

Introduction to UNIX and Shells

Reading: *Unix Power Tools*, Chapter 1; *Learning the Korn Shell*, Pages 1-3, 7-32.

- Why UNIX?
 - A simple and powerful command-line.
 - Fast, high-performance; especially the file-system.
 - Open source. Useful in research, developing specialized systems.
 - Multi-user, preemptive multi-tasking OS.
 - Utilities work well with each other. Multiplies their usefulness.
 - Stable and resilient.
- (Very short) History:
 - Developed starting in 1969 at Bell Labs.
 - 1970s: Gradually grew and evolved — spread into the computer science community.
 - 1980s and 90s:
 - Very popular for software R&D, wide-spread use in academic settings.
 - Commercial development, especially in “enterprise” settings.

- Today:
 - “UNIX” can be legally applied to any system that passes a certification process established by The Open Group.
 - IEEE/ISO POSIX standards facilitate writing software that is portable across a wide range of UNIX (and non-UNIX) systems.
 - Linux keeps getting bigger and better.
 - OS X is UNIX “underneath”.

The Shell.

- Users typically interact with UNIX via a “shell”.
 - “A command-line based environment for execution and control of programs.”
 - “Generally speaking, a shell is any user interface to the Unix operating system, i.e., any program that takes input from the user, translates it into instructions that the operating system can understand, and conveys the operating system’s output back to the user.” — from *Learning the Korn Shell*.
- There are many shells! Common features that all shells have:
 - Command execution.
 - Redirection of input and output.
 - Piping.
 - Wildcard expansion.
 - Process control.
 - Command recall and editing.
 - *Turing-complete*.

The Shell (continued):

- Common shells in use on lectura:

| | |
|---------------------------|-----|
| C Shell (csh) | 64 |
| Enhanced C Shell (tcsh) | 625 |
| Korn Shell (ksh) | 16 |
| Bourne-Again Shell (bash) | 148 |

- There are many folks who have (possibly quite strong) opinions as to which shell is the “best”.
 - I don’t! I have used the C Shell (csh) most of my “Unix career” (since 1986).
 - We will use the Korn shell (ksh) in this class:
 - It is one of the books within *The UNIX CD Bookshelf*.
 - It is available on lectura, the other Fedora Linux machines, OS X, ...
 - Even I should learn something new occasionally...
- Things that we will cover about ksh fall into three categories:
 - Things that work with just about every shell.
 - Things that work with every POSIX-compliant shell.
 - Things that are ksh specific (but have counterparts in other shells).

Command Line Basics.

- Type a command and press <ENTER> (or <RETURN>) to execute the program associated with that command.
 - The output (if there is any) will go to the screen.
 - When the program terminates, the shell will prompt for another command.
- Examples (commands that I typed are in **bold**).

```
$ hostname
lectura.cs.arizona.edu
$ whoami
patrick
$ cal
      January 2006
Su M Tu W Th F S
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

$ date
Thu Jan 12 16:14:19 MST 2006
$ uptime
 4:48pm up 6:08, 39 users, load average: 0.77, 0.54, 0.54
```

Command Line Basics (continued):

- A running instance of a program is called a *process*.
 - A running shell is a process.
 - A shell can start other processes.
- Running the Korn shell:
 - By default, users on lectura are assigned the Bourne Again shell (bash).
 - You can run the Korn shell by typing: **ksh**
 - You will then get a \$ prompt.
 - You can make ksh the default on lectura by choosing it from the web page:
 - <http://www.cs.arizona.edu/people/apply/services.html>
 - Click on the link: Change my default UNIX shell (bash/tsh/ksh).

Accessing the UNIX systems in the CS department:

- From Windows on CS departmental machines:
 - Use Secure Shell Client from the Start menu (or the desktop).
 - It will default to lectura (you can change this to one of the Fedora machines).
 - You will need to fill-in your User Name.
 - Use PuTTY, a free Telnet/SSH Client available at: <http://www.chiark.greenend.org.uk/~sgtatham/putty/>
 - Recommended by Bill Mitchell (who taught 352 in 2004-2005).
- From OS X:
 - Use the Terminal application (found in /Applications/Utilities).
 - Using Terminal->Preferences.... you can change your OS X shell to /bin/ksh.

Running ksh:

- ssh clients emulate “dumb terminals” — simple I/O devices that provide little more than a keyboard and a screen that displays a matrix of fixed-size characters.
- The stty command displays “terminal” settings:

```
$ stty -a
speed 9600 baud; rows 40; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc ixany imaxbel
opost -olcuc -ocnrl onlcr -onocr -onlret -ofill -ofdel n10 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe -echok -echonl -noflsh -xcase -tostop -echoprtr
echoctl echoke
```
- Handy control-characters:
 - ^C (control-C) kills the currently running process.
 - ^U erases the characters typed thus far on the current line.
 - ^W erases the last “word”.
 - ^S suspends output to the terminal; ^Q resumes output.
 - ^O causes output to be discarded until ^O is typed again.
 - ^Z prints “Stopped” and suspends (does not kill) the current process. Execution can be resumed with the **fg** command; **jobs** shows active “jobs”.

More Command Line Basics:

- Most commands accept one or more *operands*:

```
$ cal 8 2006
   August 2006
Su Mo Tu We Th Fr Sa
   1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

$ date
Fri Aug 18 16:07:24 MST 2006
$ date -u
Fri Aug 18 23:07:27 UTC 2006
$ date --utc
Fri Aug 18 23:07:29 UTC 2006
```

More Command Line Basics (continued):

- For many commands, the operands are file names.

```
$ cat Hello.java
public class Hello {

    public static void main(String args[]) {
        System.out.println("Hello, world!");
    } // main

} // class Hello
$ javac Hello.java
$ java Hello
Hello, world!
$ rm Hello.class
$ java Hello
Exception in thread "main" java.lang.NoClassDefFoundError: Hello
    at gnu.java.lang.MainThread.run() (/usr/lib64/libgcj.so.6.0.0)
Caused by: java.lang.ClassNotFoundException: Hello not found in
gnu.gcj.runtime.SystemClassLoader{urls=[file:./],
parent=gnu.gcj.runtime.ExtensionClassLoader{urls=[], parent=null}}
    at java.net.URLClassLoader.findClass(java.lang.String) (/usr/lib64/libgcj.so.6.0.0)
    at java.lang.ClassLoader.loadClass(java.lang.String, boolean) (/usr/lib64/libgcj.so.6.0.0)
    at java.lang.ClassLoader.loadClass(java.lang.String) (/usr/lib64/libgcj.so.6.0.0)
    at java.lang.Class.forName(java.lang.String, boolean, java.lang.ClassLoader) (/usr/lib64/libgcj.so.6.0.0)
    at gnu.java.lang.MainThread.run() (/usr/lib64/libgcj.so.6.0.0)
```

Note the evidence of the "silence is golden" philosophy, which is common in UNIX programs.

More Command Line Basics (continued):

- The **fgrep** command searches for text.
 - Its first argument is a string to search for.
- The remaining argument(s) are the files to search.

```
$ fgrep Hello Hello.java
public class Hello {
    System.out.println("Hello, world!");
} // class Hello
$ fgrep Hello Hello.java Test.java
Hello.java:public class Hello {
Hello.java:    System.out.println("Hello, world!");
Hello.java:} // class Hello
fgrep: can't open Test.java
$ fgrep Waldo Hello.java
$ fgrep world Hello.java
    System.out.println("Hello, world!");
$
```

- Does **fgrep** embrace "silence is golden"?
- Note: **fgrep** is one of a set of tools known as the "**grep** family" — named for **grep**, the first tool in the family to be developed. You will have opportunities to learn more about this family this semester :-).

More Command Line Basics (continued):

- An operand is one type of command line *argument*.
- *Options* are another type of command line argument.
- Options almost always being with a '-' (minus sign).
- By convention, options appear between the command name and the operands (if any).

- Examples:

```
$ date
Fri Aug 18 16:08:50 MST 2006
$ date -u
Fri Aug 18 23:08:52 UTC 2006
$ wc Hello.java
   7   19  144 Hello.java
$ wc -l -w Hello.java
   7   19 Hello.java
$
```

More Command Line Basics (continued):

- In some cases, an option has an associated argument:

```
$ javac -verbose -d work Hello.java
[parsing   Hello.java - #1/1]
[reading   java/lang/Object.class]
[analyzing Hello.java - #1/1]
[reading   java/lang/String.class]
[reading   java/lang/System.class]
[reading   java/io/PrintStream.class]
[writing   Hello.class - #1]
[completed Hello.java - #1/1]
[1 unit compiled]
[1 .class file generated]
$
```

- For most programs the ordering of options is not significant.
 - But, that is a convention, not a rule!

