information technology & management

viabiliti

INTRO TO OPEN SOURCE OPERATING SYSTEMS

ILLINOIS INSTITUTE OF TECHNOLOGY

ITMO456

Working with the BASH Shell

Sean Hughes-Durkin

ITMO/IT-O 456 Fall 2017

Information Technology & Management Programs

School of Applied Technology

Objectives

At the end of this lesson, the student should be able to:

- Enter, edit, and reissue shell commands
- Use command completion
- Redirect input and output of a command
- Run commands in the background from a shell
  - Manage commands running in the background
- Identify and manipulate common shell environment variables

Objectives

At the end of this lesson, the student should be able to:

- Create and export new shell variables
- Edit environment files to create variables upon shell startup
- Describe the purpose and nature of shell scripts
- Create and execute basic shell scripts
- Effectively use common decision and loop constructs in shell scripts

Why Use the Shell?

- Universal access
  - You can use any Linux or other UNIX-like system from any terminal access
- Faster access
  - Power users can do many actions much faster from the command line than using a GUI
- More efficient access
  - Combine multiple commands in a shell script

Shell Prompt

- **#** - root or superuser
  - **[root@itm456 ~]#**
- **$** - regular user
  - **[sean@itm456 ~]$**
- You can change the prompt to display any characters you like
- Controlled by the **PS1** variable
  - https://wiki.archlinux.org/index.php/Color_Bash_Prompt
  - http://misc.flogisoft.com/bash/tip_colors_and_formatting

PS1 Special Characters

```
\a      an ASCII bell character (07)
\d      the date in "Weekday Month Date" format (e.g., "Tue May 26")
\D{format}
        the  format  is  passed  to  strftime(3) and the result is inserted into the
        prompt string; an empty format results in a locale-specific time representa-
        tion.  The braces are required
\e      an ASCII escape character (033)
\h      the hostname up to the first `.'
\H      the hostname
\j      the number of jobs currently managed by the shell
\l      the basename of the shell's terminal device name
\n      newline
\r      carriage return
\s      the  name  of the shell, the basename of $0 (the portion following the final
        slash)
\t      the current time in 24-hour HH:MM:SS format
\T      the current time in 12-hour HH:MM:SS format
\@      the current time in 12-hour am/pm format
\A      the current time in 24-hour HH:MM format
\u      the username of the current user
\v      the version of bash (e.g., 2.00)
\V      the release of bash, version + patch level (e.g., 2.00.0)
\w      the current working directory, with $HOME abbreviated with a tilde (uses the
        value of the PROMPT_DIRTRIM variable)
\W      the basename of the current working directory, with $HOME abbreviated with a
        tilde
\!      the history number of this command
\#      the command number of this command
\$      if the effective UID is 0, a #, otherwise a $
\nnn    the character corresponding to the octal number nnn
\\      a backslash
\[      begin a sequence of non-printing characters, which could be used to embed  a
        terminal control sequence into the prompt
\]      end a sequence of non-printing characters
```

Customizing the **PS1** variable

```
[root@itmo456 ~]# echo $PS1
[\u@\h \W]\$
[root@itmo456 ~]#
[root@itmo456 ~]# export PS1='[\u@\h \d \W]\$ '
[root@itmo456 Sun Oct 04 ~]#
[root@itmo456 Sun Oct 04 ~]#
[root@itmo456 Sun Oct 04 ~]# export PS1='[\u@\h \D{%m/%d/%Y} \W]\$ '
[root@itmo456 10/04/2015 ~]#
[root@itmo456 10/04/2015 ~]#
[root@itmo456 10/04/2015 ~]# export PS1='[\u@\h \D{%m/%d/%Y} \t \W]\$ '
[root@itmo456 10/04/2015 17:00:01 ~]#
[root@itmo456 10/04/2015 17:00:01 ~]#
[root@itmo456 10/04/2015 17:00:02 ~]# export PS1='[\u@\h \D{%m/%d/%Y} \@ \W]\$ '
[root@itmo456 10/04/2015 05:00 PM ~]#
[root@itmo456 10/04/2015 05:00 PM ~]#
[root@itmo456 10/04/2015 05:00 PM ~]# export PS1='[\u@\h \w]\$ '
[root@itmo456 ~]# cd /etc/sysconfig/network-scripts/
[root@itmo456 /etc/sysconfig/network-scripts]#
[root@itmo456 /etc/sysconfig/network-scripts]#
[root@itmo456 /etc/sysconfig/network-scripts]# export PS1='[\[\e[0;31m\]\u@\[\e[0;35m\]\h\[\e[m\] \[
\e[0;34m\]\W\[\e[m\] ]\[\e[0;33m\]\$ \[\e[m\]'
[root@itmo456 network-scripts ]#
[root@itmo456 network-scripts ]#
[root@itmo456 network-scripts ]# cd
[root@itmo456 ~ ]#
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# pwd
/root
[root@itmo456 ~ ]#
```

The bash Shell

- **bash** (Bourne Again Shell) default in most Linux distros
  - Based on Bourne shell (**sh**) created @ Bell Labs
- Can run other shells
  - Just type shell name at the command line
  - List of installed shells located **/etc/shells**

Other Shells

- **ASH shell** (ash) - A bourne shell clone found in Linux
- **Bourne shell** (sh) - The first UNIX shell, still widely used today; bash is an enhanced version of the Bourne shell
- **C shell** (csh) - Uses a "C like" syntax and has borrowed many features from the Bourne shell but uses a different set of internal shell commands

Slide based on Lotgering, Guy; *Universal Command Guide for Operating Systems*, UCG Media 2001

Other Shells

- **Korn shell** (ksh) - Uses same syntax as the bourne shell and includes user friendly features of the C shell
- **TC shell** (tcsh) - Enhanced version of the C shell; 100% C shell compatible
- **Z shell** (zsh) - Enhanced version of ksh with many features found in bash

Slide based on Lotgering, Guy; *Universal Command Guide for Operating Systems*, UCG Media 2001

Using the Shell prompt

- Command completion
  - Pressing tab attempts to complete a partially typed command
  - **Environment variable:** If text begins with a dollar sign ($), shell completes with an environment variable from the current shell
  - **Username:** If text begins with a tilde (~), shell completes with a username

Using the Shell prompt

- Command completion
  - **Command, alias, or function:** If text begins with regular characters, shell tries to complete with a command, alias, or function name

Using the Shell prompt

Keystroke	Full Name	Meaning
Ctrl+F	Character forward	Go forward one character.
Ctrl+B	Character backward	Go backward one character.
Alt+F	Word forward	Go forward one word.
Alt+B	Word backward	Go backward one word.
Ctrl+A	Beginning of line	Go to the beginning of the current line.
Ctrl+E	End of line	Go to the end of the line.
Ctrl+L	Clear screen	Clear screen & leave line at the top of the screen.

