                print "$year is a leap year!"
        else
                print "$year is not a leap year."
        fi
else
        print "$year is not a leap year."
fi

$ ./leap_scr 2000
2000 is a leap year!

$ ./leap_scr 1900
1900 is not a leap year.

$ ./leap_scr 2050
2050 is not a leap year.
```

# Testing File Objects

| Flag | Bourne or Korn Test | Korn Test | Description |
|---|---|---|---|
| `-r` | `[ -r file ]` | `[[ -r file ]]` | Can the file be read by user? |
| `-w` | `[ -w file ]` | `[[ -w file ]]` | Can the file be altered by user? |
| `-x` | `[ -x file ]` | `[[ -x file ]]` | Can the file be executed by user? |
| `-O` | Not available | `[[ -O file ]]` | Is the file owned by the effective user ID of this process? |
| `-G` | Not available | `[[ -G file ]]` | Is the file group the effective group ID of this process? |
| `-u` | `[ -u file ]` | `[[ -u file ]]` | Does the file has the `set-user-ID` bit set? |
| `-g` | `[ -g file ]` | `[[ -g file ]]` | Does the file have the `set-group-ID` bit set? |
| `- k` | `[ -k file ]` | `[[ -k file ]]` | Does the file have the sticky bit set? |

# Boolean AND, OR, and NOT

- AND operator is &&
- OR operator is ||
- NOT operator is !

# The case Statement

- Use case to test compare a value against multiple patterns

```
case value in
pattern1)
        statement1
        ...
        statementn
        ;;

pattern2)
        statement1
        ...
        statementn
        ;;

*)
        statement1
        ...
        statementn
        ;;
esac
```

# Example: Using the case Statement

```
$ cat case.ksh
#!/bin/ksh

# Script name: case.ksh

mth=$(date +%m)

case $mth in
02)
        print "February usually has 28 days."
        print "If it is a leap year, it has 29 days."
        ;;

04|06|09|11)
        print "The current month has 30 days."
        ;;

*)
        print "The current month has 31 days."
        ;;
esac

$ date
Tue Jul 13 08:25:16 PDT 1993

$ ./case_scr2
The current month has 31 days.
$
```

# Replacing Complex `if` Statements With `case`

```
$ cat snoopy2.ksh
#!/bin/ksh

# Script name: snoopy2.ksh

name=snoopy

if [[ "$name" == "snoopy" ]]
then
        echo "It was a dark and stormy night."
elif [[ "$name" == "charlie" ]]
then
        echo "You're a good man Charlie Brown."
elif [[ "$name" == "lucy" ]]
then
        echo "The doctor is in."
elif [[ "$name" == "schroder" ]]
then
        echo "In concert."
else
        echo "Not a Snoopy character."
fi
```

# Replacing Complex `if` Statements With `case`

```
$ cat snoopy3.ksh
#!/bin/ksh

# Script name: snoopy3.ksh

name=lucy

case $name in
"snoopy")
        echo "It was a dark and stormy night."
        ;;

"charlie")
        echo "You're a good man Charlie Brown."
        ;;

"lucy")
        echo "The doctor is in."
        ;;

"schroder")
        echo "In concert."
        ;;

*)
        echo "Not a Snoopy character."
        ;;
esac

#!/bin/ksh
case $a in
   "snoopy") echo "It was a dark and stormy night." ;;
   "charlie") echo "You're a good man Charlie Brown" ;;
   "lucy") echo "The doctor is in." ;;
   *) echo "Not a Snoopy character" ;;
esac
```

# The `exit` Statement

- Terminates the execution of the entire script
- Can be used when:
  - The requested input is incorrect
  - The command ran unsuccessfully
  - Some other error occurred

# Exercise: Conditionals

- Objectives
- Tasks
- Discussion
- Solutions

# Check Your Progress

- Use the exit status of a command as a conditional control
- Use the `if` statement to test a condition
- Pass values using command-line arguments (positional parameters) into a script
- Create USAGE messages
- Place parameters on the command line
- Use conditional constructs `if`, `then`, `elif`, `else`, `fi`
- Use `exit`, `let` and test statements ( `[[ ]]`, `(( ))` )

# Check Your Progress

- Apply Boolean logic &&, ||, and !
- Use the `case` statement

# Module 8

## Interactive Scripts

# Objectives

- Use the `print` and `echo` commands to display text
- Use the `read` command to interactively assign data to a shell variable
- Read user input into one or more variables using one `read` command
- Use special characters, with `print` and `echo`, to make the displayed text more user-friendly
- Create a *here* document
- Use file descriptors to read from and write to multiple files

# Input and Output in a Script

- Read command-line arguments
- Print prompts
- Read input
- Test input
- Print error messages
- Output to a file
- Input from a file

# The Korn Shell `print` Statement

- `print` options

| | |
|---|---|
| `-n` | Suppresses the newline after printing the message. This usually is used when printing a prompt for user input. |
| `-r` | Turns off the special meaning of the `\` character. |
| `-R` | Does not interpret the "`-`" that follows as an option to the `print` statement, except if followed by n; that is, if the `-n` option follows `-R`, it is still taken as an option. This option is useful if you need to print negative numbers. |
| `--` | Same as `-R`, except that a following -n option is taken literally. |

- `print` special characters

| | |
|---|---|
| `\n` | Prints a newline character, which enables you to print a message on several lines using one `print` command |
| `\t` | Prints a tab character, which is useful when creating tables or a report |
| `\a` | Ring the bell on the terminal, which draws the attention of the user |
| `\b` | Backspace one character, which overwrites the previous character |

# Examples: Using `print`

```
$ print "Hello there.\nHow are you?"
Hello there.
How are you?