More Command Line Basics (continued):

- It is common to allow single character options to be combined into a single, multi-character option. For example, the following two commands are equivalent:

```
$ wc -l -w Hello.java
 7      19 Hello.java
$ wc -lw Hello.java
 7      19 Hello.java
```

- Some programs have verbose synonyms for single-character options. For example:

```
$ wc --words --lines Hello.java
 7      19 Hello.java
```

- As a rule, whitespace is significant in command lines. For example, the following will not work:

```
$ date-u
ksh: date-u: not found
$ ls -l -a
total 16
drwx----- 3 patrick dept 4096 Jan 6 12:06 ./
drwx----- 15 patrick dept 4096 Jan 6 11:58 ../
-rw-r--r--  1 patrick dept  144 Jan 6 12:00 Hello.java
drwx----- 2 patrick dept 4096 Jan 6 12:06 work/
$ ls -l-a
ls: invalid option -- -
Try `ls --help' for more information.
```

Conventions, not rules!

- There is nothing that prohibits a program from having its own style of argument handling. The `dd` command, a very old file manipulation utility, uses name/value pairs on the command line:

```
dd if=scores.dat ibs=90 skip=40 count=5 of=x
```

More Command Line Basics (continued):

- Java and command-line arguments:

- The `java` command invokes the Java Virtual Machine.
 - The arguments given on the command line with `java` indicate what class contains the `main` method.
 - Additional arguments that appear are passed by the OS to the JVM to pass to `main`.

- Example:

```
$ cat args.java
public class args {

    public static void main(String args[]) {
        for (int i = 0; i < args.length; i++)
            System.out.println("'" + args[i] + "'");
    } // main
} // class args
```

Compile

```
$ javac args.java
$ java args What happened?
'What'
'happened'
'in'
'class'
'today?'
```

Execute

More Command Line Basics (continued):

- Many non-alphanumeric characters have special meaning to shells:

```
$ java args :-)
ksh: syntax error: `)' unexpected
```

- Characters that have special meaning are often called *metacharacters*.
- Here are the Korn shell metacharacters: (Section 1.9, Table 1-6, *Learning the Korn Shell*)

```
~ ` # $ & * ( ) \ | [ ] { } ; ' " ? %
```

- One way to specify an argument that contains metacharacters or whitespace is to enclose the argument in quotes:

```
$ java args '-:-)' '""' 'x' 'y' 'z'z'z'
'-:-)'
'""'
'x'
'y'
'z'z'z'
```

- Note the enclosing quotes are consumed by the shell. The Java code never sees them.
- For now, always use single-quotes.
 - Some metacharacters are still interpreted even when surrounded by double quotes.

More Command Line Basics (continued):

- An alternative to wrapping with quotes is to use a backslash \ to “escape” each metacharacter.
 - The backslash suppresses any meaning associated with metacharacters.
 - You can use the backslash to escape any character, not just metacharacters.
- Example:

```
$ java args :-\ \ \ \'\"\\ x\ y \x\y\z\?
':-)'
':-)'
'x y'
'xyz?'
```
- The ASCII `nu1` character is the only character that cannot be passed in an argument.

More Command Line Basics (continued):

- Multiple commands can be specified on a single command line by separating them with semicolons:

```
$ date -u; cal 1 1950; uptime; wc Hello.java; fgrep "Hello " Hello.java
Fri Aug 18 23:12:32 UTC 2006
January 1950
S M Tu W Th F S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

16:12:32 up 2 days, 23:04, 14 users, load average: 0.27, 0.49, 0.48
7 19 144 Hello.java
public class Hello {
```

- The shell runs each command in turn.
 - The shell waits for each command to terminate before starting the next command.
- The shell does not provide any indication of where the output of one command ends and the output of the next begins!

Command Line Basics — Summary:

- As a rule, command invocations have this form:

```
command-name option1 ... optionN operand1 ... operandN
```
- Options and operands are often collectively referred to as arguments.
- Options typically start with a '-'.
 - Often are single letters.
 - Single letter options can often be combined.
- Options sometimes have arguments themselves.
- The ordering of options is usually not important.
- Many programs allow options to follow operands.
- As a rule, whitespace in options and operands is significant.
- Interpretation of metacharacters can be suppressed by enclosing the argument in quotes or preceding each metacharacter with a backslash.
- There are somewhat firm conventions, but no hard rules about options and operands.
 - You can write tools that ignore the conventions; they will work.
 - Others may get confused/frustrated and not use such tools!

On-line help: man

- The `man` command displays documentation for commands (and more). For example:

```
lectura-> man cal
CAL(1) BSD General Commands Manual CAL(1)

NAME
cal - displays a calendar

SYNOPSIS
cal [-smjy13] [[month] year]

DESCRIPTION
Cal displays a simple calendar. If arguments are not specified, the current month is displayed. The options are as follows:

-1 Display single month output. (This is the default.)
-3 Display prev/current/next month output.
-s Display Sunday as the first day of the week. (This is the default.)
-m Display Monday as the first day of the week.
-j Display Julian dates (days one-based, numbered from January 1).
-y Display a calendar for the current year.
```

On-line help: man (continued):

A single parameter specifies the year (1 - 9999) to be displayed; note the year must be fully specified: "cal 89" will not display a calendar for 1989. Two parameters denote the month (1 - 12) and year. If no parameters are specified, the current month's calendar is displayed.

A year starts on Jan 1.

The Gregorian Reformation is assumed to have occurred in 1752 on the 3rd of September. By this time, most countries had recognized the reformation (although a few did not recognize it until the early 1900's.) Ten days following that date were eliminated by the reformation, so the calendar for that month is a bit unusual.

HISTORY

A cal command appeared in Version 6 AT&T UNIX.

OTHER VERSIONS

Several much more elaborate versions of this program exist, with support for colors, holidays, birthdays, reminders and appointments, etc. For example, try the cal from <http://home.sprynet.com/~cbagwell/projects.html> or GNU gcal.

BSD June 6, 1993 BSD

On-line help: man (continued):

- The **-k** option specifies a keyword to search for in (all) the man page "NAME" entries.
 - For example (**calendar** is a command on Solaris, but it is not the same as the **cal** command)

```
$ calendar 8 2005
/usr/bin/calendar: illegal option -- 8 2005
usage: calendar [ - ]
$ calendar
/usr/bin/calendar: /home/patrick/352/examples/calendar not found
$ huh?
ksh: huh?: not found
$ man -k calendar
cal          cal (1)      - display a calendar
calendar    calendar (1) - reminder service
difftime    difftime (3c) - computes the difference between two calendar times
mktime      mktime (3c) - converts a tm structure to a calendar time
```
- Some man page names appear in more than one section of the manual. The **-s** option to man finds the entry in a specified section.
 - For example, **printf** (only a few of the results are shown):

```
$ man -k printf
printf      printf (1)   - write formatted output
printf      printf (3c)  - print formatted output
printf      printf (3ucb) - formatted output conversion
```
 - **man -s 3 printf**
 - **man -s 1 printf**

Built-in help for commands:

- Many commands have a **--help** option:

```
$ wc --help
Usage: wc [ options ] [file ...]
OPTIONS
-l, --lines    Writes the line counts.
-w, --words    Writes the word counts.
-c, --bytes|chars
               Writes the byte counts.
-m, --multibyte-chars
               Writes the character counts.
```
- Some do not work:

```
$ cal --help
cal: invalid option -- -
usage: cal [-l3smjyV] [[month] year]
$ javac --help
directory does not exist: --help
```

I/O Redirection:

Reading: *Learning the Korn Shell*, pages 18-22.

- Where does the output of a program go?
 - Screen, file, another program, hardware device, process, process on another machine, ...
- Unix uses three standard streams:
 - standard input (**stdin**).
 - standard output (**stdout**).
 - standard error (**stderr**).
- In Java:
 - **System.in** is associated with standard input.
 - **System.out** is associated with standard output.
- By default, the shell starts a program and associates:
 - standard input with the keyboard.
 - standard output with the screen.
 - standard error with the screen.

I/O Redirection (continued):

- Java can read from standard input and write to standard output:

```
import java.io.*;

public class lc {
    public static void main(String args[]) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        int count = 0;
        line = in.readLine();
        while (line != null) {
            count++;
            line = in.readLine();
        }
        System.out.println(count);
    } // main
} // class lc
```

- Examples:

```
$ java lc
something
more happens
here
now
^D
4
```

control-D: used to terminate input on standard input

I/O Redirection (continued):

- It is possible to *redirect* standard input:

```
• Use <file to indicate this.
$ java lc < lc.java
18
$ java lc < Hello.java
7
```

- It is possible to *redirect* standard output:

```
• Use >file to indicate this.
$ java lc > count
something
more happens
here
now
^D
$ cat count
4
```

control-D: used to terminate input on standard input

- Can do both at the same time:

```
$ java lc < Hello.java > count
$ cat count
7
```

- Whitespace before and after < and > is optional:

```
$ java lc<Hello.java>count
$ cat count
7
```

I/O Redirection (continued):

- Important point:

- The shell completely consumes the redirection operators and the file names that follow them!
- The program being run never sees them.

```
$ java args one two three < lc.java four
```

```
'one'
'two'
'three'
'four'
```

```
$ java args one two three < lc.java four > count five
```

```
$ cat count
```

```
'one'
'two'
'three'
'four'
'five'
```

- Re-directions can appear at any point on the command-line (see above).
- What went wrong here?
\$ java > args count two < lc.java three four
Exception in thread "main" java.lang.NoClassDefFoundError: count
- What arguments did the command `java` actually see?