Keystrokes for Navigating Command Lines

From Negus, Christopher, *Linux Bible 2010 Edition*; Wiley Publishing, Inc. 2010

Using the Shell prompt

Keystroke	Full Name	Meaning
Ctrl+D/del	Delete current	Delete the current character.
Backspace	Delete previous	Delete the previous character.
Ctrl+T	Transpose character	Switch positions of current & previous characters.
Alt+T	Transpose words	Switch positions of current and previous words.
Alt+U	Uppercase word	Change the current word to uppercase.
Alt+L	Lowercase word	Change the current word to lowercase.
Alt+C	Capitalize word	Change the current word to an initial capital.
Ctrl+V	Insert special character	Add a special character. For example, to add a Tab character, press Ctrl+V+Tab.

Keystrokes for Editing Command Lines

From Negus, Christopher, *Linux Bible 2010 Edition*; Wiley Publishing, Inc. 2010

Using the Shell prompt

Keystroke	Full Name	Meaning
Ctrl+K	Cut end of line	Cut text to the end of the line.
Ctrl+U	Cut beginning of line	Cut text to the beginning of the line.
Ctrl+W	Cut previous word	Cut the word located behind the cursor.
Alt+D	Cut next word	Cut the word following the cursor.
Ctrl+Y	Paste recent text	Paste most recently cut text.
Alt+Y	Paste earlier text	Rotate back to previously cut text and paste it.
Ctrl+C	Delete whole line	Delete the entire line.
Ctrl+K	Cut end of line	Cut text to the end of the line.

Keystrokes for Cutting and Pasting Text in Command Lines

From Negus, Christopher, *Linux Bible 2010 Edition*; Wiley Publishing, Inc. 2010

Using the Shell prompt

- Command line recall
  - Pressing up arrows cycles through recent previous commands
  - Once up is pressed, pressing down cycles back down through recent previous commands
  - View history with **`history`** command
    - Run command in history list with list line number preceded by **!**
    - Run previous command with **!!**

Using the Shell prompt

```
[root@itmo456 ~ ]# history
    1  ll /boot
    2  history
[root@itmo456 ~ ]# !1
ll /boot
total 88990
-rw-r--r--. 1 root root   131847 Dec  5  2013 config-3.11.10-301.fc20.x86_64
-rw-r--r--. 1 root root   152749 May 12 12:14 config-3.19.8-100.fc20.x86_64
drwxr-xr-x. 4 root root     1024 Dec 11  2013 efi
-rw-r--r--. 1 root root   192916 Oct 21  2014 elf-memtest86+-5.01
drwxr-xr-x. 2 root root     1024 Dec 11  2013 extlinux
drwxr-xr-x. 6 root root     1024 Sep 20 16:12 grub2
-rw-------. 1 root root 38823288 Sep 20 11:34 initramfs-0-rescue-ac106f57b96b43478e83ed5079be578e.im
g
-rw-------. 1 root root 11798360 Sep 20 11:34 initramfs-3.11.10-301.fc20.x86_64.img
-rw-------. 1 root root 17402406 Sep 20 16:12 initramfs-3.19.8-100.fc20.x86_64.img
-rw-r--r--. 1 root root   585107 Dec 11  2013 initrd-plymouth.img
drwx------. 2 root root    12288 Sep 20 11:30 lost+found
-rw-r--r--. 1 root root   191240 Oct 21  2014 memtest86+-5.01
-rw-------. 1 root root  2686629 Dec  5  2013 System.map-3.11.10-301.fc20.x86_64
-rw-------. 1 root root  3028604 May 12 12:14 System.map-3.19.8-100.fc20.x86_64
-rwxr-xr-x. 1 root root  5139320 Sep 20 11:34 vmlinuz-0-rescue-ac106f57b96b43478e83ed5079be578e
-rwxr-xr-x. 1 root root  5139320 Dec  5  2013 vmlinuz-3.11.10-301.fc20.x86_64
-rwxr-xr-x. 1 root root  5816152 May 12 12:14 vmlinuz-3.19.8-100.fc20.x86_64
```

Using the Shell prompt

Keystroke	Full Name	Meaning
Arrow Keys (↓ and ↑)	Step	Press the up and down arrow keys to step through each command line in your history list to arrive at the one you want. (Ctrl+P and Ctrl+N do the same functions, respectively.)
Ctrl+R	Reverse Incremental Search	After you press these keys, you enter a search string to do a reverse search. As you type the string, a matching command line appears that you can run or edit.
Ctrl+P	Reverse Search	Previous command in history (i.e. walk back through the command history)
Ctrl+N	Forward Search	Next command in history (i.e. walk forward through the command history)

Keystrokes for Using Command History

From Negus, Christopher, *Linux Bible 2010 Edition*; Wiley Publishing, Inc. 2010

Command Input and Output

- bash shell responsible for
  - Providing user interface
  - Interpreting commands
  - Manipulating command input and output
    - Provided user specifies certain shell metacharacters with command
- File descriptors
  - Numeric labels that define command input and command output

Command Input and Output

- There are three file descriptors available to each command that can be manipulated by the bash shell:
  - Standard Input (stdin)
  - Standard Output (stdout)
  - Standard Error (stderror)

