

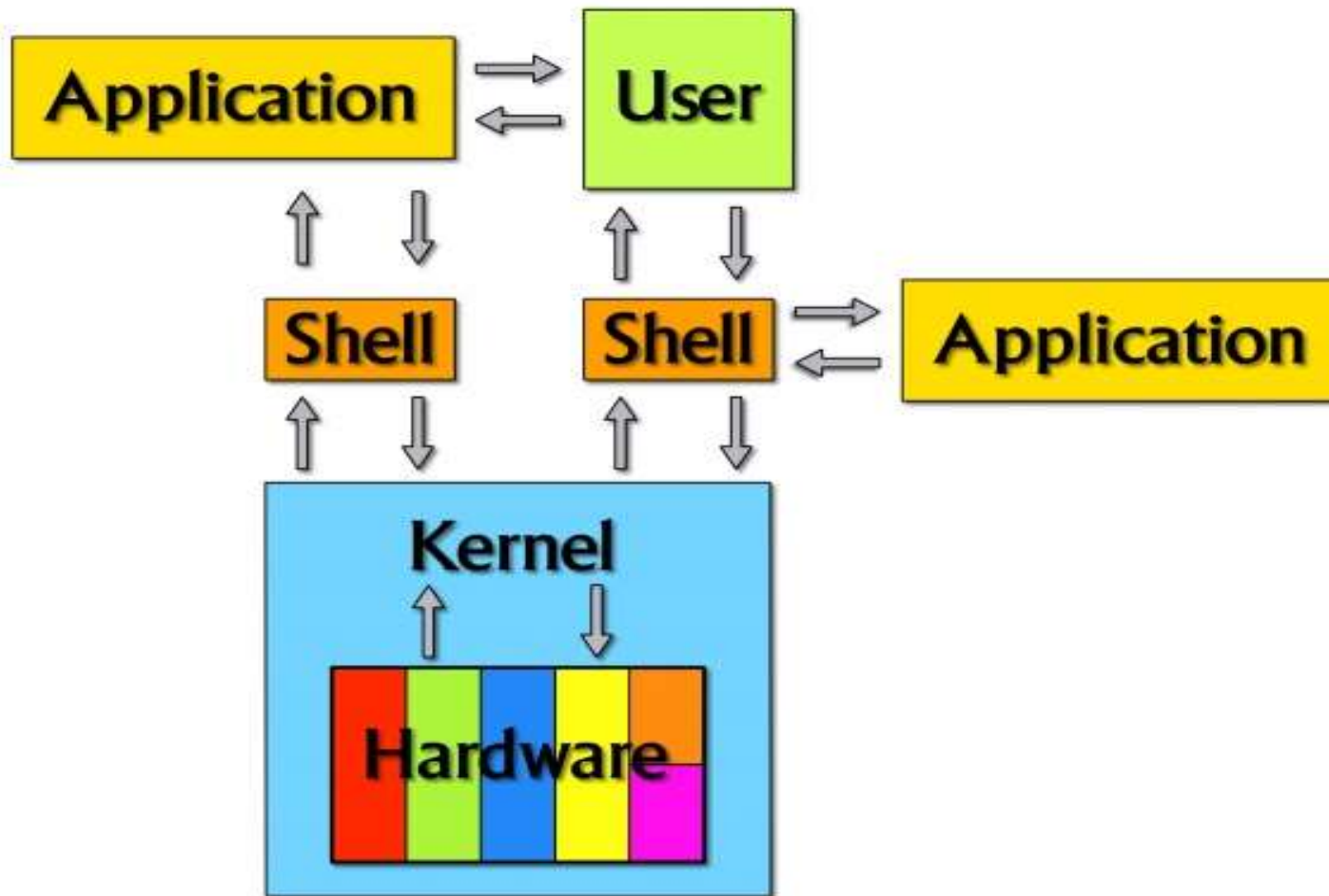
# Using the Shell

**Bok, JongSoon**  
**javaexpert@nate.com**  
**<https://github.com/swacademy/fss>**

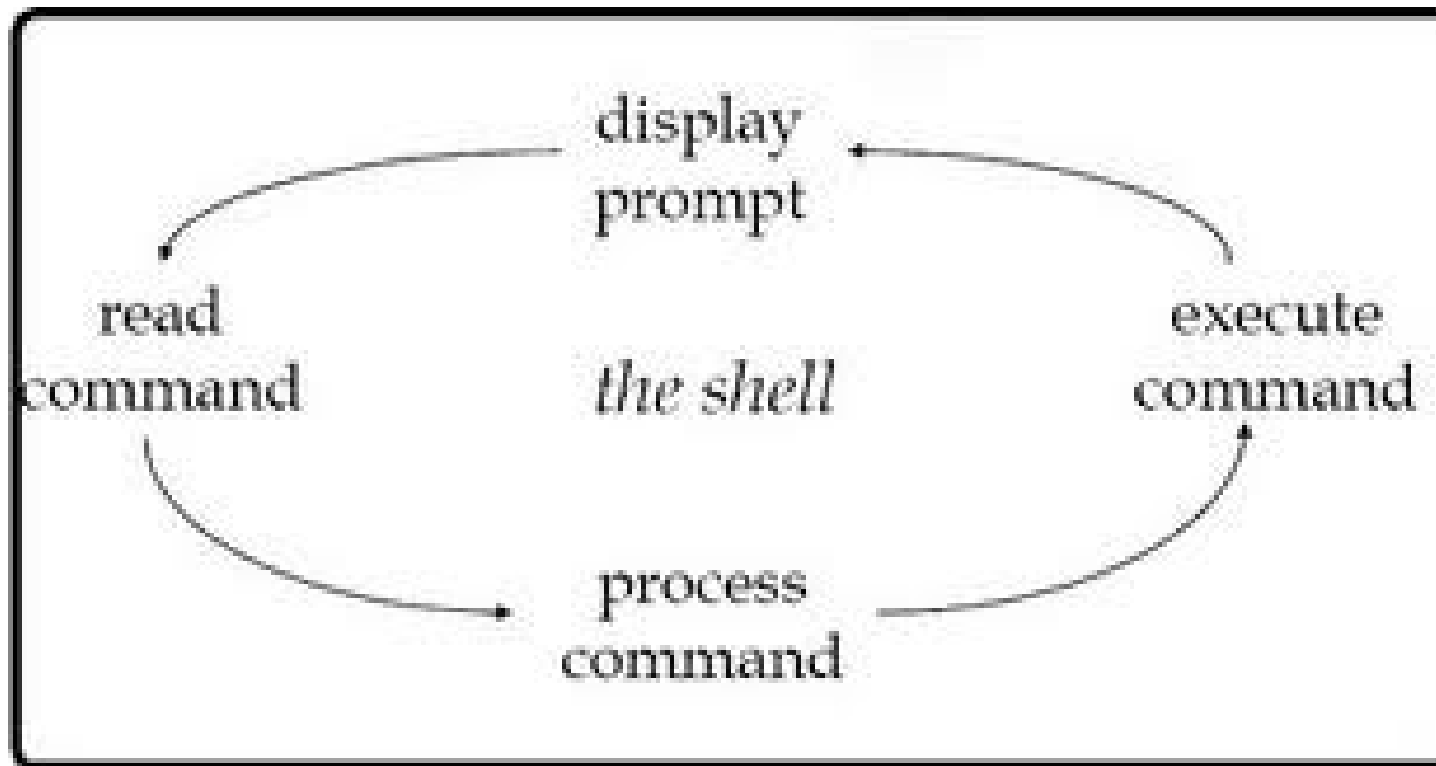
# The Shell

- Is a command interpreter.
- Provides a line-oriented interactive.
- Provides non-interactive interface between the user and the operating system.
- Enter commands on a command line → Are interpreted by the shell and then sent as instructions to the operating system.
- Several different types of shells have been developed for Linux.

## The Shell (Cont.)



## The Shell (Cont.)



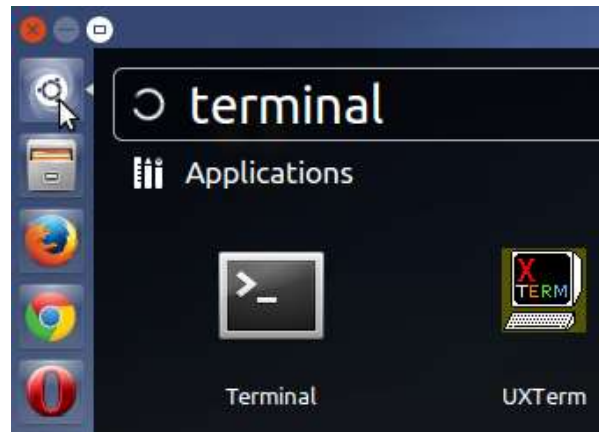
## The Shell (Cont.)

- The Bourne Again shell (*BASH*), the *Korn* shell, the *TCSH* shell, and the *Z* shell.
- *TCSH* is an enhanced version of the *C* shell used on many Unix systems, especially Berkeley Software Distribution (BSD) versions.
- Need only one type of shell to do your work.
- Linux includes all the major shells.
- By default, installs and uses the *BASH* shell.
- If use the command line shell, will be using the *BASH* shell unless you specify another.

You can find out more about the *BASH* shell at <http://gnu.org/software/bash>.

# Accessing Shells

- Can access shells in several ways.