I/O Redirection (continued):

- Many programs will accept input from either standard input or files named on the command line:

```
$ wc Hello.java
7 19 144 Hello.java
```

```
$ wc < lc.java
18 47 404
```

```
$ wc Hello.java lc.java
7 19 144 Hello.java
18 47 404 lc.java
22 60 501 total
```

```
$ wc
```

```
some lines
```

```
here for
```

```
wc to have
```

```
fun counting
```

```
^D
```

```
4 9 44
```

- What happened with the following?

```
$ wc Hello.java < lc.java
7 19 144 Hello.java
```

I/O Redirection (continued):

- Appending to a file: Use '>>' instead of '>'.
 - '>' will truncate a file; that is, if the file exists, its contents will be deleted by the shell. I.e.,

```
$ cat count
'one'
'two'
'three'
'four'
$ wc Hello.java > count
$ cat count
 7      19      144 Hello.java
```
 - To avoid this truncation, you can use '>>' to append the contents to the file

```
$ cat count
'one'
'two'
'three'
'four'
$ wc Hello.java >> count
$ cat count
'one'
'two'
'three'
'four'
 7      19      144 Hello.java
```

I/O Redirection (continued):

- Another example using '>>', and introducing the **echo** command:

```
$ cal 11 2006 > zap
$ echo ..... >> zap
$ cal 12 2006 >> zap
$ cat zap
November 2006
Su Mo Tu We Th Fr Sa
      1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
.....
December 2006
Su Mo Tu We Th Fr Sa
      1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```
- The **echo** command simply prints its argument(s) on standard output.
- Could the first command from the example above be `cal 11 2006 >> zap` instead?

I/O Redirection (continued):

- We can achieve a similar effect by using parentheses to group the commands into a *subshell* that runs each command in turn. The output of the subshell can then be sent to the file `zap`.

```
$ (cal 11 2006; echo .....; cal 12 2006) > zap
$ cat zap
November 2006
Su Mo Tu We Th Fr Sa
      1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
.....
December 2006
Su Mo Tu We Th Fr Sa
      1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```
- What does the following command do?

```
$ cal 11 2006; echo; echo .....; cal 12 2006 > zap
```

I/O Redirection (continued):

- There is a third I/O stream: *standard error*.
- By convention, programs send “normal” output to standard output, and “exceptional” output to standard error.

```
$ cal 2006 11 > zap
cal: bad month
usage: cal [ [month] year ]
$ cat zap
$
```
- Standard output and standard error can be combined with the '2>' redirection operator:

```
$ cal 2006 11 > output 2> errOut
$ cat output
$ cat errOut
cal: bad month
usage: cal [ [month] year ]
$
```
- Other shells (csh, tcsh, bash) will also accept '>&' as the redirection operator:

```
$ bash
bash-2.05$ cal 2006 12 > output >& errOut
bash-2.05$ cat errOut
cal: bad month
usage: cal [ [month] year ]
bash-2.05$
```
- Java note: **System.err** is associated with standard error.

I/O Redirection (continued):

- Major benefit:
 - Programs do not have to include any file-handling code!
 - Can simply be written in terms of standard input, standard output, and standard error.
- Consider an alternative interface:

```
java lc -input file.txt -output count.txt
```

 - Now, *lc.java* will need to include code to find the file names on the command-line, open these files if they are listed, etc.
- Another example:
 - VMS, an operating system developed by DEC for it's VAX line of computers (and still used in some applications!), had output redirection:

```
$ assign/user sys$output outfile
$ run program
```
 - One of the objectives in developing UNIX and its command-line interface was to reduce the amount of typing that was necessary!

Pipes

Reading: *Learning the Korn Shell*, pages 21-22; *Unix Power Tools*, sections: 43.1, 43.2.

- Interprocess communication (IPC): a technique that allows two (or more) processes to exchange data during execution. Examples include:
 - Networking messages (see CSc 425).
 - Semaphores, Monitors (see CSc 452, CSc 422, among others).
- Pipes are another type of IPC that allows the output of one program to be read as the input to the next program.
 - The vertical bar character '|' is used as the pipe symbol.
 - Example: (some lines from the output of **who** were deleted to make this example fit...)

```
$ who
lopa pts/0 2006-08-18 16:21 (freestyle.cs.arizona.edu)
hwang pts/18 2006-08-18 08:30 (c-71-226-32-161.hsd1.ar.comcast.net)
jcropper pts/14 2006-08-16 07:57
patrick pts/20 2006-08-18 08:57 (wolf.cs.arizona.edu)
rgollent pts/21 2006-08-18 10:15 (newbabe.pobox.com)
egkim pts/23 2006-08-16 13:06 (cougar.cs.arizona.edu)
jcropper pts/25 2006-08-17 20:24 (york.cs.arizona.edu)
patrick pts/27 2006-08-18 13:03 (wolf.cs.arizona.edu)
patrick pts/31 2006-08-18 13:03 (wolf.cs.arizona.edu)
$ who | wc -l
14
```

 - How can you determine if **patrick** is logged in?

Pipes (continued):

- Data always flows from left to right in a pipeline.
- The command on the left of the pipe sends its standard output to the pipe.
 - The standard output does not go anywhere else (including not going to the screen).
- The command on the right of the pipe reads its standard input from the pipe.
 - The command on the right does not read standard input from the keyboard.
- Pipes are a key element in UNIX
 - They amplify the usefulness of every program that uses standard input and/or standard output.
 - Any UNIX program should be usable with pipes.
 - Note: I said "should", not "required". Again, conventions, not rules.
 - Programs that read from standard input, transform the input in some way, and send their output to standard output are often called filters.

Pipes (continued):

- The **ps** command will list processes that are running on the machine. With the options **-efa**, the command will list all the processes on the machine (not just mine) and provide a full listing of information about each process.
- I only want information about processes that belong to **apache** (and just who is apache?):

```
$ ps -efa | fgrep apache
apache 2174 2145 0 Aug15 ? 00:00:06 /usr/sbin/httpd
apache 2176 2145 0 Aug15 ? 00:00:06 /usr/sbin/httpd
apache 2177 2145 0 Aug15 ? 00:00:06 /usr/sbin/httpd
apache 2178 2145 0 Aug15 ? 00:00:06 /usr/sbin/httpd
apache 2777 2145 0 Aug15 ? 00:00:06 /usr/sbin/httpd
apache 2778 2145 0 Aug15 ? 00:00:06 /usr/sbin/httpd
apache 2779 2145 0 Aug15 ? 00:00:05 /usr/sbin/httpd
apache 21031 2145 0 Aug15 ? 00:00:07 /usr/sbin/httpd
apache 21036 2145 0 Aug15 ? 00:00:05 /usr/sbin/httpd
apache 17422 2145 0 Aug16 ? 00:00:04 /usr/sbin/httpd
apache 17517 2145 0 Aug16 ? 00:00:07 /usr/sbin/httpd
apache 17745 2145 0 Aug16 ? 00:00:06 /usr/sbin/httpd
apache 17750 2145 0 Aug16 ? 00:00:05 /usr/sbin/httpd
apache 17751 2145 0 Aug16 ? 00:00:05 /usr/sbin/httpd
apache 17753 2145 0 Aug16 ? 00:00:06 /usr/sbin/httpd
apache 19450 2145 0 Aug16 ? 00:00:05 /usr/sbin/httpd
apache 20159 2145 0 Aug16 ? 00:00:05 /usr/sbin/httpd
apache 20160 2145 0 Aug16 ? 00:00:05 /usr/sbin/httpd
apache 20258 2145 0 Aug16 ? 00:00:04 /usr/sbin/httpd
apache 20279 2145 0 Aug16 ? 00:00:06 /usr/sbin/httpd
patrick 6358 25065 0 16:24 pts/27 00:00:00 fgrep apache
```

But, this gives me one line that I do not want!