Command Input and Output

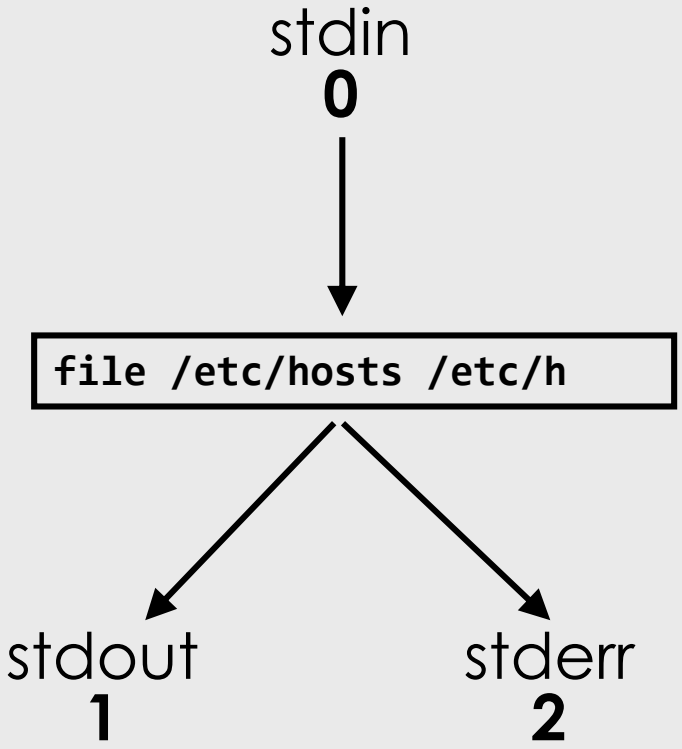
- Standard Input (stdin)
  - File descriptor representing information input to a command during execution
  - Most often user input typed on keyboard
- Standard Output (stdout)
  - File descriptor referring to normal output of the command
  - Displayed on terminal

Command Input and Output

- Standard Error (stderror)
  - File descriptor referring to error messages generated by the command
  - Displayed on terminal

Command Input and Output

Figure 7-1:
The three common file descriptors




```
[root@itm456 ~]# file /etc/hosts /etc/h STDIN
/etc/hosts: ASCII text STDOUT
/etc/h:     ERROR: cannot open `/etc/h' (No such file or directory) /etc/h STDERR
```

Redirection

- bash can redirect Standard Output and Standard Error from the terminal screen to a file
  - Uses "**>**" shell metacharacter followed by absolute or relative pathname of the file
  - Can redirect both Standard Output and Standard Error to **separate** files at the same time
- Important to use separate filenames to hold contents of Standard Output and Standard Error

Redirection


```
[root@itmo456 ~ ]# cat /etc/issue
Fedora release 20 (Heisenbug)
Kernel \r on an \m (\l)

[root@itmo456 ~ ]# cat /etc/tests
cat: /etc/tests: No such file or directory
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat /etc/issue /etc/tests > std_out_redirect.txt
cat: /etc/tests: No such file or directory
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat std_out_redirect.txt
Fedora release 20 (Heisenbug)
Kernel \r on an \m (\l)
```

Redirection

- File automatically created if it did not exist prior to redirection; overwritten if it did exist
  - Use "**>>**" to append additional output to an existing file
  - Will also create new file
- Redirecting stdin to a file
  - Use "**<**" shell metacharacter

Redirection

```
[root@itmo456 ~ ]# echo "This text will be in the newly created file" > newfile.txt
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat newfile.txt
This text will be in the newly created file
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# echo "This will over-write the text in the file" > newfile.txt
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat newfile.txt
This will over-write the text in the file
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# echo "This will append the text to the bottom of the file" >> newfile.txt
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat newfile.txt
This will over-write the text in the file
This will append the text to the bottom of the file
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat /etc/passwd | grep sean
sean:x:1000:1000:Sean:/home/sean:/bin/bash
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# grep sean </etc/passwd
sean:x:1000:1000:Sean:/home/sean:/bin/bash
[root@itmo456 ~ ]#
```

Redirection

- **`tr`** command
  - Replace characters in a file sent via stdin

```
[root@localhost ~]# tr a A < /etc/issue
FedorA releAse 19 (Schrödinger's CAt)
```

- Can redirect stdin and stdout for the same command using "**<**" and "**>**"

```
[root@itm456 ~]# tr a A < /etc/issue > output
[root@itm456 ~]# cat output
FedorA releAse 19 (Schrödinger's CAt)
```

Redirection

- Common redirection examples:

command 1>file Standard Output is sent to
command >file a file instead of the screen

command 2>file Standard Error is sent to a file instead of the screen

command 1>fileA 2>fileB
command >fileA 2>fileB
Standard Output is sent to fileA instead of the screen and the Standard Error is sent to fileB instead of the screen

```
[root@itm456 ~]# file /etc/hosts /etc/h STDIN
/etc/hosts: ASCII text STDOUT
/etc/h:     ERROR: cannot open `/etc/h' (No such file or directory) /etc/h STDERR
```

Redirection

- Common redirection examples:

```
command 1>file 2>&1
command >file 2>&1
```

- Both Standard Output and Standard Error are sent to the same file instead of the screen
  - Use "**&>**" to redirect stdout and stderr to the same file

Redirection

```
[root@itmo456 ~ ]# ls /etc/issue /etc/test
ls: cannot access /etc/test: No such file or directory
/etc/issue
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# ls /etc/issue /etc/test > std_out.txt
ls: cannot access /etc/test: No such file or directory
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat std_out.txt
/etc/issue
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# ls /etc/issue /etc/test > std_out.txt 2> std_error.txt
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat std_out.txt
/etc/issue
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat std_error.txt
ls: cannot access /etc/test: No such file or directory
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# ls /etc/issue /etc/test > std_combined.txt 2> std_combined.txt
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat std_combined.txt
/etc/issue
access /etc/test: No such file or directory
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# ls /etc/issue /etc/test > std_combined.txt 2>&1
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat std_combined.txt
ls: cannot access /etc/test: No such file or directory
/etc/issue
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# ls /etc/issue /etc/test &> std_combined.txt
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat std_combined.txt
ls: cannot access /etc/test: No such file or directory
/etc/issue
```

Redirection

```
[sean@itmo456 ~ ]$ find /etc -name yum
find: ‘/etc/vmware-tools/GuestProxyData/trusted’: Permission denied
find: ‘/etc/ipsec.d’: Permission denied
find: ‘/etc/polkit-1/rules.d’: Permission denied
find: ‘/etc/polkit-1/localauthority’: Permission denied
find: ‘/etc/lvm/archive’: Permission denied
find: ‘/etc/lvm/cache’: Permission denied
find: ‘/etc/lvm/backup’: Permission denied
find: ‘/etc/firewalld’: Permission denied
find: ‘/etc/sudoers.d’: Permission denied
find: ‘/etc/sssd’: Permission denied
find: ‘/etc/cups/ssl’: Permission denied
find: ‘/etc/audit’: Permission denied
find: ‘/etc/grub.d’: Permission denied
/etc/logrotate.d/yum
find: ‘/etc/selinux/targeted/modules/active’: Permission denied
find: ‘/etc/liboath’: Permission denied
/etc/yum
find: ‘/etc/pki/CA/private’: Permission denied
find: ‘/etc/audisp’: Permission denied
find: ‘/etc/dhcp’: Permission denied
[sean@itmo456 ~ ]$
[sean@itmo456 ~ ]$ find /etc -name yum 2>/dev/null
/etc/logrotate.d/yum
/etc/yum
[sean@itmo456 ~ ]$
```