$ print -r "Hello there.\nHow are you?"
Hello there.\nHow are you?”

$ print "-2 was the temperature this morning."
ksh: print: bad option(s)

$ print -R "-2 was the temperature this morning."
-2 was the temperature this morning.

$ print -- "-2 was the temperature this morning."
-2 was the temperature this morning.

$ print -- -n "is the option."
-n is the option.

$ print -R -n "is the option."
is the option.$

$ print -n "No newline printed here. "
No newline printed here. $

$ print "Hello\tout\tthere!"
Hello   out     there!

$ print "\aListen to me!"
<bell rings>Listen to me!

$ print "Overwrite\b the 'e' in 'Overwrite'."
Overwrit the 'e' in 'Overwrite'.
```

# Examples: Using echo

```
$ echo "Hello there.\nHow are you?"
Hello there.
How are you?

$ echo "Hello there.\\nHow are you?"
Hello there.\nHow are you?”

$ echo "-2 was the temperature this morning."
-2 was the temperature this morning.

$ echo "No newline printed here. \c"
No newline printed here. $

$ echo "Hello\tout\tthere!"
Hello   out     there!

$ echo "\007Listen to me!"
<bell rings>Listen to me!

$ echo "Overwrite\b the 'e' in 'Overwrite'."
Overwrit the 'e' in 'Overwrite'.

$ echo "Type a letter [ ]\b\b\c"
Type a letter [$
```

# The `read` Statement

- The `read` statement reads input from standard input
- The input is broken into tokens that are consecutive characters that do not contain white space
- The contents of the `IFS` variable are used as token delimiters
- The default value of `IFS` is a space, a tab, and a newline
- The first token is saved as the value of the first variable
- If there are more tokens than variables, the last variable holds all remaining tokens

# The `read` Statement

- If there are more variables than tokens, then the extra variables are assigned a null value
- If no variable names are supplied to the `read` command, then:
  - The Korn shell assigns all input to the `REPLY` variable
  - The Bourne shell gives an error message, `read: missing arguments`

# Examples: Using read

```
$ read var1 var2 var3
abc  def  ghi
$ print $var1
abc
$ print $var2
def
$ print $var3
ghi

$ read num string junk
134 bye93;alk the rest of the line is put in 'junk'
$ print $num
134
$ print $string
bye93;alk
$ print $junk
the rest of the line is saved in 'junk'

$ read token1 token2
one
$ echo $token1
one
$ echo $token2

$ read
What is this saved in?
$ print $REPLY
What is this saved in?

$ read
read: missing arguments
```

# Printing a Prompt

- The Korn shell built-in print command has a `-n` option
- The `echo` command allows a `\c` in the output string

# Printing a Prompt

```
$ cat io1.sh
#!/bin/sh

# Script name: io1.sh

# This script prompts for input and prints messages
# involving the input received.

echo "Enter your name: \c"
read name junk

echo "Hi $name, how old are you? \c"
read age junk

echo "\n\t$age is an awkward age, $name,"
echo "   You're too old to depend on your parents,"
echo "and not old enough to depend on your children."

$ ./io1.sh
Enter your name: Murdock.
Hi Murdock, how old are you? 25

        25 is an awkward age, Murdock.
   You're too old to depend on your parents,
 and not old enough to depend on your children.
```

# Prompting for Input – Korn Shell Shortcut

```
$ cat io3.ksh
#!/bin/ksh

# Script name: io3.ksh

# This script prompts for input and prints messages
# involving the input received.

read name?"Enter your name: "

read age?"Hi $name. How old are you? "

print "\n\t$age is an awkward age, $name,"
print "   You're too old to depend on your parents,"
print "and not old enough to depend on your children."

$ ./io3
Enter your name: Murdock
Hi Murdock. How old are you? 25

        25 is an awkward age, Murdock.
   You're too old to depend on your parents,
 and not old enough to depend on your children.
$
```

# File Input and Output

| Command | Description |
|---|---|
| `< file` | Takes standard input from *file* |
| `0< file` | Takes standard input from *file* |
| `> file` | Puts standard output to *file* |
| `1> file` | Puts standard output to *file* |
| `2> file` | Puts standard error to *file* |
| `exec fd> /some/filename` | Assigns the file descriptor *fd* to */some/filename* for output |
| `exec fd< /some/filename` | Assigns the file descriptor *fd* to */some/filename* for input |
| `read <&fd var1` | Reads from the file descriptor *fd* and stores into variable *var1* |
| `cmd >& fd` | Executes *cmd* and send output to the file descriptor *fd* |
| `exec fd<&-` | Closes the file descriptor *fd* |

# User-Defined File Descriptors

exec *fd*> *filename*

exec *fd*< *filename*

*command* >& *fd*

*command* <& *fd*

# File Descriptors in the Bourne Shell

```
$ cat readex2.sh
#!/bin/sh

cp /etc/hosts /tmp/hosts2
grep -v '^#' /tmp/hosts2 > /tmp/hosts3  # Strip out comment lines

exec 3< /tmp/hosts3           # fd 3 is an input file /tmp/hosts3
exec 4> /tmp/hostsfinal       # fd 4 is output file /tmp/hostsfinal

read <& 3 addr1 name1 alias    # read from fd 3

read <& 3 addr2 name2 alias    # read from fd 3

exec 3<&-   # Close fd 3


echo $name1 $addr1 >& 4        # write to fd 4 (do not write aliases)
echo $name2 $addr2 >& 4        # write to fd 4 (do not write aliases)

exec 4<&-   # close fd 4


$ ./readex2.sh
$ more /tmp/hosts2
#
# Internet host table
#
127.0.0.1       localhost
192.9.200.111   ultrabear       loghost
192.9.200.121   ladybear