Pipes (continued):

- **fgrep** has a **-v** option which performs matches but reports on those lines that do not match:

```
$ ps -efa | fgrep apache | fgrep -v patrick
apache 2174 2145 0 Aug15 ? 00:00:06 /usr/sbin/httpd
apache 2176 2145 0 Aug15 ? 00:00:06 /usr/sbin/httpd
apache 2177 2145 0 Aug15 ? 00:00:06 /usr/sbin/httpd
apache 2178 2145 0 Aug15 ? 00:00:06 /usr/sbin/httpd
apache 2777 2145 0 Aug15 ? 00:00:06 /usr/sbin/httpd
apache 2778 2145 0 Aug15 ? 00:00:06 /usr/sbin/httpd
apache 2779 2145 0 Aug15 ? 00:00:05 /usr/sbin/httpd
apache 21031 2145 0 Aug15 ? 00:00:07 /usr/sbin/httpd
apache 21036 2145 0 Aug15 ? 00:00:05 /usr/sbin/httpd
apache 17422 2145 0 Aug16 ? 00:00:04 /usr/sbin/httpd
apache 17517 2145 0 Aug16 ? 00:00:07 /usr/sbin/httpd
apache 17745 2145 0 Aug16 ? 00:00:06 /usr/sbin/httpd
apache 17750 2145 0 Aug16 ? 00:00:05 /usr/sbin/httpd
apache 17751 2145 0 Aug16 ? 00:00:05 /usr/sbin/httpd
apache 17753 2145 0 Aug16 ? 00:00:06 /usr/sbin/httpd
apache 19450 2145 0 Aug16 ? 00:00:05 /usr/sbin/httpd
apache 20159 2145 0 Aug16 ? 00:00:05 /usr/sbin/httpd
apache 20160 2145 0 Aug16 ? 00:00:05 /usr/sbin/httpd
apache 20258 2145 0 Aug16 ? 00:00:04 /usr/sbin/httpd
apache 20279 2145 0 Aug16 ? 00:00:06 /usr/sbin/httpd
```

- Note that **fgrep** appears twice in the command line above.
 - There are two processes running that both are executing the code contained in the **fgrep** executable file.

Pipes (continued):

- Write pipelines to:

1. Count the number of current login sessions for **patrick**.
2. Count the number of words in `/home/cs352/fall06/UnixExamples/lgDictionary.txt` that contain all the vowels.
3. What happens when a process on the right side of a pipe does not use standard input?
4. How many people on lectura use `tsh`? `ksh`? `bash`? `csh`? (Careful, `csh` needs different treatment here...)

Pipes (continued):

- What do each of the following do? Be specific!!!

```
date > out
```

```
> x java lc < lc.java
```

```
cat < lc.java | wc | cat > x
```

```
wc >x lc.java | wc
```

Files and File Management:

Reading: *Learning the Korn Shell*, pages 8-18.

Unix Power Tools, Part III covers these topics in general. Specifically, see sections: 7.1, 8.1 - 8.5, 8.9, 8.11, 10.9, 10.10, 10.12, 14.3 - 14.6

- Filenames.
 - Can consist of any ASCII characters with two exceptions:
 - NUL character (ASCII 0000 0000_{two}).
 - Slash '/'.
 - Maximum length of a filename is platform dependent.
 - Here are some examples (all are valid!):

```
Hello.java
a.out
core
.bashrc
a.b.c.d.
!@##$%^&*()_+*=
A collection of assorted notes about UNIX
:-)
\_ \
(three blanks and two tabs)
...

```
 - Filenames are case-sensitive. **hello.java** and **Hello.java** name two different files.
 - Exception: Mac OS X (but, you do not need to know this for exams/homeworks :-)

Filenames (continued):

- Filenames can contain shell meta-characters and/or whitespace characters.
 - This can cause problems when trying to use commands that need file names(!).
 - Use quotes and/or the backslash character to escape the meta-characters.

```
$ cd ~cs352/fall106/UnixExamples/meta-examples
$ ls
[1] file 42 pipe1|pipe2
$ wc -c 'file 42'
 144 file 42
$ wc -c '[1\]'
 357 [1]
$ wc -c '*'
 357 [1]
 144 file 42
 120 pipe1|pipe2
 621 total
```

Directories and Paths:

- Unix uses *directories* to organize files. Analogous to folders.
- The items (files and/or directories) that are within a directory are each known as *directory entries*.
- Directory names follow the same rules as file names.

• *Path:*

- Describes the location of a file or directory.
- Consists of zero or more directory names, ending with the name of a file or a directory.
- Slashes separate the path components.

• Here are some paths:

```
/home/patrick           my home directory (yours is similar, substitute your login for patrick).
/home/cs352/fall106/UnixExamples/meta-examples
/etc/passwd
/
/cs/www/classes/cs352/fall106  the location of the web pages for this class.
```

Directories and Paths (continued):

- It is not always obvious whether a path specifies a file or a directory.
- If a path starts with a slash '/', it is an *absolute path*.
- The file or directory specified by an absolute path is located by:
 - Start at "the root"; this is the directory named / (that's right, the name is just / Nothing else!)
 - Traverse every directory in the path in turn.
 - /home/cs352/fall106/UnixExamples/sample traverses 5 directories
 - Note: cannot tell just from the pathname if *sample* is a file or directory (it is not one of the 5 directories referred to above!). So, is *sample* a file or a directory?
 - Every file and directory can be reached from the root.
- Traversing a pathname may carry one across disk and machine boundaries.
 - You cannot tell just from looking at the path!
- Examples:
 - *etc/passwd* is on a disk local to lectura
 - *home/patrick* is on a disk that is attached to a different computer
 - I cannot tell this from either name!

Directories and Paths (continued):

- Some examples of places in the file structure on lectura (some of these are "common" or "normal" on other versions of Unix as well).
 - **/usr/bin** Location of standard programs such as **cal**, **man**, and **wc**.
 - **/usr/local** Location of software not supplied by default with Fedora. **acroread**, a pdf reader.
 - **/usr/lib** Location of code and data.
 - **/usr/lib/libm.a** Collection of math routines for C.
 - **/usr/lib/perl5** Directory containing version 5.8.6 of **perl**.
 - **/tmp** Temporary files. Anybody can create a file or directory here. Try it!
 - **/etc** Files related to system administration.
 - **/etc/passwd** List of all users on the system.
 - **/etc/mail/** various files used by mail.
 - **/cs/www** The web files that are read by the web server that handles all www.cs.arizona.edu webpages.

Directories and Paths (continued):

- Every process has a *current working directory*.
 - The `pwd` command will display this directory:

```
$ pwd
/home/patrick
```
- The `cd` command is used to change the current directory of the shell:

```
$ pwd
/home/patrick
$ cd /home/cs352/fall106
$ pwd
/home/cs352/fall106
```
- When you (or any user) first log in, the current directory of the shell is set to the user's *home directory*.
 - The shell finds the home directory in the `/etc/passwd` file (use `fgrep` on this file to find your entry).
 - With no arguments, `cd` changes to the home directory.

```
$ pwd
/home/cs352/fall106
$ cd
$ pwd
/home/patrick
```
- Side note: On CS department Windows machines, a user's UNIX home directory is mapped to `h:`.

Directories and Paths (continued):

- *Relative path*: a path that does not start with a slash.
 - Interpreted relative to the current directory of the process.

```
$ cd /home/cs352/fall106
$ pwd
/home/cs352/fall106
$ cd UnixExamples
$ pwd
/home/cs352/fall106/UnixExamples
$ cd meta-examples
$ pwd
/home/cs352/fall106/UnixExamples/meta-examples
```
 - How does a process started by the shell “know” where it is?
 - It “inherits” the current working directory from the shell.

```
$ pwd
/home/cs352/fall106/UnixExamples/text-files
$ wc gettysburg.txt sawyer.txt
 23 280 1495 gettysburg.txt
8303 71910 400480 sawyer.txt
8326 72190 401975 total
$ wc /home/cs352/fall106/UnixExamples/text-files/gettysburg.txt /home/cs352/fall106/
UnixExamples/text-files/sawyer.txt
 23 280 1495 /home/cs352/fall105/assign1/sample/gettysburg.txt
8303 71910 400480 /home/cs352/fall105/assign1/sample/sawyer.txt
8326 72190 401975 total
```

Directories and Paths (continued):

- Two special directory entries appear in every directory: `.` (read as “dot”) and `..` (read as “dot dot”).
- `.` specifies the current directory. (We'll see more uses of this later.)

```
$ pwd
/home/cs352/fall106/UnixExamples/text-files
$ wc gettysburg.txt
 23 280 1495 gettysburg.txt
$ wc ./gettysburg.txt
 23 280 1495 ./gettysburg.txt
```
- The name `..` refers to the *parent* of the current directory.

```
$ pwd
/home/cs352/fall106/UnixExamples/text-files
$ cd ..
$ pwd
/home/cs352/fall106/UnixExamples
$ cd ..
$ pwd
/home/cs352/fall106
$ cd UnixExamples/text-files
$ wc sawyer.txt ../Hello.java
8303 71910 400480 sawyer.txt
 7 19 144 ../Hello.java
8310 71929 400624 total
```

 - Note: `../Hello.java` refers to the file named `Hello.java` that is in the “directory above” or the *parent* of the current directory.

ls command:

- The `ls` command (that is the letter l, not the number 1).
- Lists the contents (names of the files and directories) of a directory.
 - By default, it operates on the current directory.

```
$ pwd
/home/cs352/fall106/UnixExamples
$ ls
args.java Hello.java lc.java meta-examples sample text-files
$ ls text-files
gettysburg.txt sawyer.txt
```
 - The `-l` (the letter l, not the number 1) option provides a *long* listing:

```
$ pwd
/home/cs352/fall106/UnixExamples
$ ls -l
total 108
-rw-rw-r-- 1 patrick cs352f06 199 Aug 18 13:04 args1.java
-rw-rw-r-- 1 patrick cs352f06 193 Aug 18 13:04 args.java
-rw-rw-r-- 1 patrick cs352f06 191 Aug 18 13:04 bad.java
-rw-rw-r-- 1 patrick cs352f06 54 Aug 18 13:04 classfiles
-rwxrwxr-x 1 patrick cs352f06 169 Aug 18 13:04 countBoth
-rwxrwxr-x 1 patrick cs352f06 133 Aug 18 13:04 countBoth2
```

← total of 28 1-K blocks of storage

| | | | | | | |
|-------------|------------|-------|-------|------|-------------------------|----------|
| ↑ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| permissions | link count | owner | group | size | last modified date/time | filename |

ls command (continued):

- **ls** has many options (perhaps too many?). The **-t** option will sort files by last date modified, most recent first:

```
$ pwd
/home/cs352/fall06/UnixExamples
$ ls -l
total 108
-rw-rw-r-- 1 patrick cs352f06 199 Aug 18 13:04 args1.java
-rw-rw-r-- 1 patrick cs352f06 193 Aug 18 13:04 args.java
-rw-rw-r-- 1 patrick cs352f06 191 Aug 18 13:04 bad.java
-rw-rw-r-- 1 patrick cs352f06 54 Aug 18 13:04 classfiles
-rwxrwxr-x 1 patrick cs352f06 169 Aug 18 13:04 countBoth*
-rwxrwxr-x 1 patrick cs352f06 133 Aug 18 13:04 countBoth2*
```

- If files or directories are specified on the **ls** command line, then **ls** operates on them instead of all:

```
$ pwd
/home/cs352/fall06/UnixExamples/text-files
$ ls -l sawyer.txt ../Hello.java
-rw-rw-r-- 1 patrick cs352f06 144 Jan 12 16:25 ../Hello.java
-rw-rw-r-- 1 patrick cs352f06 400480 Jan 16 17:47 sawyer.txt
```

- Files and directories whose names start with **.** are "hidden"; **ls** will not display them unless **-a** is provided.

```
$ wc lc.java > .output
$ ls
args.java Hello.java lc.java meta-examples sample text-files
$ ls -a
. args.java lc.java .output text-files
.. Hello.java meta-examples sample
```

ls command (continued):

- **ls -R** can be used to recursively list the contents of directories:

```
$ pwd
/home/cs352/fall06
$ ls
Cexamples UnixExamples
$ ls -R UnixExamples
.:
Cexamples UnixExamples

./UnixExamples:
args.java Hello.java lc.java meta-examples sample text-files

./UnixExamples/meta-examples:
[1] file 42 pipe1|pipe2

./UnixExamples/text-files:
gettysburg.txt sawyer.txt
```

Note: Only some of the results are shown here to make things fit on the slide. Try this yourself to see the full listing of the contents of /home/cs352/fall06

ls command (continued):

- Read the man page for **ls** to learn more about options.

- The **-F** option is particularly useful!

```
$ pwd
/home/cs352/fall06/UnixExamples
$ ls -F
args.java Hello.java lc.java meta-examples/ sample text-files/
```

- The **-c**, **-1** (one), **-u**, **-s** options are also worth reading about.

- What is happening here?

```
$ pwd
/home/cs352/fall06/UnixExamples
$ ls -l | wc
  33    290   1724
```

mkdir command:

- Use **mkdir** to make one (or more) directories:

```
$ ls -F
args.java Hello.java lc.java meta-examples/ sample text-files/
$ mkdir patrick
$ mkdir abc zap
$ ls -F
abc/      Hello.java meta-examples/ sample zap/
args.java lc.java patrick/      text-files/
```

- **mkdir** will not create intermediate directories unless you supply **-p** to tell it to do so:

```
$ ls
args.java Hello.java lc.java meta-examples sample text-files
$ mkdir patrick/layer/two
mkdir: patrick/layer/two: [No such file or directory]
$ mkdir -p patrick/layer/two
$ ls -RF
.:
args.java Hello.java lc.java meta-examples/ patrick/ sample text-files/

./meta-examples:
[1] file 42 pipe1|pipe2

./patrick:
layer/

./patrick/layer:
two/

./patrick/layer/two:

./text-files:
gettysburg.txt sawyer.txt
```

Copying (cp), deleting (rm), and moving/rename (mv) files:

- Copying files with **cp**:

- **cp** copies one file to another, or will copy one or more files to a directory:

```
$ ls -lF
total 0
$ echo "some text for the file" > one.txt
$ cp one.txt two.txt
$ cp one.txt three.txt
$ ls -lF
total 0
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:15 one.txt
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:15 three.txt
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:15 two.txt
```

Copying (cp), deleting (rm), and moving/rename (mv) files (continued):

- Copying files with **cp** (continued):

- To copy one or more files to a directory, you provide the names of the files with the directory name last:

```
$ ls -lF
one.txt three.txt two.txt
$ mkdir layer
$ cp one.txt two.txt layer
$ ls -lFR
.:
total 4
drwxr-xr-x 2 patrick dept 4096 Jan 16 18:17 layer/
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:15 one.txt
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:15 three.txt
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:15 two.txt

./layer:
total 0
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:17 one.txt
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:17 two.txt
```

- What does the following command do?

```
cp /etc/passwd /usr/bin/cal ../lc.java .
```

Copying (cp), deleting (rm), and moving/rename (mv) files (continued):

- Copying files with **cp** (continued):

- The **-i** option prevents automatically overwriting an existing file. It will prompt before replacing.

```
$ ls -lF
total 4
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:22 one.txt
-rw-rw-r-- 1 patrick dept 69 Jan 16 18:23 three.txt
-rw-rw-r-- 1 patrick dept 46 Jan 16 18:22 two.txt
$ cp one.txt two.txt
$ ls -lF
total 4
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:22 one.txt
-rw-rw-r-- 1 patrick dept 69 Jan 16 18:23 three.txt
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:23 two.txt
$ cp -i one.txt three.txt
cp: overwrite 'three.txt'? n
$ ls -lF
total 4
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:22 one.txt
-rw-rw-r-- 1 patrick dept 69 Jan 16 18:23 three.txt
-rw-rw-r-- 1 patrick dept 23 Jan 16 18:23 two.txt
```

two.txt was replaced with a copy of *one.txt*

three.txt was not replaced

Copying (cp), deleting (rm), and moving/rename (mv) files (continued):

- Copying files with **cp**, continued:

- The **-r** option can be used to recursively copy directory contents to a new location:

```
$ ls -lRF
.:
another/ one.txt
./another:
two.txt zap-sub/
./another/zap-sub:
three.txt
$ cp -r another zap
$ ls -lRF
.:
another/ one.txt zap/
./another:
two.txt zap-sub/
./another/zap-sub:
three.txt
./zap:
two.txt zap-sub/
./zap/zap-sub:
three.txt
```

- Another useful option is **-p**, which will preserve the modification times for the files being copied.

Copying (cp), deleting (rm), and moving/rename (mv) files (continued):

- Deleting files with **rm**:

- **rm** (remove) will permanently delete one or more files:

```
$ ls -F
another/ one.txt zap/
$ rm one.txt
$ ls -F
another/ zap/
```

- To remove a directory, use **rmdir**.

- Note that the directory must already be empty (no entries in it).

```
$ ls -F
another/ zap/
$ rmdir zap
rmdir: directory "zap": Directory not empty
$ ls -Fa zap
./          ../          two.txt
$ rm zap/two.txt
$ rmdir zap
$ ls -F
another/
```

- The **-i** option of **rm** prompts before file is removed.

- This is a most useful option!

```
$ ls -F
another/ three.txt two.txt
$ rm -i *
rm: another is a directory
rm: remove three.txt (yes/no)? y
rm: remove two.txt (yes/no)? n
$ ls -F
another/ two.txt
```

Copying (cp), deleting (rm), and moving/rename (mv) files (continued):

- Moving/rename files with **mv**:

- Used to rename a file or directory.

- General form: **mv old-name new-name**

```
$ mv args.java ARGS.java
$ mv lc.java lc.java.keep
```

- UNIX makes the change by deleting the directory entry for the old name and making an entry for the new name. The file itself does not change!

- **mv** can also be used to move one or more files (and/or directories) between directories:

```
mkdir docs
mv index.html LectureSchedule.html docs
mv ProjectSchedule.html /home/patrick/352
mv /home/cs352/fall106/assign1 /home/patrick/352
```

- What did this last command do?

Copying (cp), deleting (rm), and moving/rename (mv) files (continued):

- Moving/rename files with **mv** (continued):

- Can handle cross-device moves.

- In this case, the move is not just a renaming of a directory entry.

- A copy of the file is made in the new location.

- The old file is then deleted.

```
$ mkdir /tmp/patrick
$ mv args.java /tmp/patrick
```

- Anyone can create files and/or sub-directories in **/tmp**.

- **/tmp** is on a disk that is local to lecturera.