Pipes

- Send stdout of one command to another command as stdin
- Pipe
  - A string of commands connected by "|" metacharacters
  - stdout on left, stdin on right
- Commonly used to reduce amount of information displayed on terminal screen

Pipes

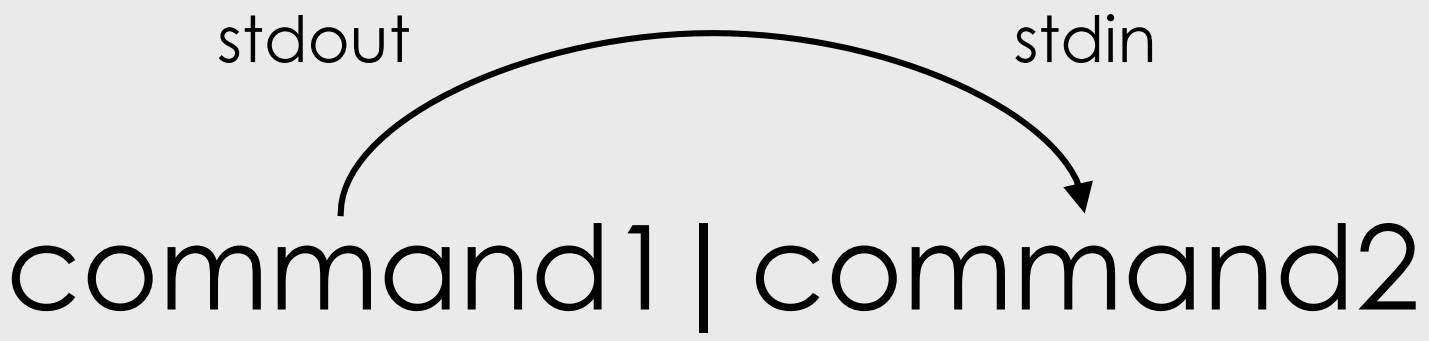


Figure 7-2: Piping information from one command to another

Pipes

- Can use multiple pipes on command line
  - Pass information from one command to another over a series of commands
- filter commands
  - Commands that can take from stdin and give to stdout
  - Can be on either side of a pipe
- tee commands (most common is **`tee`**)
  - Filter commands that also send information to a file as well as stdout

Pipes

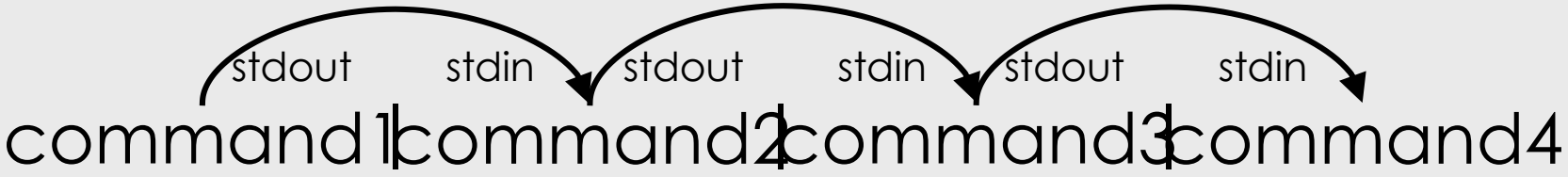


Figure 7-3: Piping several commands

Pipes

Command	Description
sort **sort -r**	Sorts lines in a file alphanumerically Reverse sorts lines in a file alphanumerically
wc **wc -l** **wc -w** **wc -c**	Counts the number of lines, words, and characters in a file Counts the number of lines in a file Counts the number of words in a file Counts the number of characters in a file
pr **pr -d**	Formats a file for printing (has several options available); places a date and page number at the top of each page Formats a file double-spaced
tr	Replaces characters in the text of a file
grep	Displays lines in a file that match a regular expression
nl	Numbers lines in a file
awk	Can be used to extract, manipulate and format text using pattern-action statements
sed	Can be used to manipulate text using search-and-replace expressions

Table 7-2: Common filter commands

Pipes

- Can combine redirection and piping
  - Input redirection must occur at beginning of pipe
  - Output redirection must occur at end of pipe
- **`sed`** filter command
  - Search for and replace text strings
  - Can be used to replace some or all appearances of the search term and to delete specific input lines
- **`awk`** filter command
  - designed for text processing and typically used as a data extraction and reporting tool

Pipes

```
# awk '{print $5}' devices.txt | sort | uniq -c | sort -nr
  79770 example.com/CT349434SC.example.com
  41757 example.com/RI345908SC.example.com
  15815 example.com/NY308831SC.example.com
  14331 example.com/IL332969SC.example.com
  12190 example.com/MA302987SC.example.com
  12069 example.com/NY342057SC.example.com
  11281 example.com/LA351731SC.example.com
  11091 example.com/RI301384SC.example.com
  10604 example.com/NY304010SC.example.com
  10582 example.com/FL310131SC.example.com
  10027 example.com/CT336361SC.example.com
   9485 example.com/NY340611SC.example.com
   9432 example.com/NY342190SC.example.com
   9432 example.com/FL332002SC.example.com
   9145 example.com/NY304771SC.example.com
```

Pipes

```
[root@itmo456 ~ ]# grep sean /etc/passwd
sean:x:1000:1000:Sean:/home/sean:/bin/bash
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# awk -F : '{print $6, $7}' /etc/passwd | grep sean
/home/sean /bin/bash
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat /etc/issue
Fedora release 20 (Heisenbug)
Kernel \r on an \m (\l)

[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat /etc/issue
Fedora release 20 (Heisenbug)
Kernel \r on an \m (\l)

[root@itmo456 ~ ]#
[root@itmo456 ~ ]# sed s/Fedora/ITMO456/ /etc/issue
ITMO456 release 20 (Heisenbug)
Kernel \r on an \m (\l)

[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat /etc/issue
Fedora release 20 (Heisenbug)
Kernel \r on an \m (\l)

[root@itmo456 ~ ]#
[root@itmo456 ~ ]# sed -i s/Fedora/ITMO456/ /etc/issue
[root@itmo456 ~ ]#
[root@itmo456 ~ ]# cat /etc/issue
ITMO456 release 20 (Heisenbug)
Kernel \r on an \m (\l)
```