- From the desktop, can use a *terminal* window.
- Can also boot directly to a command line interface, starting up in a shell command line.
- Once a *terminal* window is open you can enter shell commands.



# Bash Output

## ■ echo

- Display message on screen, writes each given STRING to standard output, with a space between each and a newline after the last one.
- Is a BASH built-in command.
- *echo* [options] ... [string]...
- *-n* : Do not output the trailing newline.
- examples

```
$ echo
```

```
$ echo linux
```

```
$ echo "Ubuntu Linux"
```

# Bash Output (Cont.)

## ■ printf

- Format and print data.
- Write the formatted *arguments* to the standard output under the control of the *format*.
- *printf format [argument]...*
- *\n, \b, \r, \t, %, %s, %d, %f*
- examples

```
$ printf "%d\n" 5
```

```
$ printf "%f\n" 5
```

```
$ printf "Hello, $USER.\n\n"
```

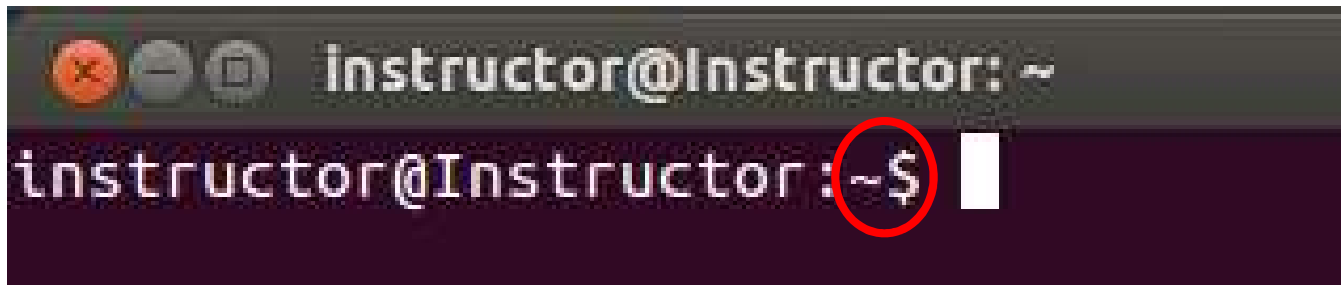


# The Command Line

- The Linux command line interface consists of a single line.
- A shell *prompt*, such as the one shown here, marks the beginning of the command line:

\$

- By default, the *BASH* shell uses a dollar sign (\$) prompt.
- The *root* user will have a different prompt, the #.