- Home directories (such as **/home/patrick**) are located on one of the CS department's NetApp file servers.

- **mv** will overwrite the destination file. It will do this silently (without warning!).

- The **-i** option will cause **mv** to query before overwriting an existing file.

```
$ mv -i temp /cs/www/classes/cs352/fall106/index.html
mv: overwrite /cs/www/classes/cs352/fall106/index.html (yes/no)? n
```

Tilde (~) substitution:

- The home directory is often used. Either as the start of a pathname, or because the user wishes to return to it.

- The shell treats the tilde character, **~**, as a shorthand for the user's home directory.

```
$ pwd
/home/patrick/352/examples
$ java args ~
'/home/patrick'
```

- The shell will also expand **~user** to the home directory of the specified user:

```
$ pwd
/home/cs352/spring06/UnixExamples
$ cd ~
$ pwd
/home/patrick
$ cd ~cs352/fall106/UnixExamples
$ pwd
/home/cs352/fall106/UnixExamples
$ cd ~patrick
$ pwd
/home/patrick
```

Tilde '~' substitution (continued):

- Another example, and note how **echo** and another command can be combined to see the results of filename substitution: (The command is not executed; **echo** simply prints what will be seen by the command.)

```
$ echo cp -cs352/fall106/UnixExamples/args.java .
cp /home/cs352/fall106/UnixExamples/args.java .
```
- The tilde substitution is done by the shell, *not* by the operating system, and *not* by the command being executed.
- A program may, or may not, be coded to support shell-like tilde expansion — some do, some do not.
- The **java.io.File** class does not handle tilde:

```
$ ls -l /home/cs352/fall106/UnixExamples/args.java
-rw-rw-r-- 1 patrick cs352 193 Jan 12 16:25 /home/cs352/spring06/UnixExamples/args.java
$ cat tilde.java
import java.io.*;

public class tilde {
    public static void main(String args[]) {
        System.out.println( new File("/home/cs352/fall106/UnixExamples/args.java").exists() );
        System.out.println( new File("~/cs352/fall106/UnixExamples/args.java").exists() );
    } // main
} // class tilde
$ javac tilde.java
$ java tilde
true
false
```

Wildcards:

- Wildcards allow the user to specify files and directories using textual patterns.
- The question mark, **?**, meta-character:
 - Matches any one character.
 - **?** Matches one-character names. I.e.,

```
a B : - z
```
 - **a?c** Matches names that are three-characters long, starting with **a** and ending with **c**. I.e.,

```
abc axc aac acc a-c a+c a?c
```
 - What would be matched by the following?

???

a??b

???.?

Wildcards (continued):

- When wildcard meta-characters are present on a command-line, the shell replaces them with a list of file names in the current directory that match the specified pattern.

```
$ ls
42 42.7 7 args.class bat z za
$ echo ?
7 z
$ echo ??
42 za
$ echo ???
bat
$ echo ????
42.7
$ echo ?? ???
42 za bat
$ java args ??
'42'
'za'
$ ls -l ??
-rw-rw-r-- 1 patrick 119 0 Aug 18 13:04 42
-rw-rw-r-- 1 patrick 119 0 Aug 18 13:04 za
```
- Wildcard expansion: replacing an argument containing wildcard meta-character(s) with file names.
- Note (again): the shell handles wildcard expansion. It is not done by individual programs.

Example directory:
~cs352/fall106/UnixExamples/question-mark

Wildcards (continued):

- If a wildcard argument matches no files, it is passed unchanged to the program:

```
$ ls
42 42.7 7 args.class bat z za
$ java args ?? ab? ?
'42'
'za'
'ab?'
'7'
'z'
```
- Hidden files (recall: those that start with a dot **.**) are not matched by a wildcard.
 - Unless the pattern starts with a dot:

```
$ echo "something to put in the file" > .abc
$ java args ????
'42.7'
$ java args .???
'.abc'
```

Wildcards (continued):

- The asterisk, *, meta-character matches any sequence of characters, including an empty sequence.
 - * Matches every (non-hidden) name.
 - *.java Matches every name that ends with .java.
 - *x* Matches every name that contains an x. I.e., x x0xx xxx abxcxd a+x
 - .* Matches every hidden name, including . and ..
- What would be matched by the following?
 - *old.*
 - *x*y
 - *,*
- I use Keynote (a Macintosh presentation tool) to prepare the slides for class. I then generate two pdf files for each set of lecture notes (4 slides to a page and 2 slides to a page). What would the following command do?
`ls -lt l*pdf`

Wildcards (continued):

- Wildcards can be (and often are) combined. Examples:
 - ??* Matches names that are two or more characters long.
 - *.* Matches names whose next to last character is a dot.
- What would be matched by the following?
 - ?x?*
 - *-?-*
 - *-
- Square brackets:
 - Use when a single character can be any one of a set of characters:

| | |
|-----------------|--|
| [a-z] | Matches names that consist of a single lowercase letter. |
| *.[hcm] | Matches names that end in .h, .c, or .m |
| [A-Za-z]*.[0-9] | Matches names that start with a letter (uppercase or lowercase) and ends with a dot followed by a digit. |
| *[!0-9] | Matches names that end with a non-digit character. Can also use: *[^0-9] |
| [Tt]ext | Matches Text and text |

Wildcards (continued):

- Problems:
 - How many (at most) files could be matched with this pattern: `[csc][35][2]`
 - Move all files whose names start with a lower case vowel into a directory named **v**.
 - Move all files whose names start with a lower case consonant into a directory named **c**.
- Slashes can be included in a pattern to match files elsewhere than the current directory:
 - `wc ~/352/*.java`
 - `wc` will use every *java* file in my *352* directory.
- Problems: What will each of the following do?
 - `ls */TreeWalker.java`
 - `cat *.java */*.java */**/*.java > jsrsc`
 - `ls -ld /?????/?`
 - `echo /*/`
- ? * [] are the most commonly used wildcards. (There are others...)

Introduction to Vi

Reading: *Learning the Korn Shell*, Chapter 2. Especially pages 33-36, 47-60. Pages 47-60 are the most useful. *Learning the vi Editor*, Part I: Basic and Advanced vi. Especially, chapters 1, 2, 3.

- There have been four prominent and widely popular UNIX editors.
 - **ed** is the original UNIX editor.
 - Line-oriented, terse, elegant.
 - ed, or a lookalike, is on most UNIX systems.
 - **vi**, short for visual, created by Bill Joy in 1976.
 - Screen-oriented and “modal”.
 - Arguably the fastest plain-text editor for touch-typists.
 - Based on **ex**.
 - An improved version of **ed**.
 - **Emacs**, short for editor macros.
 - GNU Emacs, 1984-85, by Richard Stallman is the most commonly used version.
 - Extensible through a built-in Lisp interpreter.
 - Why **vi**?
 - Every UNIX user needs to know one editor.
 - Patrick knows **vi** way better than he knows **emacs** :-).

There is a tutorial for **vi**.
Do the following to use the tutorial:
`cp ~cs352/fall106/vi.tut .`
`vi vi.tut`

vi Introduction:

- **vi** is “modal”. There are two modes:
 - Insert mode: Anything typed becomes text in the document.
 - Command mode: Everything else except putting text in the document.
 - Saving, moving within the file, deleting, searching, replacing, etc.
- Starting **vi**:
 - vi**
 - Will start **vi** with an empty screen. File is not named, but can be named when saving.
 - vi filename**
 - Will start **vi** and open the specified file.
 - Will display first part of file with the cursor on the first character of the file.
 - If filename does not exist, **vi** will create an empty file and display an empty screen.
 - vi +45 filename**
 - Will start **vi** and open the specified file.
 - Will put the cursor on the specified line of the file, 45 in this case.
- **vi** always starts in Command Mode.

vi Introduction (continued):

- Versions of **vi**:
 - Most of what I will put in the slides will work on any version of **vi**.
 - One example of a different version of **vi** is **vim**, short for vi improved.
 - **vim** is the default version of **vi** on the department’s Fedora machines.
 - **vim** is the default version of **vi** on Mac OS X.
 - **vim** is the default version of **vi** on lectura.

Insert mode:

- Simpler of the two modes.
- Allows you to type text.
- The backspace key can be used to correct mistakes.
- Arrow keys may, or may not, work in insert mode. Depends on the version of **vi**.
 - **vim** allows the arrow keys to work when in insert mode.
- **ESC** (the escape key): press this to go from insert mode to command mode.

Command mode:

- Basic navigation:
 - Short-range movement:
 - **Arrow keys** to move up, down, left, right.
 - **h, j, k, l** keys:
 - **h** and **l** move left and right, respectively.
 - **j** and **k** move down and up, respectively.
 - Very useful when touch-typing!
 - **0** (the number zero) will move to the first character on the current line.
 - **\$** will move to the last character on the current line.
 - **^** will move to the first non-whitespace character on the current line (compare this move with 0).
 - **w** will move forward to the beginning of the next word.
 - **W** will move forward to the beginning of the next word, punctuation characters not considered.
 - **b,B** same as **w,W** but will move backward instead of forward.