$ more /tmp/hosts3
127.0.0.1       localhost
192.9.200.111   ultrabear       loghost
192.9.200.121   ladybear
```

# File Descriptors in the Korn Shell

```
$ cat readex.ksh
#!/bin/ksh

cp /etc/hosts /tmp/hosts2
grep -v '^#' /tmp/hosts2 > /tmp/hosts3  # Strip out comment lines

exec 3< /tmp/hosts3          # fd 3 is an input file /tmp/hosts3
exec 4> /tmp/hostsfinal      # fd 4 is output file /tmp/hostsfinal

read -u3 addr1 name1 alias   # read from fd 3 -> NOTE -u option is
Korn shell specific

read -u3 addr2 name2 alias   # read from fd 3

exec 3<&-                    # close fd 3


print -u4 $name1 $addr1      # write to fd 4 (do not write aliases)
print -u4 $name2 $addr2      # write to fd 4 (do not write aliases) ->
NOTE the
                             #   -u option to print is Korn shell
specific

exec 4<&-   # close fd 4



$ ./readex.ksh
$ more /tmp/hosts2
#
# Internet host table
#
127.0.0.1       localhost
192.9.200.111   ultrabear       loghost
192.9.200.121   ladybear
```

# The *here* Document

```
$ cat termheredoc.ksh
#!/bin/ksh

# Script name: termheredoc.ksh

print "Select a terminal type"
cat << ENDINPUT
    sun
    ansi
    wyse50
ENDINPUT

print -n "Which would you prefer? "
read termchoice

print
print "You choice is terminal type: $termchoice"

$ ./termheredoc.ksh
Select a terminal type
    sun
    ansi
    wyse50
Which would you prefer? sun

You choice is terminal type: sun
```

# Exercise: Interactive Scripts

- Objectives
- Tasks
- Discussion
- Solutions

# Check Your Progress

- Use the `print` and `echo` commands to display text
- Use the `read` command to interactively assign data to a shell variable
- Read user input into one or more variables using one `read` command
- Use special characters, with `print` and `echo` to make the displayed text more user-friendly
- Create a *here* document
- Use file descriptors to read from and write to multiple files

# Module 9

## Loops

# Objectives

- Write scripts that use `for`, `while`, and `until` loops
- Write a script using the `select` statement
- Describe when to use loops within a script
- Generate argument lists using command, variable, and file name substitution

# Shell Loops

- Repeatedly execute a block of statements
- `for` loop – Executes for each value in a list
- `while` loop – Executes while a condition is true
- `until` loop – Executes until a condition is true

# The `for` Loop

```
for var in argument_list ...
do
        statement1
        ...
        statementn
done
```

# The `for` Loop Argument List

- Explicit list
- Variable's contents
- Command-line arguments
- Command substitution
- File-name substitution

# Explicit List

```
for var in arg1 arg2 arg3 arg4 ... argn
```

```
for fruit in apple orange banana peach kiwi
   do
   print "Value of fruit is:  $fruit"
done
```

## The output for this `for` loop is:

```
Value of fruit is:  apple
Value of fruit is:  orange
Value of fruit is:  banana
Value of fruit is:  peach
Value of fruit is:  kiwi
```

# Variable's Contents

```
for var in $var_sub

$ cat ex1.sh
#!/bin/sh

# Script name: ex1.sh

echo "Enter some text: \c"
read INPUT

for var in $INPUT
do
        echo "var contains: $var"
done


$ ./ex1.sh
Enter some text: I like the Korn shell.
var contains: I
var contains: like
var contains: the
var contains: Korn
var contains: shell.
```

# Command Substitution

Korn shell:

```
for var in $(cmd_sub)
```

Bourne shell

```
for var in `cmd_sub`
```

# Command Substitution

```
$ cat fruit1
apple
orange
banana
peach
kiwi

$ cat ex3.ksh
#!/bin/ksh

# Script name: ex3.ksh

for var in $(cat fruit)
do
        print  "$var"
done

$ ./ex3.ksh
apple
orange
banana
peach
kiwi
```

# File Names in Command Substitution

```
$ cat ex7.ksh
#!/bin/ksh

# Script name: ex7.ksh

for var in $(ls /etc/p*)
do
        print  "var contains: $var"
done
$ cat for.ksh
#!/bin/ksh

# Script name: for.ksh

print "Subdirectories in $(pwd):"

for fname in $(ls)              # using command substitution to
do                              # generate an argument list
        if [[ -d $fname ]]
        then
                print $fname
        fi
done

$ cd /usr/share
$ /SA245_LF/mod9/examples/for.ksh
Subdirectories in /usr/share:
lib
man
src

$ cd

$ ./for.ksh
Subdirectories in /home/user200:
filesub
functions
```

# File-Name Substitution

```
for var in file_list
```

```
ls /etc/p*
/etc/passwd /etc/profile /etc/prtvtoc
...
for var in /etc/p*
```

# Mid-Module Exercise: Loops

- Objectives
- Tasks
- Discussion
- Solutions

# The `while` Loop

- As long as the *`command_control`* succeeds, the loop body continues to execute

```
while command_control
do
    statement1
    ...
    statementn
done
```

# while Loop Syntax

While the contents of $var are equal to "value," the loop continues.