A screenshot of a Linux terminal window. The title bar shows window control buttons and the text 'instructor@Instructor: ~'. The terminal content shows the prompt 'instructor@Instructor: ~\$' with a white cursor. A red circle highlights the '\$' character in the prompt.

```
instructor@Instructor: ~$
```

# The Command Line (Cont.)

## ■ Syntax

- `$ command options [argument1, [argument2,...`



```
$ ls -l
```

```
$ ls -l mydata
```

*Some commands can be complex and take some time to execute.*

*When you mistakenly execute the wrong command, you can interrupt and stop such commands with the interrupt key—**CTRL-C**.*

# BASH Metacharacters

- Provides a set of special characters that search out, match, and generate a list of filenames.
- These are `*`, `?`, `[ ]`, `~`, `-`, `;`, `|`, `'`, `"`, ```, `\`, `>`, `<`, `>>`.
- Given a partial filename, the shell uses these matching operators to search for files and expand to a list of filenames found.

**\***, **?**, **[ ]**

■ **\***

- Matching.
- Matches any number of any character.
- \$ **ls e\*.txt**

■ **?**

- Matching.
- Matches any single character.
- \$ **ls e?.ext**

■ **[ ]**

- Matching, listing.
- One of the characters within the square brackets must be matched.
- \$ **ls -l e[abc].txt**

**\*, ?, [ ] (Cont.)**

**\$ ls**

doc1 doc2 document docs mydoc Monday Tuesday

**\$ ls doc\***

doc1 doc2 document docs

**\$ ls \*day**

monday tuesday

**\$ ls m\*d\***

monday

**\*, ?, [ ] (Cont.)**

```
$ ls *.c
```

```
calc.c    main.c
```

```
$ rm doc*
```

## **\*, ?, [ ] (Cont.)**

**\$ ls**

doc1          docA          document

**\$ ls doc?**

doc1          docA

## **\*, ?, [ ] (Cont.)**

```
$ ls
```

```
doc1    doc2    doc3    docA    docB    docD    document
```

```
$ ls doc[1A]
```

```
doc1    docA
```

```
$ ls *. [co]
```

```
main.c  main.o  calc.c
```

```
$ ls doc[1-3]
```

```
doc1    doc2    doc3
```

```
$ ls doc[B-E]
```

```
docB    docD
```



~, -

## ■ ~(tilde)

- Home directory character.
- Corresponds to the *\$HOME* internal variable.
- \$ **ls** ~
- \$ **cd** ~
- \$ **echo** ~

## ■ -

- Previous working directory character.
- Uses the *\$OLDPWD* environmental variable.
- \$ **cd** -

;  
|

■ ;

- Allows to list multiple commands on a single line, separated by this character.
- `$ ls ; date ; pwd`

■ |

- Pipe.
- This sends the output of the command to the left as the input to the command on the right of the symbol.
- `$ ls -al | more`

`\ ' , \"`

■ `\ '`

- Quoting.
- Used to prevent the shell from interpreting special characters within the quoted string.
- `$ echo '$HOME' → $HOME`

■ `\"`

- Quoting.
- Used to prevent the shell from interpreting special characters within the quoted string except `$`, ```, `|`.
- `$ echo "$HOME" → /home/instructor`

## ‘ (back quotes), \ (backslash)



‘ ‘

- Unquoting.
- Used within a quoted string to force the shell to interpret and run the command between the back quotes.
- `$ echo "Today is `date`"`



\

- Escape.
- Ignore the shell's special meaning for the character after this symbol.
- `$ echo \ $HOME`
- `$ echo "I have \$300.00"`

# &&, ||

## ■ &&

- Run the command to the right of the double-ampersand *ONLY IF* the command on the left *succeeded* in running.
- `$ mkdir stuff && echo "Made the directory"`

## ■ ||

- Run the command on the right of the double pipe *ONLY IF* the command on the left *failed*.
- `$ mkdir stuff || echo "mkdir failed!"`

# Matching Shell Symbols

- Often a file expansion character is actually part of a filename.
- In these cases, need to quote the character by preceding it with a backslash (\) to reference the file.

```
$ ls answers\?
```

```
answers?
```

```
$ ls "answers?"
```

```
answers?
```

```
$ ls My\ Documents
```

```
My Documents
```

```
$ ls "My Documents"
```

```
My Documents
```

# Generating Patterns

- { }

- Is often useful for generating names that you can use to create or modify files and directories.

```
$ echo doc{ument,final,draft}
```

```
document    docfinal    docdraft
```

```
$ mkdir { fall,winter,spring }report
```

```
$ ls
```

```
fallreport    springreport    winterreport
```

# Command and Filename Completion

- Automatic completions can be effected by pressing the *TAB* key.
- If enter an incomplete pattern as a command or filename argument, can press the *TAB* key to activate the command and filename completion feature.

```
$ cat pre TAB
```

```
$ cat preface
```



## Command and Filename Completion (Cont.)

- A listing of possible automatic completions follows:
  - Filenames begin with any text or **/**.
  - Shell variable text begins with a **\$** sign.
  - Username text begins with a **~** sign.
  - Host name text begins with an **@**.
  - Commands, aliases, and text in files begin with normal text.

```
$ echo $HOM TAB
```

```
$ echo $HOME
```

## Command and Filename Completion (Cont.)

```
$ echo $H TAB TAB
```

```
$HISTCMD $HISTFILE $HISTSIZE $HOSTNAME
```

```
$HISTCONTROL $HISTFILESIZE $HOME
```

```
$HOSTTYPE
```

# History

- Keeps a *history list* of all the commands you enter.
- Can display each command, in turn, on command line by pressing the *UP ARROW* key.
- Press the *DOWN ARROW* key to move down the list.

# History Events

- The *history utility* keeps a record of the most recent commands you have executed.
- The commands are numbered starting at 1, and a limit exists to the number of commands remembered—the default is 500.
- To see the set of most recent commands, type **history** on the command line and press *ENTER*.