Running Background Commands

- Commands run from the shell can be run in the background
  - Otherwise shell is unavailable until command completes
  - Syntax is **`command &`**
  - Useful if running a GUI program from a shell command line; example

```
[itm456@itm456fedora]$ firefox &
[1] 5137
[itm456@itm456fedora]$
```

  - `[1]` is the job number aka jobnum
  - **`5137`** is the process ID aka PID or processnum

Running Background Commands

- Other commands useful when executing jobs in the background:
  - **`jobs`** – lists all background jobs in current shell with their job numbers
  - **`fg`** ***`%jobnum`*** – brings a job in the background to the foreground
  - **`bg`** ***`%jobnum`*** – brings a job in the foreground to the background
    - *Must use* **`crtl + z`**
  - **`kill`** ***`%jobnum`*** – cancel and end a job in the background
    - May not be allowed unless you are root

Running Background Commands


```
[sean@itmo456 ~ ]$ firefox
^Z
[1]+  Stopped                 firefox
[sean@itmo456 ~ ]$ jobs
[1]+  Stopped                 firefox
[sean@itmo456 ~ ]$ bg %1
[1]+ firefox &
[sean@itmo456 ~ ]$
[sean@itmo456 ~ ]$ jobs
[1]+  Running                 firefox &
[sean@itmo456 ~ ]$
[sean@itmo456 ~ ]$ fg %1
firefox
```

Shell Variables

- Variable
  - A reserved portion of memory containing accessible information
  - BASH shell has several variables in memory
- Environment variables
  - Contain information that system and programs access regularly

Shell Variables

- User-defined variables
  - Custom variables defined by users
- Special variables
  - Useful when executing commands and creating new files and directories

Environment Variables

- Many environment variables
  - Set by default in bash
- **`set`** command
  - Lists environment variables and current values
- **`echo`** command
  - View contents of a specified variable
  - Prefix variable name with **`$`** shell metacharacter
- Changing value of a variable (assignment):
  - Specify variable name followed by equal sign (**`=`**) and new value
  - **`HOME`** and **`PWD`** variables should not be changed

Environment Variables

- Can also view environment variables using **printenv** command
  - Lists the current values of the variable specified or all variables if none is specified
  - Example:

```
[itm456@itm456 ~]$ printenv PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin
:/usr/sbin:/home/itm456/.local/bin:/home/itm
456/bin
[itm456@itm456 ~]$
```

Environment Variables

Variable	Description
BASH	Full path to the BASH shell
BASH_VERSION	The version of the current BASH shell
DISPLAY	Used to redirect the output of X Windows to another computer or device
ENV	Location of the BASH runtime configuration file (usually **`~/.bashrc`**)
EUID	Effective UID (User ID) of the current user
HISTFILE	The filename used to store previously entered commands in the BASH shell (usually **`~/.bash_history`**)
HISTFILESIZE	The number of previously entered commands that can be stored in the HISTFILE upon logout for use during the next login—it is typically 1000 commands
HISTSIZE	The number of previously entered commands that will be stored in memory during the current login session—typically 1000
HOME	The absolute pathname of the current user's home directory

Table 7-3: Common BASH environment variables

Environment Variables

Variable	Description
HOSTNAME	The hostname of the Linux system
LOGNAME	The username of the current user when logging into the shell
MAIL	The location of the mailbox file (where e-mail is stored)
OLDPWD	The most recent previous working directory
OSTYPE	Identifies the current operating system
PATH	The directories to search for executable program files in the absence of an absolute or relative pathname containing a / character
PS1	The current shell prompt
PWD	The current working directory
RANDOM	Creates a random number when accessed
SHELL	The absolute pathname of the current shell
TERM	Used to determine the terminal settings—it is typically set to "linux" on newer Linux systems and "console" on older Linux systems

Table 7-3: Common BASH environment variables

User-Defined Variables

- Variable identifier
  - Name of a variable
- Creating new variables
  - Specify variable identifier followed by equal sign and the new contents
- Features of variable identifiers
  - Can contain alphanumeric characters, dash characters, or underscore characters
  - Must not start with a number
  - Typically capitalized to follow convention

User-Defined Variables

- Created by assigning a value to a variable identifier

```
VARIABLENAME="variable value"
```

- Subshell
  - Shell created by current shell
  - Most shell commands run in a subshell
  - Variables created in current shell are not available to subshells

User-Defined Variables

- **`export`** command
  - Exports user-defined variables to subshells
  - Ensures that programs started by current shell have access to variables
- **`env`** command
  - Lists all exported environment and user-defined variables in a shell

Other Variables /Special Variables

- Not displayed by the **`set`** or **`env`** commands
  - Perform specialized functions in the shell
- Examples
  - UMASK: Performs a special function in the BASH shell and must be set by the **`umask`** command
  - Alias: allows you to define your own commands through use of the **`alias`** command

Aliases

- Aliases allow you to define your own commands
  - Use unique alias names
  - Aliases stored in special variables
  - Can create single alias to multiple commands
    - Use `;` metacharacter
- Aliases may be set using the command **`alias`** but only persist for the current shell session
  - Current aliases may be viewed by using the **`alias`** command without assigning a value

Aliases

- Local user aliases may be included in the alias section of your **`.bashrc`** file in your root directory
  - Will work anytime you log in
- Some shells store their aliases in a **`.alias`** file which is executed by the login profile or **`rc`** file
  - Can be done with bash if you choose to

Aliases

```
[sean@localhost ~]$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias which='(alias; declare -f) | /usr/bin/which --tty-only --read-alias --read-functions --show-tilde --s
how-dot'
alias xzegrep='xzegrep --color=auto'
alias xzfgrep='xzfgrep --color=auto'
alias xzgrep='xzgrep --color=auto'
alias zegrep='zegrep --color=auto'
alias zfgrep='zfgrep --color=auto'
alias zgrep='zgrep --color=auto'
```

Environment Files

- When exiting bash shell, all stored variables are destroyed
- Environment files
  - Store variables and values
  - Executed each time bash is started
  - Ensures variables are always accessible

Environment Files

- Some common BASH shell environment files and the order they are executed in are listed below:
  - **`/etc/profile`** (global environment)
  - **`/etc/bashrc`** (global functions & aliases)
  - **`~/.bashrc`** (local functions & aliases)
  - **`~/.bash_profile`** (local environment)
  - **`~/.bash_login`**
  - **`~/.profile`**