```
while [ "$var" = "value" ]
while [[ "$var" == "value" ]]
```

While the value of $num is less than or equal to 10, the loop continues.

```
while [ $num -le 10 ]
while (( num <= 10 ))
```

```
$ cat whiletest.sh
#!/bin/sh

# Script name: whiletest.sh

num=5

while [ $num -le 10 ]
do
        echo $num
        num=‘expr $num + 1‘
done
```

```
$ cat whiletest.ksh
#!/bin/ksh

# Script name: whiletest.ksh

num=5

while (( num <= 10 ))
do
        echo $num
        (( num = num + 1 ))     # let num=num+1
done
```

# Example: `while` Loop

```
$ cat while.ksh
#!/bin/ksh


# Script name: while.ksh

num=1

while (( num < 6 ))
do
        print "The value of num is: $num"
        (( num = num + 1 ))             # let num=num+1
done

print "Done."

$
$ ./while.ksh
Value of n is: 1
Value of n is: 2
Value of n is: 3
Value of n is: 4
Value of n is: 5
Done.
$
```

# Keyboard Input

```
$ cat readinput.ksh
#!/bin/ksh

# Script name: readinput.ksh

print -n "Enter a string: "

while read var
do
        print "Keyboard input is: $var"
        print -n "\nEnter a string: "
done

print "End of input."


$ ./readinput.ksh
Enter a string: OK
Keyboard input is: OK

Enter a string: This is fun
Keyboard input is: This is fun

Enter a string: I'm finished.
Keyboard input is: I'm finished.

Enter a string: End of input.
$
```

# Redirecting Input for a `while` Loop

```
done < phonelist

$ cat phonelist2
Claude Rains:214-555-5107
Agnes Moorehead:710-555-6538
Rosalind Russel:710-555-0482
Loretta Young:409-555-9327
James Mason:212-555-2189
$
$ cat internal_redir.ksh
#!/bin/ksh

# Script name: internal_redir.ksh

# set the Internal Field Separator to a colon
IFS=:

while read name number
do
        print "The phone number for $name is $number"
done < phonelist

$ ./internal_redir.ksh
The phone number for Claude Rains is 214-555-5107
The phone number for Agnes Moorehead is 710-555-6538
The phone number for Rosalind Russel is 710-555-0482
The phone number for Loretta Young is 409-555-9327
The phone number for James Mason is 212-555-2189
```

# The `until` Loop

```
until control_command
do
   statement1
   ...
   statementn
done
```

```
$ cat until.ksh
#!/bin/ksh

# Script name: until.ksh

num=1

until (( num == 6 ))
do
        print "The value of num is: $num"
        (( num = num + 1 ))
done

print "Done."
```

```
$ ./until.ksh
The value of num is: 1
The value of num is: 2
The value of num is: 3
The value of num is: 4
The value of num is: 5
Done.
```

# The break Statement

```
$ cat break.ksh
#!/bin/ksh

# Script name: break.ksh

typeset -i  num=0

while true
do
        print -n "Enter any number (0 to exit): "
        read num junk

        if (( num == 0 ))
        then
                break
        else
                print "Square of $num is $(( num * num )). \n"
        fi
done

print "script has ended"

$ ./break.ksh
Enter any number (0 to exit): 5
Square of 5 is 25.

Enter any number (0 to exit): -5
Square of -5 is 25.

Enter any number (0 to exit): 259
Square of 259 is 67081.

Enter any number (0 to exit): 0
script has ended
$
```

# The `continue` Statement

- Use `continue` to return to the top of the loop

# Example: continue Statement

```
$ cat continue.ksh
#!/bin/ksh

# Script name: continue.ksh

typeset -l new

for file in *
do
        print "Working on file $file..."

        if [[ $file != *[A-Z]* ]]
        then
                continue
        fi

        orig=$file
        new=$file
        mv $orig $new
        print "New file name for $orig is $new."
done

print "Done."
```

# Example: `continue` Statement

```
$ cd test.dir
$ ls
Als             a               sOrt.dAtA       slAlk
Data.File       recreate_names  scR1            teXtfile

$ ../continue.ksh
Working on file Als...
New file name for Als is als.
Working on file Data.File...
New file name for Data.File is data.file.
Working on file a...
Working on file recreate_names...
Working on file sOrt.dAtA...
New file name for sOrt.dAtA is sort.data.
Working on file scR1...
New file name for scR1 is scr1.
Working on file slAlk...
New file name for slAlk is slalk.
Working on file teXtfile...
New file name for teXtfile is textfile.
Done.

$ ls
a               data.file       scr1            sort.data
als             recreate_names  slalk           textfile
```

# The Korn Shell `select` Loop

```
select var in list
do
    statement1
    ...
    statementn
done
```

# The PS3 Reserved Variable

- Default value #?
- Can be set to any string value

# Example of a Menu

```
$ cat menu.ksh
#!/bin/ksh

# Script name: menu.ksh


PS3="Enter the number for your fruit choice: "

select fruit in apple orange banana peach pear
do
        case $fruit in
        apple)
                print "An apple has 80 calories."
                ;;

        orange)
                print "An orange has 65 calories."
                ;;

        banana)
                print "A banana has 100 calories."
                ;;

        peach)
                print "A peach has 38 calories."
                ;;

        pear)
                print "A pear has 100 calories."
                ;;