## History Events (Cont.)

```
$ history
```

```
1 cp mydata today
2 vi mydata
3 mv mydata reports
4 cd reports
5 ls
```

```
$ history | grep cd
```

```
4 cd reports
```

```
$ history 3
```

```
3 mv mydata reports
4 cd reports
5 ls
```

# History Events (Cont.)

History Event References	
<b>! event num</b>	References an event with an event number
<b>!!</b>	References the previous command
<b>! characters</b>	References an event with beginning characters
<b>!? pattern?</b>	References an event with a pattern in the event
<b>! - event num</b>	References an event with an offset from the first event
<b>! num - num</b>	References a range of events

## History Events (Cont.)

```
$ !3
```

```
mv mydata reports
```

```
$ !mv my
```

```
mv mydata reports
```

```
$ !-4
```

```
vi mydata
```

```
$ !!
```

```
ls
```

```
mydata today reports
```

# Standard Input / Output

- When a Linux command is executed and produces output, this output is placed in the standard output data stream.
- The default destination for the standard output data stream is a device—in this case, the *screen*.
- *Devices*, such as the keyboard and screen, are treated as files.
- The screen is a device that displays a continuous stream of bytes.



# Standard Input / Output (Cont.)

## ■ The **ls** command :

1. Generates a list of all filenames and outputs this list to the standard output.
2. This stream of bytes in the standard output is directed to the screen device.
3. The list of filenames is printed on the screen.

## ■ The **cat** command :

1. Sends output to the standard output.
2. The contents of a file are copied to the standard output.
3. Default destination is the screen.
4. The contents of the file are then displayed on the screen.

# Redirecting the Standard Output : > and >>

## ■ command > filename

- Redirect the standard output to a file or device.
- Create the file if it does not exist.
- Overwrite the file if it does exist.

## ■ command < filename

- Redirect the standard input from a file or device to a program.

## ■ command >> filename

- Redirect the standard output to a file or device.
- Appending the output to the end of the file.

## Redirecting the Standard Output : > and >> (Cont.)

```
$ ls -l *.c > programlist
```

```
$ ls
```

```
mydata      intro      preface
```

```
$ ls > listf
```

```
$ cat listf
```

```
mydata      intro      listf      preface
```

```
$ cat myletter >> alletters
```

```
$ cat oldletter >> alletters
```



---



# Lab. Output redirecting

>

1. `$ cd ~`
2. `$ ls -lR`
3. `$ ls -lR > dir-tree.list`
4. `$ cat dir-tree.list`

1. `$ date`
2. `$ date > out_date`
3. `$ cat out_date`
4. `$ date 1> out_date1`
5. `$ cat out_date1`

## > (Cont.)

1. `$ date > out1`
2. `$ cat out1`
3. `$ ls -al > out1`
4. `$ cat out1`
5. `$ rm out1`
6. `$ date > out1`
7. `$ set -o noclobber`
8. `$ ls -al > out1`

`#bash: out1 : cannot overwrite existing file`

## > (Cont.)

1. `$ date > out2`

2. `$ set -o noclobber`

3. `$ ls -al > out2`

`#bash: out2 : cannot overwrite existing file`

4. `$ set +o noclobber`

5. `$ ls -al > out2`

6. `$ cat > out2`

## > (Cont.)

1. `$ cat > out3`

Ubuntu Linux

I love Linux

^C

2. `$ cat out3`



## >> (Cont.)

1. \$ **cat out3**

Ubuntu Linux

I love Linux

2. \$ **date >> out3**

3. \$ **cat out3**

Ubuntu Linux

I love Linux

4. \$ **: > test**

5. \$ **ls -l test**

# The Standard Input (stdin)

- The standard input itself receives data from a device or a file.
- The default device for the standard input is the *keyboard*.
- Can also redirect the standard input, receiving input from a file rather than the keyboard.
- The operator for redirecting the standard input is the less-than sign (<).

## The Standard Input (Cont.)

```
$ cat < myletter
```

Hello Christopher

How are you today

```
$
```

```
$ cat < myletter > newletter
```



# Lab. Standard error redirecting



# The Standard Error (stderr)

- `$ ls`

out1 out2 out3

- `$ ls /abc`

ls : cannot access /abc : No such file or directory

- `$ ls > ls.out`

- `$ ls /abc > ls.err`

ls : cannot access /abc : No such file or directory

- `$ cat ls.err`

# The Standard Error (stderr)

- `$ ls /abc 2> ls.err`

- `$ cat ls.err`

`ls : cannot access /abc : No such file or directory`

- `$ ls . /abc > ls.out 2> ls.err`

- `$ cat ls.out`

- `$ cat ls.err`

- `$ ls /abc 2> /dev/null`

- `$ ls . /abc > ls.out 2>&1`

# Shell Configuration

- 4 different major shells are commonly used on Linux systems.
- The Bourne Again shell (BASH)
  - An advanced version of the Bourne shell
  - Includes most of the advanced features developed for the *Korn* shell and the *C* shell
- The AT&T Korn shell : open source
- The TCSH shell
  - An enhanced version of the C shell
  - Originally developed for BSD versions of Unix
- The Z shell
  - An enhanced version of the *Korn* shell.

## Shell Configuration (Cont.)

- The *BASH* shell is the default.
- The shell prompt is a dollar sign (\$).
- Can change to another shell by entering its name.
- Entering **tcsh** invokes the *TCSH* shell, **bash** the *BASH* shell, **ksh** the *Korn* shell, and **zsh** the *Z* shell.
- Can exit a shell by pressing **CTRL + D** or using the **exit** command.



## Shell Configuration (Cont.)

Command	Description
bash	BASH shell, /bin/bash
bsh	BASH shell, /bin/bsh (link to /bin/bash)
sh	BASH shell, /bin/sh (link to /bin/bash)
tcsh	TCSH shell, /usr/tcsh
csch	TCSH shell, /bin/csh (link to /bin/tcsh)
ksh	Korn shell, /bin/ksh (also added link /usr/bin/ksh)
zsh	Z shell, /bin/zsh

# *BASH* Shell Configuration Files

Filename	Function
<b>.profile</b>	Login initialization file
<b>.bashrc</b>	BASH shell configuration file
<b>.bash_logout</b>	Logout name
<b>.bash_history</b>	History file
<b>.bash_aliases</b>	Aliases definition file
<b>/etc/profile</b>	System login initialization file
<b>/etc/bash.bashrc</b>	System BASH shell configuration file

# Aliases

- Can use the **alias** command to create another name for a command.
- The **alias** command operates like a *macro*.
- An **alias** command begins with the keyword **alias** and the new name for the command.
- Followed by an equal sign and the command the **alias** will reference.
- No spaces should be placed around the equal sign used in the **alias** command.