Environment Files

- **`/etc/profile`** (global environment)
  - Sets up user environment for every user
  - Provides values for path
  - Sets environment variables for mailbox location, size of history files and more
  - Gathers shell settings from configuration files in **`/etc/profile.d`** directory
    - Sets aliases
- **`/etc/bashrc`** (global functions & aliases)
  - Sets default prompt (**`$PS1`**) and may add one or more aliases

Environment Files

```
[sean@localhost ~]$ ll /etc/profile.d/
total 80
-rw-r--r--. 1 root root  771 Feb  3  2016 256term.csh
-rw-r--r--. 1 root root  841 Feb  3  2016 256term.sh
-rw-r--r--. 1 root root  666 Mar 28  2016 bash_completion.sh
-rw-r--r--. 1 root root  196 Apr 22 07:55 colorgrep.csh
-rw-r--r--. 1 root root  201 Apr 22 07:55 colorgrep.sh
-rw-r--r--. 1 root root 1741 Mar  5  2016 colorls.csh
-rw-r--r--. 1 root root 1609 Mar  5  2016 colorls.sh
-rw-r--r--. 1 root root  162 Feb  5  2016 colorxzgrep.csh
-rw-r--r--. 1 root root  183 Feb  5  2016 colorxzgrep.sh
-rw-r--r--. 1 root root  216 Feb  3  2016 colorzgrep.csh
-rw-r--r--. 1 root root  220 Feb  3  2016 colorzgrep.sh
-rw-r--r--. 1 root root 1706 Feb  3  2016 lang.csh
-rw-r--r--. 1 root root 2703 Feb  3  2016 lang.sh
-rw-r--r--. 1 root root  500 Apr 24 23:56 less.csh
-rw-r--r--. 1 root root  253 Apr 24 23:56 less.sh
lrwxrwxrwx. 1 root root   29 Jun 14 11:35 modules.csh -> /etc/alternatives/modules.csh
lrwxrwxrwx. 1 root root   28 Jun 14 11:35 modules.sh -> /etc/alternatives/modules.sh
-rw-r--r--. 1 root root 1206 Jun  7 10:23 PackageKit.sh
-rw-r--r--. 1 root root  663 Jan 20  2015 scl-init.sh
-rw-r--r--. 1 root root 2092 May 11 05:45 vte.sh
-rw-r--r--. 1 root root  164 Feb  5  2016 which2.csh
-rw-r--r--. 1 root root  200 Feb  5  2016 which2.sh
```

Environment Files

```
[sean@localhost ~]$ cat /etc/bashrc
# /etc/bashrc

# System wide functions and aliases
# Environment stuff goes in /etc/profile

# It's NOT a good idea to change this file unless you know what you
# are doing. It's much better to create a custom.sh shell script in
# /etc/profile.d/ to make custom changes to your environment, as this
# will prevent the need for merging in future updates.

# are we an interactive shell?
if [ "$PS1" ]; then
  if [ -z "$PROMPT_COMMAND" ]; then
    case $TERM in
    xterm*|vte*)
      if [ -e /etc/sysconfig/bash-prompt-xterm ]; then
          PROMPT_COMMAND=/etc/sysconfig/bash-prompt-xterm
      elif [ "${VTE_VERSION:-0}" -ge 3405 ]; then
          PROMPT_COMMAND="__vte_prompt_command"
      else
          PROMPT_COMMAND='printf "\033]0;%s@%s:%s\007" "${USER}" "${HOSTNAME%%.*}" "${PWD/#$HOME/\~}"'
      fi
      ;;
    screen*)
      if [ -e /etc/sysconfig/bash-prompt-screen ]; then
          PROMPT_COMMAND=/etc/sysconfig/bash-prompt-screen
```

Environment Files

- Hidden environment files allow users to set customized variables
- To add a variable, add a line to an environment file
  - Use command line syntax
- Any command can be placed inside any environment file
  - e.g., alias creation

Environment Files

- **`~/.bashrc`** (local functions & aliases)
  - Normally runs or call user aliases and any other programs/commands to run on shell start
  - first hidden environment file executed at login
- **`~/.bash_profile`** (local environment)
  - Sets the local PATH and other enviroment variables

Environment Files

```
[root@itmo456 ~ ]# cat .bashrc
# .bashrc

# User specific aliases and functions

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi
[root@itmo456 ~ ]# cat .bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH
export PS1='[\[\e[0;31m\]\u@\[\e[0;35m\]\h\[\e[m\] \[\e[0;34m\]\W\[\e[m\] ]\[\e[0;33m\]\$ \[\e[m\]'
[root@itmo456 ~ ]#
```

Shell Scripts

- Shell script
  - Text files that contain a list of commands or constructs for the shell to execute in order
  - May contain any command that can be entered on command line
- Hashpling (**`#!/bin/bash`**) (aka hashbang)
  - First line in a shell script
  - Defines the shell that will be used to interpret the commands in the script file

Sample Shell Script

```
[itm456@itm456fedora]cat myscript
#!/bin/bash
echo "Today's date is:"
date
echo ""
echo "The people logged into the system
   include:"
who
echo ""
echo "The contents of the / directory are:"
ls -F /
```

Sample Shell Script Output

```
[itm456@itm456fedora]# ./myscript
Today's date is:
Sat Jun  1 08:55:52 EDT 2003

The people logged into the system include:
root     pts/0    Jun  1 08:44 (3.0.0.2)

The contents of the / directory are:
bin/          dev/        home/   lib/   misc/   opt/
root/         tftpboot/   usr/    boot/  etc/    initrd/
lost+found/   mnt/        proc/   sbin/  trap/   var/
[itm456@itm456fedora]#
```

Escape Sequences

- Escape sequences
  - Character sequences that have special meaning inside the **`echo`** command
  - Prefixed by **`\`** character
  - Must specify with the **`-e`** option to the **`echo`** command

Escape Sequences

Escape Sequence	Description
\???	Inserts an ASCII character represented by a three-digit octal number (???)
****	Backslash
\a	ASCII beep
\b	Backspace
\c	Prevents a new line following the command
\f	Form feed
\n	Starts a new line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Table 7-4: Common **echo** escape sequences

Shell Script with Escape Sequences

```
[itm456@itm456fedora]# cat myscript
#!/bin/bash
echo -e "Today's date is:" \c
date
echo -e "\nThe people logged into the system include:"
who
echo -e "\nThe contents of the / directory are:"
ls -F /
```