        *)
                print "Please try again. Use '1'-'5'"
                ;;
esac

done
```

# Example of a Menu

```
$ ./menu.ksh
1) apple
2) orange
3) banana
4) peach
5) pear
Enter the number for your fruit choice: 3
A banana has 100 calories.
Enter the number for your fruit choice: 7
Please try again. Use '1'-'5'
Enter the number for your fruit choice: apple
Please try again. Use '1'-'5'
Enter the number for your fruit choice: 1
An apple has 80 calories.
Enter the number for your fruit choice: ^d
$
```

# Exiting the Menu Loop

```
$ cat menu1.ksh
#!/bin/ksh

# Script name: menu.ksh

PS3="Enter the number for your fruit choice: "

select fruit in apple orange banana peach pear "Quit Menu"
do
        case $fruit in
                apple)
                        print "An apple has 80 calories."
                        ;;

                orange)
                        print "An orange has 65 calories."
                        ;;

                banana)
                        print "A banana has 100 calories."
                        ;;

                peach)
                        print "A peach has 38 calories."
                        ;;

                pear)
                        print "A pear has 100 calories."
                        ;;

                "Quit Menu")
                        break
                        ;;

                *)
                        print "You did not enter a correct choice."
                        ;;
        esac
done
```

# Exiting the Menu Loop

```
$ ./menu1.ksh
1) apple
2) orange
3) banana
4) peach
5) pear
6) Quit Menu
Enter the number for your fruit choice: 3
A banana has 100 calories.
Enter the number for your fruit choice: 6
$
```

# Submenus

- The `select` loop can contain another `select` loop
- Include the `break` statement in the action to exit the submenu loop
- Reset the `PS3` variable to the prompt needed when moving from one menu to another

# Example: Using Submenus

```
$ cat submenu.ksh
#!/bin/ksh
main_prompt="Main Menu: What would you like to order? "
dessert_menu="Enter number for dessert choice: "
PS3=$main_prompt

select order in "broasted chicken" "prime rib" "stuffed lobster"
dessert "Order Completed"
do
   case $order in
      "broasted chicken") print 'Broasted chicken with baked potato,
rolls, and salad is $14.95.';;
      "prime rib") print 'Prime rib with baked potato, rolls, and fresh
vegetable is $17.95.';;
      "stuffed lobster") print 'Stuffed lobster with rice pilaf, rolls,
and salad is $15.95.';;
      dessert)
         PS3=$dessert_menu
         select dessert in "apple pie" "sherbet" "fudge cake" "carrot
cake"
         do
            case $dessert in
               "apple pie") print 'Fresh baked apple pie is $2.95.'
                  break;;
               "sherbet") print 'Orange sherbet is $1.25.'
                  break;;
               "fudge cake") print 'Triple layer fudge cake is $3.95.'
                  break;;
               "carrot cake") print 'Carrot cake is $2.95.'
                  break;;
               *) print 'Not a dessert choice.';;
            esac
         done
         PS3=$main_prompt;;
      "Order Completed") break;;
      *) print 'Not a main entree choice.';;
   esac
done
print 'Enjoy your meal.'
```

# Sample Run of submenu.ksh Script

```
$  ./submenu.ksh
1) broasted chicken
2) prime rib
3) stuffed lobster
4) dessert
5) Order Completed
Main Menu: What would you like to order? 3
Stuffed lobster with rice pilaf, rolls, and salad is $15.95.
Main Menu: What would you like to order? 4
1) apple pie
2) sherbet
3) fudge cake
4) carrot cake
Enter number for dessert choice: 3
Triple layer fudge cake is $3.95.
Main Menu: What would you like to order? 5
Enjoy your meal.
$
```

# The `for` and `select` Statements Revisited

```
$ cat pospara.ksh
#!/bin/ksh

# Script name: pospara.ksh

set uno duo tres        # resets the value of the positional parameters
print "Executing script $0\n"

print "One, two, three in Latin is:"
for x                   # defaults to "in $*"
do
        print $x
done

$ ./pospara.ksh
Executing script pos_par1.scr

One, two, three in Latin is:
uno
duo
tres
```

# The `shift` Statement

- Process the arguments to the script in a `while` loop
- Use `shift` to process the next argument

# Example: Using the `shift` Statement

```
$ cat shift.ksh
#!/bin/ksh

# Script name: shift.ksh

USAGE="usage: $0 arg1 arg2 ... argN"

if (( $# == 0 ))
then
        print $USAGE
        exit 1
fi

print "The arguments to the script are:"
while (($#))
do
        print $1
        shift
done

print 'The value of $* is now:' $*


$ ./shift.ksh one two three four
The arguments to the script are:
one
two
three
four
The value of $* is now:
$
```

# Exercise: Loops

- Objectives
- Tasks
- Discussion
- Solutions

# Check Your Progress

- Write scripts that use `for`, `while`, and `until` loops
- Write a script using the `select` statement
- Describe when to use loops within a script
- Generate argument lists using command, variable, and file-name substitution

# Module 10

# Advanced Variables, Parameters, and Argument Lists

# Objectives

- Declare strings, integers, and array variables
- Manipulate string variables
- Change the values of the positional parameters using the `set` statement within a script
- Use Korn shell arrays
- Set default values for parameters
- Use the Korn shell built-in statements `let`, `print`, `set`, and `typeset`

# Types of Variables

- Bourne and Korn shells
  - Strings
- Korn shell
  - Integers
  - Arrays
  - Constants

# The Korn Shell `typeset` Command

| Syntax | Description |
|---|---|
| `${#var}` | Returns the length of string *var*. |
| `typeset -u var` | Converts *var* to all uppercase characters. |
| `typeset -l var` | Converts *var* to all lowercase characters. |
| `typeset -LZ var` | Strips leading zeros from the string *var*. |
| `typeset -Lnum var` | Left-justifies *var* within the field width specified by *num*. |
| `typeset -Rnum var` | Right-justifies *var* within the field width specified by *num*. |
| `typeset -i var` | *var* can only contain integer values. |
| `typeset -r var` | *var* is read-only. The value in *var* cannot be changed by subsequent assignment. |

# Example: String Manipulation