## Aliases (Cont.)

```
$ alias list=ls
```

```
$ ls
```

```
mydata    today
```

```
$ list
```

```
mydata    today
```

```
$ alias c=clear
```

```
$ alias
```

```
alias c='clear'
```

```
$ c
```

## Aliases (Cont.)

- Can also place aliases in the `.bashrc` file directly.
- Some are already defined, though commented out.
- Can edit the `.bashrc` file and remove the `#` comment symbols from those lines to activate the aliases:

```
# some more ls aliases  
alias ll='ls -l'  
alias la='ls -A'  
alias l='ls -CF'
```

## Aliases (Cont.)

- Any command alias that contains spaces must be enclosed in single quotes.

```
$ alias lss='ls -s'
```

```
$ lss
```

```
mydata 14      today 6      reports 1
```

```
$ alias lsa='ls -F'
```

```
$ lsa
```

```
mydata      today  reports/
```

```
$
```

## Aliases (Cont.)

```
$ alias listlong='ls -lh'
```

```
$ listlong
```

```
-rw-r--r-- 1 root root 51K Sep 18 2003 mydata
```

```
-rw-r--r-- 1 root root 16K Sep 27 2003 today
```

```
$ alias lsc='ls *.[co]'
```

```
$ lsc
```

```
main.c      main.o      lib.c      lib.o
```

## Aliases (Cont.)

- The **alias** command provides a list of all aliases that have been defined, showing the commands.
- Can remove an **alias** by using the **unalias** command.

```
$ alias
```

```
lsa=ls -F
```

```
list=ls
```

```
rm=rm -i
```

```
$ unalias lsa
```



# Shell variables

- Can use variables as in any programming languages.
- Are no data types.
- Can contain a number, a character, a string of characters.
- Syntax
  - Variable\_name=string
  - No space between variable and value
  - \$ **SOME=test**
  - \$ **echo \$SOME**

# Shell variables (Cont.)

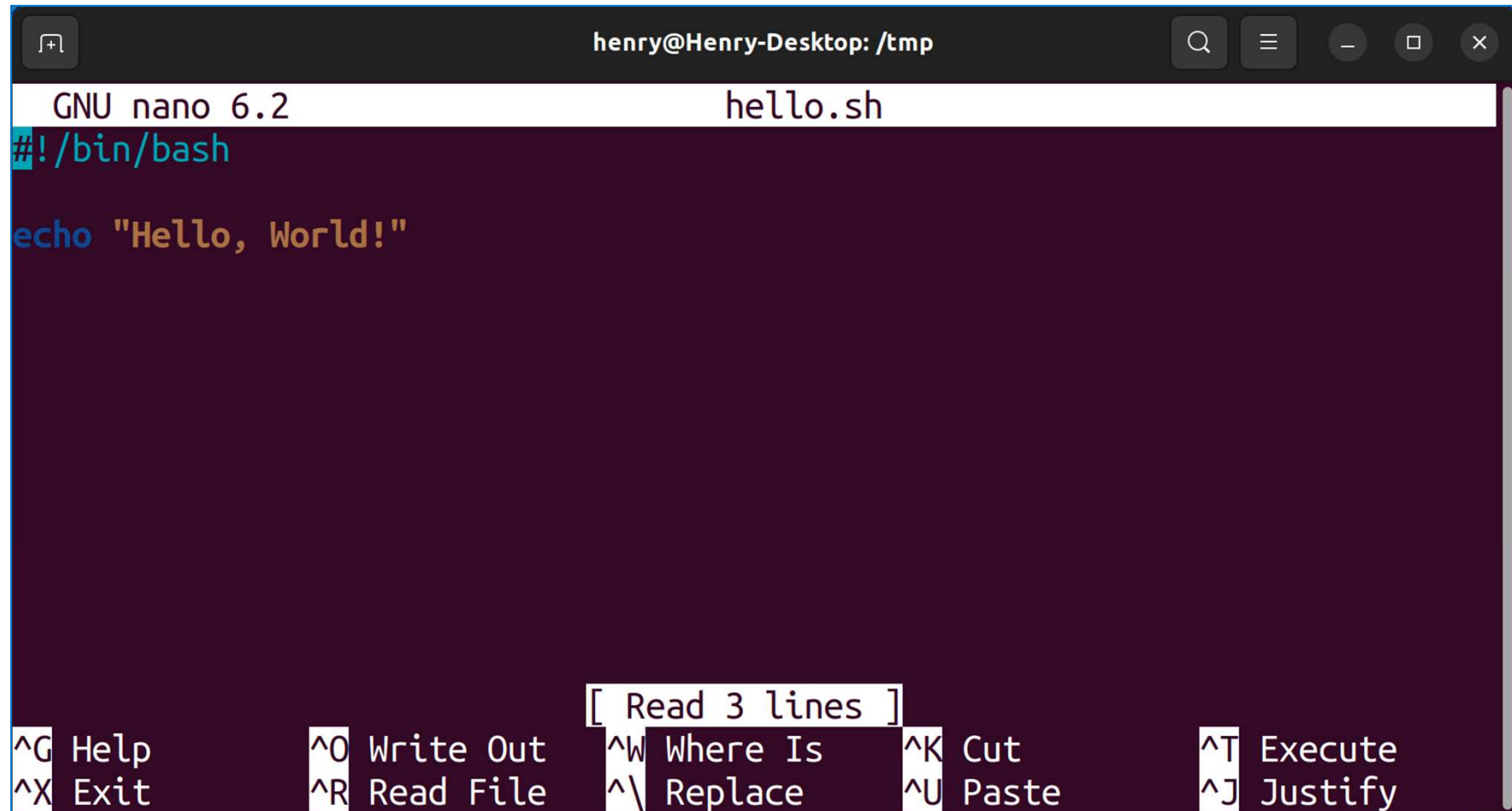
## ■ set

- Manipulate shell variables.
- Example
  - `$ Mydept=Sales`

## ■ unset

- Remove variable.
- Syntax
  - `unset [name]`
- Example
  - `$ unset SOME`
- Is a bash built in command.