Shell Script Output

```
[itm456@itm456fedora]# ./myscript
Today's date is: Sat Jun  1 08:55:52 EDT 2003

The people logged into the system include:
root     pts/0    Jun  1 08:44 (3.0.0.2)

The contents of the / directory are:
bin/          dev/        home/   lib/   misc/   opt/
root/         tftpboot/   usr/    boot/  etc/    initrd/
lost+found/   mnt/        proc/   sbin/  trap/   var/
[itm456@itm456fedora]#
```

Reading Standard Input

- Shell scripts may need input from user
  - Input may be stored in a variable for later use
- **`read`** command
  - Takes user input from stdin
  - Places it in a variable specified by an argument to the **`read`** command

Reading Standard Input

```
[itm456@itm456fedora]# cat newscript
#!/bin/bash
echo -e "What is your name? -->\c"
read USERNAME
echo "Hello $USERNAME"
[itm456@itm456fedora]# chmod a+x newscript
[itm456@itm456fedora]# ./newscript
What is your name? --> Fred
Hello Fred
[itm456@itm456fedora]#
```

Decision Constructs

- Most common type of construct used in shell scripts
- Alter flow of a program:
  - Based on whether a command completed successfully
  - Based on user input

Decision Constructs

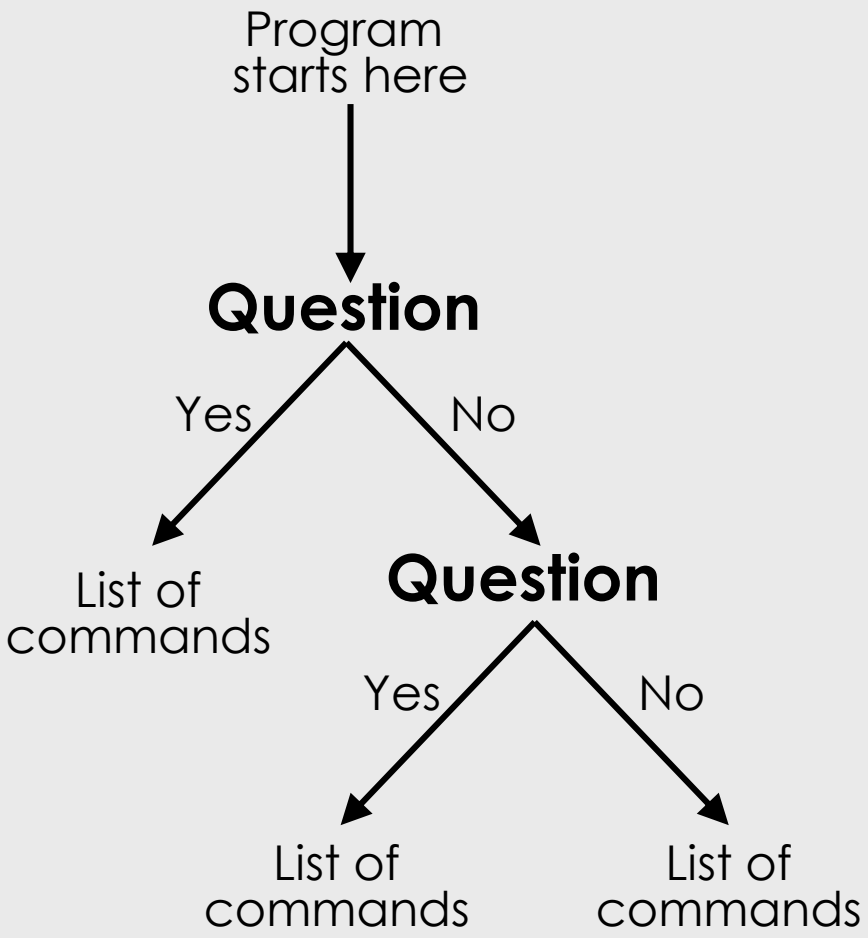


Figure 7-4: A two-question decision construct

Decision Constructs

Figure 7-5: A command-based decision construct

The `if` Construct

- Control flow of program based on true/false decisions
- Syntax:

```
if     this is true
then
  do these commands
elif   this is true
then
  do these commands
else
  do these commands
fi
```

The `if` Construct

- Common rules that govern **`if`** constructs:
  - **`elif`** (else if) and **`else`** statements are optional
  - Unlimited number of **`elif`** statements
  - *do these commands* section may consist of multiple commands
    - One per line

The if Construct

- Common rules that govern if constructs (continued):
  - *do these commands* section typically indented from the left hand side of the text file for readability (don't need to be)
  - End of the statement must be backwards "if": **`fi`**
  - *this is true* of the **`if`** syntax seen earlier — the "condition"—may be a **command** or a **test statement**

The if Construct

- test statement
  - Used to test a condition
  - Generates a true/false value
  - Inside of square brackets (**[…]**) or prefixed by the word "test"
    - Must have spaces after "[" and before "]"
- Special comparison operators:
  - **-o** (OR)
  - **-a** (AND)
  - **!** (NOT)

The if Construct

Test Statement	Returns true if
[A = B]	String A is equal to string B
[A != B]	String A is not equal to string B
[A -eq B]	A is numerically equal to B
[A -ne B]	A is numerically not equal to B
[A -lt B]	A is numerically less than B
[A -gt B]	A is numerically greater than B
[A -le B]	A is numerically less than or equal to B
[A -ge B]	A is numerically greater than or equal to B
[-r A]	A is a file/directory that exists and is readable (r permission)
[-w A]	A is a file/directory that exists and is writeable (w permission)
[-x A]	A is a file/directory that exists and is executable (x permission)
[-f A]	A is a file that exists
[-d A]	A is a directory that exists

Table 7-5. Common test statements

The if Construct

Test Statement	Returns true if
[A = B -o C = D]	String A is equal to string B OR string C is equal to string D
[A = B -a C = D]	String A is equal to string B AND string C is equal to string D
[! A = B]	String A is NOT equal to string B

Table 7-6: Special operators in test statements

The if Construct

```
[student@itm456 ~]$ cat basic_if
#!/bin/bash
read -p "Word 1: " word1
read -p "Word 2: " word2

if test "$word1" = "$word2"
        then
            echo "Match"
        else
            echo "No Match"
fi
[student@itm456 ~]$ ./basic_if
Word 1: itm456
Word 2: linux
No Match
```