```
$ cat strman1.ksh
#!/bin/ksh

# Script name: strman1.ksh

typeset -R8 word="happy"

typeset -L5 word1="depressed"

print "123456789"
print "$word"
print

print "123456789"
print "$word1"
$
$ ./strman1.ksh
123456789
   happy

123456789
depre
```

# Example: String Manipulation

```
$ cat strman2.ksh
#!/bin/ksh

# Script name: strman2.ksh

typeset -R8 word="happy"

typeset -L5 word1="depressed"

print "123456789"
print $word
print

print "123456789"
print $word1

$ ./strman2.ksh
123456789
happy

123456789
depre
```

# Example: Using typeset

```
$ cat strman3.ksh
#!/bin/ksh

string1="manipulation"
print "Length of string1 is ${#string1} characters\n"

string2="CaSes"
print "string2 is $string2"
typeset -u string2
print "string2 in upper case: $string2"
typeset -l string2
print "string2 in lower case: $string2\n"

typeset -L7 ljust
ljust="hi there"

typeset -R5 rjust
rjust="farewell"

print "                123456789"
print "Value of ljust: $ljust"
print "Value of rjust: $rjust"

lzero="00034;lsl"
print "Value of lzero: $lzero"
typeset -LZ lzero
print "New value of lzero: $lzero"
$
$ ./strman3.ksh
Length of string1 is 12 characters
string2 is CaSes
string2 in upper case: CASES
string2 in lower case: cases
          123456789
Value of ljust: hi ther
Value of rjust: ewell
Value of lzero: 00034;lsl
New value of lzero: 34;lsl
```

# Declaring an Integer Variable

Example 1:

```
$ typeset -i num
$ num=5
$ print $num
5
$
```

Example 2:

```
$ typeset -i num
$ num=25.34
$ print $num
25
$
```

Example 3:

```
$ typeset -i num  # base 10 integer
$ num=27
$ print $num
27
$ typeset -i8 num # change to base 8
$ print $num
8#33
$

$ num=two
/usr/bin/ksh: two: bad number
$ print $num
8#33
```

# Arithmetic Operations on Korn Shell Variables

| Operator | Operations | Example | Result |
|---|---|---|---|
| + | Addition | ((x = 24 + 25)) | 49 |
| - | Subtraction | ((x = 100 - 25)) | 75 |
| * | Multiplication | ((x = 4 * 5)) | 20 |
| / | Division | ((x = 10 / 3)) | 3 |
| % | Modulo (remainder) | ((x = 10 % 3)) | 1 |
| # | Base | 2#1101010 or 16#6A | 10#106 |
| << | Shift bits left | ((x = 2#11 << 3)) | 2#11000 |
| >> | Shift bits right | ((x = 2#1001 >> 2)) | 2#10 |
| & | Bit-wise AND | ((x = 2#101 & 2#110)) | 2#100 |
| \| | Bit-wise OR | ((x = 2#101 \| 2#110)) | 2#111 |
| ^ | Bit-wise exclusive OR | (((x = 2#101 ^ 2#110)) | 2#11 |

# Bit-wise Operations

- The # operator designates the base of the value
- The << operator performs a binary shift left
- The >> operator performs a binary shift right
- The & operator ANDs two binary numbers together
- The | operator ORs two binary numbers together
- The ^ operator exclusively ORs two binary numbers together

# Creating Bourne Shell Constants

```
readonly var[=value]

$ sh
$ var=constant
$ readonly var
$ unset var
var: is read only

$ var=new_value
var: is read only
```

# Creating Korn Shell Constants

```
typeset -r var[=value]
readonly var[=value]

$ ksh
$ typeset -r cvar=constant
$ unset cvar
ksh: cvar: is read only

$ cvar=new_value
ksh: cvar: is read only
```

# Removing Portions of a String

| Syntax | Description |
|---|---|
| ${*str_var%pattern*} | Removes the smallest right-most substring of string *str_var* that matches *pattern*. |
| ${*str_var%%pattern*} | Removes the largest right most-substring of string *str_var* that matches *pattern*. |
| ${*str_var#pattern*} | Removes the smallest left most-substring of string *str_var* that matches *pattern*. |
| ${*str_var##pattern*} | Removes the largest left most-substring of string *str_var* that matches *pattern*. |

# Examples: Removing Portions of a String

```
stringx=/usr/bin/local/bin

print ${stringx%/bin}
/usr/bin/local

print ${stringx%/bin*}
/usr/bin/local

print ${stringx%%/bin}
/usr/bin/local

print ${stringx%%/bin*}
/usr

print ${stringx#/usr/bin}
/local/bin

print ${stringx#*/bin}
/local/bin

print ${stringx##/usr/bin}
/local/bin

print ${stringx##*/bin}


print ${stringx##*/}
bin
```

# Korn Shell Arrays

- Contain more than one value
- Are created when you use them
- By default, each value is a string

# Examples: Array

To create an array of three strings:

```
arr[0]=big
arr[1]=small
arr[2]="medium sized"
```

To create an array of three strings using the `set` command:

```
set -A arr big small "medium sized"
```

To create an array of five integers:

```
integer num
num[0]=0
num[1]=100
num[2]=200
num[3]=300
num[4]=400
```

To print the number of array elements in array *num*:

```
$ print ${#num[*]}
5
```

To print the values of all array elements in array *arr*:

```
$ print ${arr[*]}
big small medium sized
```

To unset the array *arr*:

```
unset arr
```

# Command-Line Arguments

| Positional Parameter Name | Description |
|---|---|
| `$0` | The name of the script |
| `$1` | The first argument to the script |
| `$2` | The second argument to the script |
| `$9` | The ninth argument to the script |
| `${10}, ${11},${`*n*`}` | The tenth and up argument to the script (Korn shell only) |
| `$#` | The number of arguments to the script |
| `$@` | A list of all arguments to the script |
| `$*` | A list of all arguments to the script |
| `${#`*N*`}` | The length of the value of positional parameter *N* (Korn shell only) |

# Using the `shift` Statement

```
$ cat argtest.sh
#!/bin/sh

# Script name: argtest.sh

echo '$#: ' $#
echo '$@: ' $@
echo '$*: ' $*
echo
echo '$1 $2 $9 $10 are: ' $1 $2 $9 $10
echo

shift
echo '$#: ' $#
echo '$@: ' $@
echo '$*: ' $*
echo
echo '$1 $2 $9 are: ' $1 $2 $9

shift 2
echo '$#: ' $#
echo '$@: ' $@
echo '$*: ' $*
echo
echo '$1 $2 $9 are: ' $1 $2 $9

echo '${10}: ' ${10}
```

# Using the `shift` Statement

```
$ ./argtest.sh a b c d e f g h i j k l m n
$#:  14
$@:  a b c d e f g h i j k l m n
$*:  a b c d e f g h i j k l m n

$1 $2 $9 $10 are:  a b i a0

$#:  13
$@:  b c d e f g h i j k l m n
$*:  b c d e f g h i j k l m n

$1 $2 $9 are:  b c j
$#:  11
$@:  d e f g h i j k l m n
$*:  d e f g h i j k l m n

$1 $2 $9 are:  d e l
./argtest.sh: bad substitution
```

# Using Positional Parameters

- Scripts should check the number of positional parameters
- You cannot use a positional parameter on the left of an assignment statement

# Assigning Positional Parameter Values Using set

```
set value1 value2 ... valueN

set $(cal)
set $var1

set -s

set --
```

```
$ set $(cal)
$ echo $1
September
$ echo $#
39
$
```

# Example: Using set

```
$ cat pospara2.ksh
#!/bin/ksh

# Script name: pospara2.ksh

print "Executing script $0 \n"
print "$1 $2 $3"

set uno duo tres
print "One two three in Latin is:"
print "$1"
print "$2"
print "$3 \n"

textline="name phone address birthdate salary"
set $textline
print "$*"
print ’At this time $1 =’ $1 ’and $4 =’ $4 "\n"

set -s
print "$* \n"

set --
print "$0 $*"

$ ./pospars.ksh a b c
Executing script ./pospara2.sc

a b c
One two three in Latin is:
uno
duo
tres

name phone address birthdate salary
At this time $1 = name and $4 = birthdate

address birthdate name phone salary
```

# The Values of "$@" and "$*"

- The values of $@ and $* are identical, but the values of "$@" and "$*" are different.
- "$@" expands to "$1" "$2" "$3" ... "$n"; that is, *n* separate strings.
- "$*" expands to "$1x$2x$3x...$n", where x is the first character in the set of delimiters for the IFS variable, which means that "$*" is one long string.

# Exercise: Advanced Variables, Parameters and Argument Lists

- Objectives
- Tasks
- Discussion
- Solutions

# Check Your Progress

- Declare strings, integers, and array variables
- Manipulate string variables
- Change the values of the positional parameters using the `set` statement within a script
- Use Korn shell arrays
- Set default values for parameters
- Use the Korn shell built-in statements `let`, `print`, `set`, and `typeset`

# Module 11

## Functions

# Objectives

- Create user-defined functions in a shell script
- Create, invoke, and display functions from the command line
- Pass arguments into a function
- Call functions from special (function) files that are saved in one or more function directories
- Describe where functions are available for use

# Functions in the Shell

- Bourne shell syntax:

```
function_name ()
{
    block of command lines
}
```

- Korn shell syntax:

```
function function_name
{
    block of command lines
}
```

# Positional Parameters

```
$ cat funparas.ksh
#!/bin/ksh

# Script name: funparas.ksh

function hello
{
        print '$1 the function is: ' $1
}


print 'Input passed and stored in $1 is: ' $1

hello John              # execute the function hello

print
print 'After the function $1 is still ' $1

$ ./funparas.ksh Susan
Input passed and stored in $1 is:  Susan
$1 the function is:  John

After the function $1 is still  Susan
```

# Positional Parameters

```
$ cat /.kshrc

rgrep ()                # Bourne shell syntax
{
        find $2 -type file -exec grep $1 {} \; | more
}



function rcgrep         # Korn shell syntax
{
        grep $1 /etc/init.d/* |more
}



function killit         # Korn shell syntax
{
        pkill -u $1
        print -n "Had to kill process for user: $1 "
        print "on $(date +%D) at $(date +%T)"

        # The previous print statement may be appended to a log file.
}

$ killit annette
Had to kill process for user: annette on 05/23/00 at 21:38:32

$ rcgrep sed
/etc/init.d/autoinstall:# are available, then the default profiles in
are used.
/etc/init.d/dtlogin:# This version of the dtlogin.rc script can be
used on the Solaris(TM)
/etc/init.d/inetinit:# tcp used when it loaded.