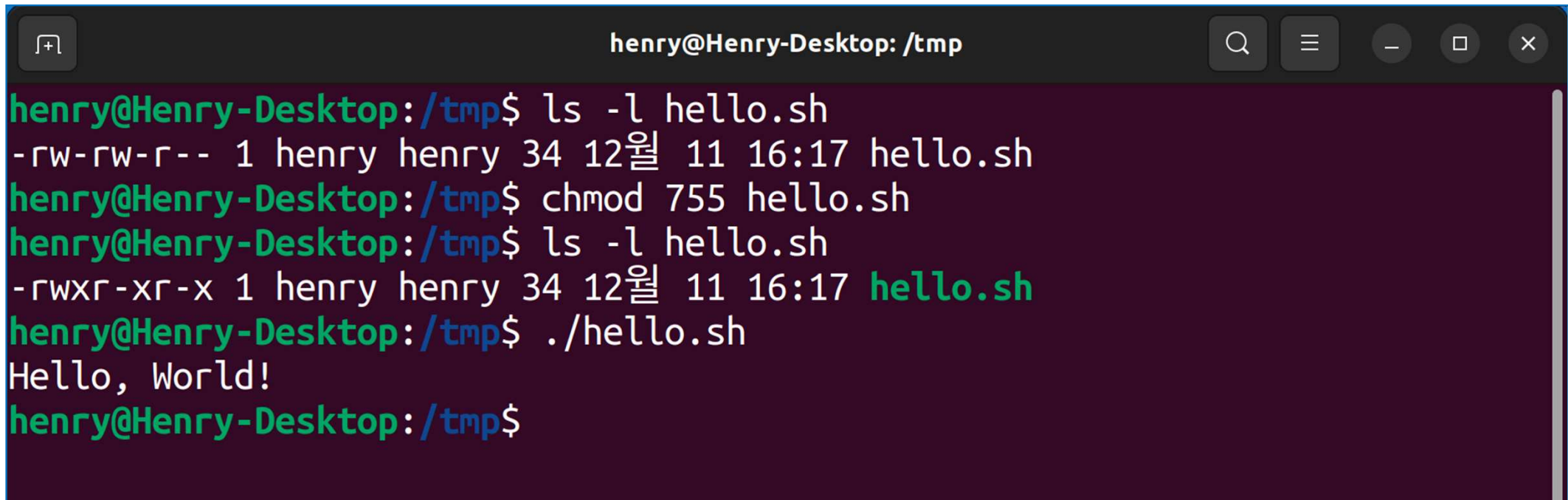
# Shell variables (Cont.)



```
henry@Henry-Desktop: /tmp
GNU nano 6.2 hello.sh
#!/bin/bash
echo "Hello, World!"

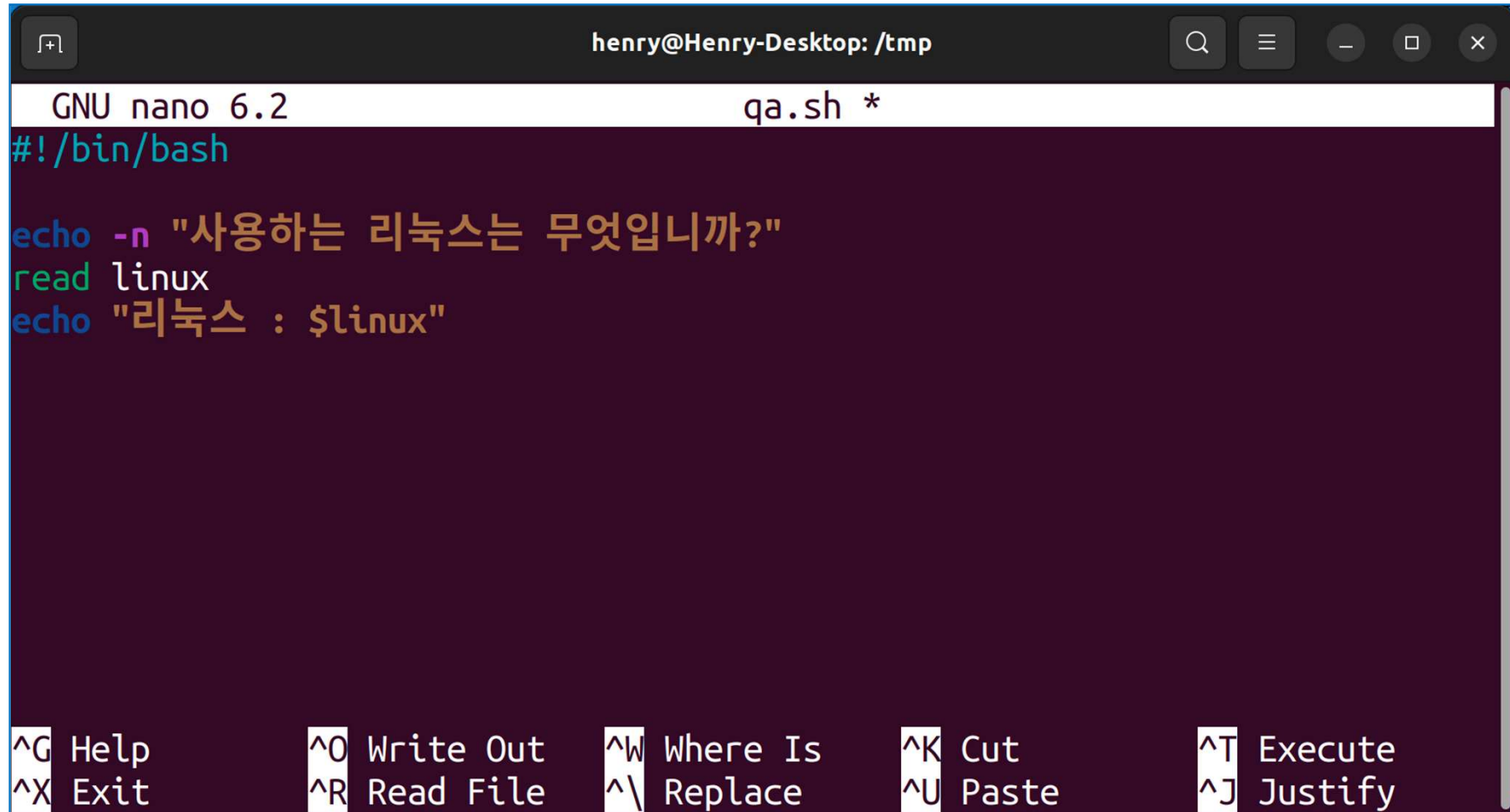
[ Read 3 lines ]
^G Help      ^O Write Out  ^W Where Is   ^K Cut        ^T Execute
^X Exit      ^R Read File  ^\ Replace    ^U Paste      ^J Justify
```