Command mode (continued):

- Basic navigation (continued):
 - Longer-range moves:
 - **^D, ^U** move cursor down or up.
 - Default move is half the vertical lines of the current screen size.
 - A number before the **^D** or **^U** will move that many lines and will change the default.
 - **G** move to the last line of the file.
 - **245G** move to line 245 in the file. **1G** will move to the first line in the file.
 - Where are you in the file?
 - **^G** will show: current line number, total lines in the file, whether file has been modified, filename.
 - **vim** will also show the character number.

Command mode (continued):

- Entering insert mode, listed in order of usefulness:
 - **i** insert, editing will start at the cursor's location.
 - **I** insert, editing will start at the beginning of the line containing the cursor.
 - **a** append, editing will start just after the cursor's position.
 - **A** append, editing to start at the end of the line containing the cursor.
 - **o** (lowercase letter "oh") create a new line below the cursor, editing will start at beginning of the new line.
 - **O** (uppercase letter "oh") create a new line above the cursor, editing will start at beginning of the new line.

Command mode (continued):

- Deleting text:
 - **x** delete the character under the cursor. Text to the right of the cursor shifts left.
 - **X** delete the character to the left of the cursor. Text from cursor to the end of the line shifts left.
 - **dd** delete the line, including the newline character. Text below will shift up.
 - **dw** delete characters up to the start of the next word.
 - **dW** delete characters up to the start of the next word. Will delete punctuation characters as well.
 - **d\$** delete characters up to the end of the current line. Will not delete the newline character.
- Saving, quitting, saving-and-quitting.
 - **ZZ** Save current contents of the file to the default filename and quit **vi**.
 - **:w** save the current contents of the file. Does not quit **vi**.
 - **:w filename** save the current contents of the file to *filename*. *filename* becomes the default name for future saves. Note: there is a blank space between **:w** and **filename**. Does not quit **vi**.
 - **:w!filename** overwrite an existing file with the current contents of the file. Does not quit **vi**.
 - **:q** quit. Will not allow quitting if file has unsaved changes.
 - **:q!** quit. Will not warn about unsaved changes. Unsaved changes are (quietly) lost.
 - **:wq** Save current contents of the file to the default filename and quit **vi**.
 - Note: Commands that start with **:** are commands to the **ex** editor. **vi** is a superset of the **ex** editor.

Command mode (continued):

- Changing text (continued):
 - **r** followed by a character: replace the character under the cursor with that character.
 - does not enter insert mode.
 - **R** replace characters. Enters insert mode, but will overwrite characters. Press **ESC** when done.
 - **s** replace the character under the cursor with a string; that is, **vi** deletes the character under the cursor and then enters insert mode. Press **ESC** when done.
 - **S** delete the entire line under the cursor, leaving only the newline. Enters insert mode with the cursor at the beginning of the now blank line.
 - **cw** delete the current word (compare with the **dw** command) and enter insert mode.
 - **cW** delete the current word including punctuation (compare with the **dw** command) and enter insert mode.
 - **c\$** delete characters to the end of the current line (compare with the **d\$** command) and enter insert mode.

Command mode (continued):

- Undo and Redo
 - **u** Undoes the last edit (insert, delete) performed.
 - In **vi**, **u** will toggle between undo and redo: alternates between undoing the last command and redoing the last command.
 - In **vim** (Fedora, OS X), **u** will undo multiple commands.
 - In **vim** (Fedora, OS X), **^R** will redo multiple commands.
- Repeat last edit:
 - **.** (period) Will repeat the last edit (insert, delete) performed.
 - Can be used multiple times.
 - Example: Want to indent a set of lines an additional 3 spaces each:
 - Use **^I** to insert the three spaces on one of the lines.
 - Use arrow keys (or j,k keys) to move to next (previous) line, and **.** to repeat command on each line.

Command mode (continued):

- Searching:
 - **/** Will cause a prompt to appear at the bottom on the screen (on the status line). Enter the text to search for and press return.
 - Searches forward in the file from the current cursor location for the next occurrence of the search text. Puts cursor at beginning of the text it finds.
 - Will wrap around bottom of file. Will report wrap-around on status line.
 - Will beep (or flash screen) if text is not found in the file.
 - **?** Same as **/** but will search backwards in the file.
 - Will wrap around the top of the file.
 - **/** and **?** can be used without a search string.
 - This will search using the previous search string.
 - Will print an error message on the status line if there is no previous search string.
 - **n** Will repeat the previous search. Uses the previous search string, and searches in the previous direction.

Command mode (continued):

- Yanking and pasting (cutting and pasting):
 - **yy** or **Y** “Yank” the current line. Puts a copy of the current line into an internal buffer.
 - Does not change the current line.
 - Can precede **yy** or **Y** with a number: will “yank” that many lines, starting at current line.
 - **p** Pastes the contents of the buffer after the cursor.
 - If the contents are line(s), the paste will create a new line below the cursor.
 - **P** Similar to **p**, but will paste before (or above) the cursor.
- Any deleted text (using any of the delete edit commands) is also placed in an internal buffer.
 - Can then be pasted using **p** or **P**.
 - Example: if **dw** is used to delete a word, you can navigate to a new location and use **p** to paste the word after the cursor (or **P** to paste the word before the cursor).
- Up to 9 internal buffers are available in vi to hold lines of text.
 - You can yank lines to a specific buffer, and paste from a specific buffer.
 - See Section 4.3 of *Learning the vi Editor* for more details.

Editing the ksh command-line with vi:

Using vi to edit the command-line in ksh:

- Add the following line to your *.profile* in your home directory:
`VISUAL=$ (whence vi)`
 - Filename completion:
 - Insert mode: TAB to complete the name
 - Command mode: \ to complete the name.
- The command line is then automatically in insert mode.
- Use **ESC** to change to command mode to edit the current line.
- To scroll backwards through previous commands:
 - Press **ESC** to enter command mode.
 - Use the **k**, **-**, or up-arrow keys to scroll backwards through previous commands.
 - Use the **j**, **+**, or down-arrow keys to scroll forward.
 - Can prefix **kj**, **+** with a number to indicate distance to scroll (does not work with arrow keys).
- To edit a particular line when you get to it:
 - You are already in command mode.
 - Move (using the **h** and **l**, or left- and right-arrow keys) to position the cursor.
 - Use the various edit commands. Examples include using **i** or **a** to enter insert mode, **x** to delete a char, **dw** to delete a word, **cw** to change a word (and enter Insert mode), etc.
- When ready to execute the line type 'Return'.
- Use **^c** to cancel (not execute the current command) and clear the command line.

Shell Script Basics

Reading: *Learning the Korn Shell*, pages 91-105 (sections 4.1, 4.2).

- A *shell script* is simply a file that contains a series of shell commands.
- Example:

```
$ cat ucount
#!/bin/ksh
echo -n "Current logins:"
who | wc -l | tr -s " "
```
- One way to run a script: run **ksh** with the script as an argument:

```
$ ksh ucount
Current logins: 51
```

 - Another way: use **.** and the script:

```
$ . ucount
Current logins: 51
```
- It would be nice to only type **ucount**, but that produces an error:

```
$ ucount
ksh: ucount: cannot execute
```
- The problem is that the permissions of **ucount** do not indicate that it is executable.
 - The **chmod** command adjusts permissions:

```
$ ls -lF ucount
-rw----- 1 patrick cs352f06 61 Aug 18 13:04 ucount
$ chmod u+x ucount
$ ls -lF ucount
-rwx----- 1 patrick cs352f06 61 Aug 18 13:04 ucount+
$ ucount
Current logins: 16
```

Shell Script Basics (continued):

- The **tr** command can be used to translate characters. In the example from the previous page:

```
tr -s " "
```

 - Will read its standard input. Any character listed in the set specified between the quotation marks will be reduced from a multiple character sequence to a single character. Thus, the command will truncate a run of blank spaces to a single blank space. For example:

```
$ ls -lF ucount
-rwx----- 1 patrick dept          49 Sep 13 09:33 ucount*
$ ls -lF ucount | tr -s " "
-rwx----- 1 patrick dept 49 Sep 13 09:33 ucount*
```
 - See the manpage for other **tr** features.
- The **echo** command has both built-in and stand-alone versions. Most shells (including **ksh**) use a built-in version. The manpage for **echo** explains both the built-in and stand-alone versions.

```
echo -n Current logins:
```

 - Does not print a newline character.

Scripts and I/O Streams:

- Redirecting a script's standard output produces a catenation of standard output of all the commands in the script:

```
$ cat ucount
#!/bin/ksh
echo -n "Current logins:"
who | wc -l | tr -s " "
$ ucount > u.out
$ cat u.out
Current logins: 57
```

 - The file *u.out* ends up with the output of each command in turn.
- Redirecting the output of a script does not affect output redirections inside the script.
 - Here is (an oddly) divided version of **ucount**:

```
$ cat ucount-odd
#!/bin/ksh
echo -n "Current logins:"
who | wc -l > odd.out
tr -s " " < odd.out
$ ucount > u.out
$ cat u.out
Current logins: 55
```
 - Questions:
 - What does **ucount-odd** put in the file *odd.out*?
 - Is the file *odd.out* still present after **ucount-odd** finishes executing?