<Output truncated>
```

# Return Values

- Pass a value from the function back to the shell
- Use the `return` command
- `return` terminates the function

# `typeset` and `unset`

- `typeset -f` Lists the known functions and their definitions
- `typeset +f` Lists the known function names
- `function` Is an alias for `typeset -f`
- `unset -f` *`name`* Unsets the value of the function

# Function Files

- The function file:
  - Does not need to be executable
  - Can be autoloaded
  - Can be loaded into the current shell environment

# Autoloading Korn Shell Functions With the FPATH Variable

```
$ FPATH=$HOME/function_dir ; export FPATH
```

- The directories listed in the FPATH variable should only contain function files
- The function is treated the same as a built-in command
- Define a function at the start of a script
- Declare the FPATH variable before any command lines attempt to invoke the command

# Exercise: Functions

- Objectives
- Tasks
- Discussion
- Solutions

# Check Your Progress

- Create user-defined functions in a shell script
- Create, invoke, and display functions from the command line
- Pass arguments into a function
- Call functions from special (function) files that are saved in one or more function directories
- Describe where functions are available for use

# Module 12

## Traps

# Objectives

- Describe how the `trap` statement works
- Include `trap` statements in a script
- Use the `trap` statement to catch signals and handle errors

# Shell Signal Values

| | | | | |
|---|---|---|---|---|
| EXIT | HUP | INT | QUIT | ILL |
| TRAP | ABRT | EMT | FPE | KILL |
| BUS | SEGV | SYS | PIPE | ALRM |
| TERM | USR1 | USR2 | CLD | PWR |
| WINCH | URG | POLL | STOP | TSTP |
| CONT | TTIN | TTOU | VTALRM | PROF |
| XCPU | XFSZ | WAITING | LWP | FREEZE |
| THAW | CANCEL | LOST | RTMIN | RTMIN+1 |
| RTMIN+2 | RTMIN+3 | RTMAX-3 | RTMAX-2 | RTMAX-1 |
| RTMAX | | | | |

# Catching Signals With `trap`

The syntax for using the trap statement is:
trap '*action*' signal

```
trap ’echo "Control-C not available"’ INT

trap ’echo "Control-C not available"
echo "Core dumps not allowed"
sleep 1
continue’ INT QUIT

kill -9 script_PID
kill -KILL script_PID

trap - signal
```

# Example: Using `trap`

```
$ cat trapsig.ksh
#!/bin/ksh

# Script name: trapsig.ksh

trap 'print "Control-C cannot terminate this script."' INT
trap 'print "Control- cannot terminate this script."' QUIT
trap 'print "Control-Z cannot terminate this script."' TSTP


print "Enter any string (type 'dough' to exit)."
while (( 1 ))
do
        print -n "Rolling..."
        read string

        if [[ "$string" = "dough" ]]
        then
                break
        fi
done

print "Exiting normally"

$ ./trapsig.ksh
Enter any string (type 'dough' to exit).
Rolling...
Rolling...d
Rolling...s
Rolling...Rolling...Rolling...Rolling...Rolling...4
Rolling...^c
Control-C cannot terminate this script.
Rolling...^\
Control- cannot terminate this script.
Rolling...^z
Control-Z cannot terminate this script.
Rolling...dough
Exiting normally
$
```

# Catching User Errors With `trap`

```
$  cat traperr1.ksh
#!/bin/ksh

# Script name: traperr1.ksh

integer num=2

while (( 1 ))
do
        read num?"Enter any number ( -1 to exit ): "

        if (( num == -1 ))
        then
                break
        else
                print "Square of $num is $(( num * num )). \n"
        fi
done

print "Exiting normally"

$ ./traperr1.ksh
Enter any number (-1 to exit): r
trap_err1[9]: r: bad number
Square of 2 is 4.

Enter any number (-1 to exit): -1
$
```

# Example: Using `trap` With ERR

```
$ cat trapsig2.ksh
#!/bin/ksh
# Script name: trapsig2.ksh

integer num

exec 2> /dev/null
trap 'print "You did not enter an integer."' ERR

while (( 1 ))
do
        print -n "Enter any number ( -1 to exit ): "
        read num

        status=$?

        if (( num == -1 ))
        then
                break
        elif (( $status == 0 ))
        then
                print "Square of $num is $(( num * num )). \n"
        fi
done
print "Exiting normally"

$ ./trapsig2.ksh
Enter any number ( -1 to exit ): 3
Square of 3 is 9.

Enter any number ( -1 to exit ): r
You did not enter an integer.
Enter any number ( -1 to exit ): 8
Square of 8 is 64.

Enter any number ( -1 to exit ): -1
Exiting normally
```

# When to Declare a `trap`

- To trap a signal any time during execution, define the trap at the start of the script.
- To trap a signal only when certain command lines are executed, turn on the trap before the lines, and then turn off the trap after the lines.
- If a loop is being used, a trap can include the `continue` command to make the loop start again from its beginning.
- You can also trap the `EXIT` signal so that certain commands are executed only when the shell script is being terminated with no errors.

# Exercise: Traps

- Objectives
- Tasks
- Discussion
- Solutions

# Check Your Progress

- Describe how the `trap` statement works
- Include `trap` statements in a script
- Use the `trap` statement to catch signals and handle errors

# Course Contents