## Shell variables (Cont.)



```
henry@Henry-Desktop: /tmp
henry@Henry-Desktop:/tmp$ ls -l hello.sh
-rw-rw-r-- 1 henry henry 34 12월 11 16:17 hello.sh
henry@Henry-Desktop:/tmp$ chmod 755 hello.sh
henry@Henry-Desktop:/tmp$ ls -l hello.sh
-rwxr-xr-x 1 henry henry 34 12월 11 16:17 hello.sh
henry@Henry-Desktop:/tmp$ ./hello.sh
Hello, World!
henry@Henry-Desktop:/tmp$
```

# Shell variables (Cont.)



The screenshot shows a terminal window with a dark background. At the top, the window title is "henry@Henry-Desktop: /tmp". Below the title bar, the nano editor interface is visible. The top status bar shows "GNU nano 6.2" on the left and "qa.sh \*" on the right. The main editing area contains the following text:   
#!/bin/bash  
  
echo -n "사용하는 리눅스는 무엇입니까?"  
read linux  
echo "리눅스 : \$linux"  
  
At the bottom of the terminal, there is a row of keyboard shortcuts for nano:   
^G Help      ^O Write Out      ^W Where Is      ^K Cut      ^T Execute  
^X Exit      ^R Read File    ^\ Replace      ^U Paste      ^J Justify

## Shell variables (Cont.)

```
henry@Henry-Desktop: /tmp
henry@Henry-Desktop:/tmp$ ls *.sh
hello.sh  qa.sh
henry@Henry-Desktop:/tmp$ chmod 755 qa.sh
henry@Henry-Desktop:/tmp$ ls *.sh
hello.sh  qa.sh
henry@Henry-Desktop:/tmp$ ./qa.sh
사용하는 리눅스는 무엇입니까? Ubuntu Linux 22.04 LTS
리눅스 : Ubuntu Linux 22.04 LTS
henry@Henry-Desktop:/tmp$
```

# Environment Variables and Subshells :

## `export`

- To make an environment variable, apply the `export` command to a variable you have already defined with shell variable.
- The `export` command instructs the system to define a copy of that variable for each new shell generated.
- Syntax
  - `export [-n] [shell variable]`
  - `-n` : converts environment variable to shell variable.
  - `$ export SOME`
  - `$ export SOME=test`

# Environment Variables and Subshells :

## export

```
henry@Henry-Desktop: /tmp$ echo $SOME
henry@Henry-Desktop: /tmp$ SOME=hello
henry@Henry-Desktop: /tmp$ echo $SOME
hello
henry@Henry-Desktop: /tmp$ export SOME
henry@Henry-Desktop: /tmp$ env > env.file
henry@Henry-Desktop: /tmp$ gedit env.file
```

```
env.file
/tmp
Save
5 XDG_CONFIG_DIRS=/etc/xag/xag-ubuntu:/etc/xag
6 SSH_AGENT_LAUNCHER=gnome-keyring
7 XDG_MENU_PREFIX=gnome-
8 GNOME_DESKTOP_SESSION_ID=this-is-deprecated
9 LANGUAGE=en_US:en
10 LC_ADDRESS=ko_KR.UTF-8
11 GNOME_SHELL_SESSION_MODE=ubuntu
12 LC_NAME=ko_KR.UTF-8
13 SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
14 SOME=hello
15 XMODIFIERS=@im=ibus
16 DESKTOP_SESSION=ubuntu
17 LC_MONETARY=ko_KR.UTF-8
18 GTK_MODULES=gail:atk-bridge
```



# Shell Parameter Variables

- Many of the shell parameter variables automatically defined.
- Assigned initial values by the system.
- When you log in can be changed, if you wish.
- You can obtain a listing of the currently defined shell variables using the **env** command.
- The **env** command operates like the **set** command, but it lists only parameter variables.

# Shell Parameter Variables (Cont.)

Shell Variables	Description
BASH	Full pathname of BASH command
BASH_VERSION	The current BASH version number
GROUPS	Groups to which the user belongs
HISTCMD	Number of the current command in the history list
HOME	Pathname for user's home directory
HOSTNAME	Hostname
HOSTTYPE	Type of machine on which the host runs
OLDPWD	Previous working directory
OSTYPE	Operating system in use
PATH	Pathnames for directories searched for executable commands
PPID	Processes ID for shell's parent shell
PWD	User's working directory
RANDOM	Generated random number when referenced

# Shell Parameter Variables (Cont.)

Shell Variables	Description
SHLVL	Current shell level, number of shells invoked
HOSTFILE	Sets the name of the hosts file, if other than <code>/etc/hosts</code>
CDPATH	Search path for the <code>cd</code> command
EXINIT	Initialization commands for Ex/Vi editor
FCEDIT	Editor used by the history <code>fc</code> command
HISTFILE	The pathname of the history file
HISTSIZE	Number of commands allowed for history
HISTFILESIZE	Size of the history file in lines
IFS	Interfield delimiter symbol
IGNOREEOF	If not set, EOF character will close the shell; can be set to the number of EOF characters to ignore before accepting one to close the shell (default is 10)
INPUTRC	The <code>inputrc</code> configuration file for Readline (command line); default is current directory, <code>.inputrc</code> ; most Linux distributions set this to <code>/etc/inputrc</code>