The `if` Construct

```
[student@itm456 ~]$ cat enhanced_if
#!/bin/bash
read -p "Word 1: " word1
read -p "Word 2: " word2
read -p "Word 3: " word3

if [ "$word1" = "$word2" -a "$word2" = "$word3" ]
        then
            echo "Match: words 1, 2, and 3"
        elif [ "$word1" = "$word2" ]
        then
            echo "Match: words 1 and 2"
        elif [ "$word1" = "$word3" ]
        then
            echo "Match: words 1 and 3"
        elif [ "$word2" = "$word3" ]
        then
            echo "Match: words 2 and 3"
        else
            echo "No Match"
fi
```

The if Construct

```
[student@itm456 ~]$ ./enhanced_if
Word 1: itm456
Word 2: test
Word 3: apple
No Match
[student@itm456 ~]$ ./enhanced_if
Word 1: itm456
Word 2: test
Word 3: itm456
Match: words 1 and 3
[student@itm456 ~]$ ./enhanced_if
Word 1: test
Word 2: itm456
Word 3: itm456
Match: words 2 and 3
[student@itm456 ~]$ ./enhanced_if
Word 1: itm456
Word 2: itm456
Word 3: itm456
Match: words 1, 2, and 3
```

The && and || Constructs

- Time-saving shortcut constructs
  - When only one decision needs to be made during execution
- Syntax :

```
command1   &&   command2
command1   ||   command2
```

  - **&&**: 2nd command executes only if 1st completes successfully
  - **||**: 2nd command executes only if 1st fails

The && and || Constructs

```
[root@localhost ~]# cat example.sh
#!/bin/bash

mkdir /etc/sample && cp /etc/hosts /etc/sample
[root@localhost ~]# ./example.sh
[root@localhost ~]# ll /etc/sample/
total 4
-rw-r--r--. 1 root root 158 Oct  4 11:08 hosts
[root@localhost ~]#
[root@localhost ~]# cat example2.sh
#!/bin/bash

mkdir /etc/sample || echo "Could not create /etc/sample"
cp /etc/hosts /etc/sample || echo "Could not copy /etc/hosts"
[root@localhost ~]# ./example2.sh
mkdir: cannot create directory ‘/etc/sample’: File exists
Could not create /etc/sample
```

The `loop` Construct

- Shell scripting also supports looping
  - Loops execute commands repetitively
  - Alter the flow of a program based on the results of a particular statement
  - Repeat entire program until reach desired result
- Loop control structures include
  - `while`
  - `until`
  - `for`
  - `case`

The `case` Construct

- Compares the value of a variable with several different patterns of text or numbers
- If there is a match, commands to the right of the pattern will be executed
- As with the **`if`** construct, the **`case`** construct must be ended by a backwards "case" (**`esac`**)

The `case` Construct

- Syntax:

```
case  variable in
        pattern1   )   do this
                       ;;
        pattern2   )   do this
                       ;;
        pattern3   )   do this
                       ;;
esac
```

The `case` Construct

```
[student@itm456 ~]$ cat case.sh
#!/bin/bash
read -p "Enter A, B, or C: " letter
case "$letter" in
    a|A)
        echo "You entered A"
        ;;
    b|B)
        echo "You entered B"
        ;;
    c|C)
        echo "You entered C"
        ;;
    *)
        echo "You did not enter A, B, or C"
        ;;
esac
[student@itm456 ~]$ ./case.sh
Enter A, B, or C: D
You did not enter A, B, or C
[student@itm456 ~]$ ./case.sh
Enter A, B, or C: C
You entered C
[student@itm456 ~]$ ./case.sh
Enter A, B, or C: a
You entered A
```

The `for` Construct

- Used to process a list of objects
- Syntax:

```
for var_name in string1 string2 … …
 do
 these commands
 done
```

- During execution sets **`var_name`** to a string name, and executes commands between do and done for that string
  - Repeats for all strings

The **for** Construct

```
[student@itm456 ~]$ cat for
#!/bin/bash
echo -e "What directory has the files that you would like to rename?
-->\c"
read DIR
for NAME in $DIR/*
do
mv $NAME $NAME.txt
done
[student@itm456 ~]$ ls temp
file1  file2  file3  file4  file5
[student@itm456 ~]$ ./for
What directory has the files that you would like to rename?
-->temp
[student@itm456 ~]$ ls temp/
file1.txt  file2.txt  file3.txt  file4.txt  file5.txt
```

The `while` Construct

- Begins with test statement
  - Commands within loop construct are executed as long as the test statement returns true
  - Contains a counter variable
    - Value changes with each iteration of the loop
- Syntax:

```
while this returns true
 do
 these commands
 done
```

The **while** Construct

```
[student@itm456 ~]$ cat while_loop.sh
#!/bin/bash
COUNTER=0
while [ $COUNTER -lt 7 ]
do
        echo "I love ITM456!!!"
        ((COUNTER +=1))
done
[student@itm456 ~]$ ./while_loop.sh
I love ITM456!!!
I love ITM456!!!
I love ITM456!!!
I love ITM456!!!
I love ITM456!!!
I love ITM456!!!
I love ITM456!!!
```

The `until` Construct

- Similar to while construct
  - Differs only in the way they test
- Syntax:

```
until this returns true
 do
 these commands
 done
```

The `until` Construct

```
[student@itm456 ~]$ cat until.sh
#!/bin/bash
secretname=sean
name=noname
echo "Try to guess the secret name!"
echo
until [ "$name" = "$secretname" ]
do
        read -p "Your guess: " name
done
echo "You guessed the name!"
[student@itm456 ~]$ ./until.sh
Try to guess the secret name!

Your guess: tim
Your guess: john
Your guess: sam
Your guess: sean
You guessed the name!
```

Summary

- Users must log in to a terminal and receive a shell before they can interact with the Linux system and kernel
- The BASH shell is the default shell in most Linux distributions
- Three components are available to commands: Standard Input, Standard Output, and Standard Error
- Standard Input is typically user input taken from the keyboard

Summary

- Standard Output and Standard Error are sent to the terminal screen by default
- You may redirect the Standard Output and Standard Error of a command to a file using redirection symbols
- Use the pipe symbol to redirect the Standard Output from one command to the Standard Input of another

Summary

- Most variables available to the BASH shell are environment variables, which are loaded into memory after login from environment files
- You may create your own variables in the BASH shell and export them so they are available to programs started by the shell
- You may create your own variables in the BASH shell and export them so they're available to programs started by the shell

Summary

- Shell scripts can be used to execute several Linux commands
- Decision constructs may be used within shell scripts to execute certain Linux commands based on user input or the results of a certain command
- Loop constructs can be used within shell scripts to execute a series of commands repetitively

The End…

- Questions?