# Shell Parameter Variables (Cont.)

Shell Variables	Description
KDEDIR	The pathname location for the KDE desktop
LOGNAME	Login name
MAIL	Name of specific mail file checked by Mail utility for received messages, if <b>MAILPATH</b> is not set
MAILCHECK	Interval for checking for received mail
MAILPATH	List of mail files to be checked by Mail for received messages
HOSTTYPE	Linux platforms, such as i686, x86_64, or PPC
PROMPT_COMMAND	Command to be executed before each prompt, integrating the result as part of the prompt
PS1	Primary shell prompt
PS2	Secondary shell prompt
QTDIR	Location of the Qt library (used for KDE)
SHELL	Pathname of program for type of shell you are using

# Shell Parameter Variables (Cont.)

Shell Variables	Description
TERM	Terminal type
TMOUT	Time that the shell remains active awaiting input
USER	Username
UID	Real user ID (numeric)
EUID	Effective user ID (numeric); usually the same as the UID but can be different when the user changes IDs, as with the <b>su</b> command, which allows a user to become an effective root user

# Using Initialization Files

- **.profile** file is an initialization file executed every time you log in.
- It contains definitions and assignments of parameter variables.
- **.profile** file is basically a shell script.
- Can edit with any text editor, such as the **Vi** editor.
- To define or redefine a parameter variable, need to export it to make it an environment variable.
- Must be accompanied by an **export** command.

# Your Home Directory : **HOME**

- Contains the pathname of your home directory.
- Determined by the parameter administrator when account is created.
- The pathname for home directory is automatically read into your **HOME** variable when you log in.

```
$ echo $HOME
```

```
/home/chris
```

```
$ ls $HOME/Downloads
```

# Command Locations : **PATH**

- Contains a series of directory paths separated by *colons*.
- Each time a command is executed, the paths listed in the **PATH** variable are searched one by one.
- The system defines and assigns **PATH** an initial set of pathnames.
- In Linux, the initial pathnames are **/bin** and **/usr/bin**.

```
$ echo $PATH
```

```
/bin:/usr/sbin:
```



## Command Locations : **PATH** (Cont.)

- To add a new directory to your **PATH** permanently, you need to edit your **.profile** file and find the assignment for the **PATH** variable.
- Then, you simply insert the directory, preceded by a *colon*, into the set of pathnames assigned to **PATH**.

# Specifying the BASH Environment :

## **BASH\_ENV**

- Holds the name of the **BASH** shell initialization file to be executed.
- When a **BASH** shell script is executed, the **BASH\_ENV** variable is checked.
- Usually holds **\$HOME/.bashrc**.
- This is the **.bashrc** file in the user's home directory.

# Configuring the Shell Prompt

- The **PS1** and **PS2** variables contain the primary and secondary prompt symbols.
- The primary prompt symbol for the **BASH** shell is a dollar sign (**\$**).
- Can change the prompt symbol by assigning a new set of characters to the **PS1** variable.

```
$ PS1= '->'
```

```
-> export PS1
```

```
->
```

## Configuring the Shell Prompt (Cont.)

- Can change the prompt to be any set of characters, including a string.

```
$ PS1="Please enter a command: "
```

```
Please enter a command: export PS1
```

```
Please enter a command: ls
```

```
mydata      /reports
```

```
Please enter a command:
```

## Configuring the Shell Prompt (Cont.)

- The **PS2** variable holds the secondary prompt symbol.
- Is used for commands that take several lines to complete.
- The default secondary prompt is **>**.
- The added command lines begin with the secondary prompt instead of the primary prompt.

```
$ PS2="@"
```

## Configuring the Shell Prompt (Cont.)

- The **BASH** shell provides a predefined set of codes you can use to configure your prompt.
- Each code is preceded by a **\** symbol.
- The codes must be included within a quoted string.

```
$PS1="\w $"
```

```
/home/dylan $
```

# Configuring the Shell Prompt (Cont.)

Prompt Codes	Description
\!	Current history number
\\$	Use \$ as prompt for all users except the root user, which has the # as its prompt
\d	Current date
\#	History command number for just the current shell
\h	Hostname
\s	Shell type currently active
\t	Time of day in hours, minutes, and seconds
\u	Username

# Configuring the Shell Prompt (Cont.)

Prompt Codes	Description
<code>\v</code>	Shell version
<code>\w</code>	Full pathname of the current working directory
<code>\W</code>	Name of the current working directory
<code>\\</code>	Backslash character
<code>\n</code>	Newline character
<code>\[ \]</code>	Allows entry of terminal specific display characters for features such as color or bold font
<code>\nnn</code>	Character specified in octal format



## Configuring the Shell Prompt (Cont.)

- The default **BASH** prompt is `\s-\v\$` to display the type of shell, the shell version, and the `$` symbol as the prompt.
- In Ubuntu, have changed this to a more complex command consisting of the username, the hostname, and the name of the current working directory.

```
$ PS1="\u@\h:\w$"
```

```
richard@turtle.com:~$
```

## Configuring Your Login Shell : `.profile`

- Is the **BASH** shell's login initialization file:
  - cf) `.bash_profile` in SUSE and Fedora Linux.
- Is a script file that is automatically executed whenever a user logs in.
- Contains shell commands that define system environment variables used to manage your shell.
- They may be either redefinitions of system-defined variables or definitions of user-defined variables.

# Exporting Variables

- Any new parameter variables you may add to the `.profile` file will also need to be exported, using the `export` command.
- Can export several variables in one `export` command by listing them as arguments.

```
export JAVA_HOME=/usr/lib/jvm/jdk1.7.0_25
```

## Manually Re-executing the `.profile` Script

- The `.profile` file is an initialization file that is executed only when you log in.
- To want to take advantage of any changes you make to it without having to log out and log in again, Can re-execute the `.profile` shell script with the dot (`.`) command.

```
$ . .profile
```

- Alternatively, Can use the source command to execute the `.profile`.

```
$ source .profile
```

# System Shell Profile Script

- Linux system also has its own profile file that it executes whenever any user logs in.
- This system initialization file is simply called profile and is found in the `/etc` directory: `/etc/profile`.