Scripts and I/O Streams (continued):

- Programs run inside a script “inherit” the standard input stream of the script:

```
$ cat countChars
#!/bin/ksh
wc -c
$ countChars < args.java
193
$ cal | countChars
142
```

 - Here is a trivial script that avoids the nuisance of having to type **java** when running the **lc.java** utility:

```
$ cat lc
#!/bin/ksh
java lc
$ lc < args.java
8
$ who | lc
55
```
- Command-line arguments, however, are not “inherited”:

```
$ java args one two three
'one'
'two'
'three'
$ cat args
#!/bin/ksh
java args
$ args one two three
$
```

Scripts and I/O Streams (continued):

- A way to “capture” standard input and give it to two (or more) commands:

```
$ cat countBoth
#!/bin/ksh
cat > .countBoth.tempfile
echo -n "wc says:"
wc -l < .countBoth.tempfile | tr -s " "
echo -n "lc says: "
java lc < .countBoth.tempfile
rm .countBoth.tempfile
$ cal | countBoth
wc says: 8
lc says: 8
```

 - **cat** is used to create a temporary file.
 - The temporary file is then used as re-directed standard input to both **wc** and **java lc**.
 - What might go wrong with such a simple-minded use of a temporary file?

Variables:

- **ksh** (and all shells) provide for variables in a shell script.
- As an example, here is **countBoth2** that uses a variable to avoid repetitious (and possibly erroneous) specification of the temporary file name:

```
$ cat countBoth2
#!/bin/ksh
tmp=.countboth.tmpfile
cat > $tmp
echo -n "wc says:"
wc -l < $tmp | tr -s " "
echo -n "lc says: "
java lc < $tmp
rm $tmp
$ who | countBoth2
wc says: 61
lc says: 61
```

- Variables come into existence when created. Note there was no declaration of **tmp** in **countBoth2**.
- When assigning a value to a variable, do not put whitespace around the = sign.
- **\$** followed by characters indicates that a variable's value should be substituted at that point.
- If whitespace is part of the value to be assigned to a variable, how do you assign the value?
- The value of a variable is assumed to be a string:
tmp=1234.5 # assigns the string "1234.5" as the value of tmp

Variables (continued):

- Strings can be concatenated in scripts:

```
$ cat countBoth3
#!/bin/ksh
tmp=.countboth.tmp
cat > $tmp
echo -n "wc says:"
wc -l < $tmp | tr -s " "
echo -n "lc says: "
java lc < $tmp
echo "countBoth3 is using $tmp for the filename"
rm $tmp
$ cat countBoth3 | countBoth3
wc says: 9
lc says: 9
countBoth3 uses .countboth.tmp for the filename
```

- Suppose the script was typed wrong, and the line entered in the script was

```
$ cat countBoth4
tmp=.countboth.tmp
...
echo "countBoth4 uses ($tmp)for the filename"
rm $tmp
$ who | countBoth4
wc says: 62
lc says: 62
countBoth4 uses the filename
```

Can fix this by using `{ }`'s:

```
$ cat countBoth5
tmp=.countboth.tmp
...
echo "countBoth5 uses ${tmp}for the filename"
rm $tmp
$ who | countBoth5
wc says: 61
lc says: 61
countBoth5 uses .countboth.tmpfor the filename
```

Variables (continued):

- Variable interpolation is done inside double quotes:
- Variable interpolation is not done inside single quotes:

```
$ cat countBoth6
#!/bin/ksh
tmp=.countboth.tmp
cat > $tmp
echo -n "wc says:"
wc -l < $tmp | tr -s " "
echo -n "lc says: "
java lc < $tmp
echo "countBoth6 uses $tmp for the filename"
echo 'countBoth6 uses $tmp for the filename'
rm $tmp
$ who | countBoth6
wc says: 68
lc says: 68
countBoth6 uses .countboth.tmp for the filename
countBoth6 uses $tmp for the filename
```

Command-line arguments for scripts:

- **\$N**, where **N** is a number, refers to the command-line argument in that position:

```
$ cat printargs1
#!/bin/ksh
echo 1st argument is $1
echo 2nd argument is \"$2\"
echo 3rd argument is \"'$3'\"
$ printargs1 one two three
1st argument is one
2nd argument is 'two'
3rd argument is "three"
```

- If there is no argument at a specified position, the variable simply has no value:

```
$ printargs1 one two
1st argument is one
2nd argument is 'two'
3rd argument is ""
```

- Is **\$1** really the first argument on the command-line?

```
$ cat printargs2
#!/bin/ksh
echo 0th argument is $0
echo 1st argument is $1
echo 2nd argument is \"$2\"
echo 3rd argument is \"'$3'\"
$ printargs2 one two three
0th argument is printargs2
1st argument is one
2nd argument is 'two'
3rd argument is "three"
```

Command-line arguments for scripts (continued):

- Can use the special variable `$*` to indicate we want all the arguments:

```
$ cat printargs3
#!/bin/ksh
echo The arguments are $*
echo 2nd argument is \"$2\"
echo 3rd argument is \"$3\"
$ printargs3 one two three four five six seven eight nine ten eleven
The arguments are one two three four five six seven eight nine ten eleven
2nd argument is 'two'
3rd argument is 'three'
```

- The special variable `$#` to indicate we want a count of the arguments:

```
$ cat printargs4
#!/bin/ksh
echo There are $# arguments.
echo The arguments are $*
echo 2nd argument is \"$2\"
echo 3rd argument is \"$3\"
$ printargs4 one two "three four five" six seven eight nine ten eleven
There are 9 arguments.
The arguments are one two three four five six seven eight nine ten eleven
2nd argument is 'two'
3rd argument is "three four five"
```

Control Structures:

- There are flow control structures available in ksh scripts.
 - `for`, `while`, `if`, etc.
 - They also exist in other scripting languages as well; though the details will vary!

The for statement:

Reading: *Learning the Korn Shell*, Section 5.1.

- A common use of `for` is to iterate over the command-line arguments:

```
$ cat forargs
#!/bin/ksh
echo There are $# arguments
echo They are $*
for i in $*
do
    echo \"$i\"
done
$ forargs one two three
There are 3 arguments
They are one two three
'one'
'two'
'three'
```

The for statement (continued):

- The general form of the for statement:

```
for i in words
do
    cmd1
    ...
    cmdN
done
```

- The shell is very line oriented.

- What happened here?

```
$ cat forargs2
#!/bin/ksh
echo There are $# arguments:
echo They are $*
for i in $* do echo $i done
$ forargs2 one two three four
There are 4 arguments:
They are one two three four
forargs2[3]: syntax error at line 3 : `for' unmatched
$
```

- Can be fixed with semi-colons:

```
for i in $*; do echo $i; done
```

The for statement (continued):

- “On the fly”: for loop at the shell prompt:

```
$ for i in one two three four five six
> do
> echo \"$i\"
> done
'one'
'two'
'three'
'four'
'five'
'six'
$
```

- The `>` at the beginning of the lines is printed by the shell.

- The loop is executed as soon as `done` is typed.

- The line can be recalled (use the up-arrow):

```
$ for i in one two three four five six^Jdo^Jecho \"$i\"^Jdone
```

- The `^J`'s occur at the line breaks.

- Problem: Write a script that echo's its command-line arguments in reverse order:

```
$ revargs one two three four
four three two one
```

The if/else statement:

- Unix commands, including shell scripts and compiled programs, return an exit status.
 - By convention, the exit status is 0 if the program/command terminated “OK”. The status is positive, in the range 1-255, if the command did not terminate “OK”.
 - Each command/program may have its own notion of what is “OK” and what is not.
- The general form of the if/else statement:

```
if command ran successfully
then
    command(s) to execute on successful result
else
    command(s) to execute on error
fi
```
- Example, we had looked at a script named `lc` before that would execute `java lc` for us. But, what if `lc.class` is not present in the directory? Write a script that will check for `lc.class` first and will compile it if it is missing:

```
$ cat lineCount1
#!/bin/ksh
if ls lc.class          $ rm lc.class
                        rm: remove lc.class (yes/no)? y
then
                        $ lineCount1 < forargs
                        lc.class: No such file or directory
:
else
                        7
    javac lc.java
fi
java lc
```

The if/else statement (continued):

- But, the `lc` script prints an error message when the `lc.class` file is not present. This would be confusing to a user.
 - Re-direct standard error to `/dev/null`. This is a device that goes no-where. Also known as the “bit bucket”.
- The special variable `$?` holds the exit status of the last command executed. This can be used from the command-line and/or within scripts:

```
$ ls lc.class
lc.class
$ echo $?
0
$ rm lc.class
rm: remove lc.class (yes/no)? y
$ ls lc.class
lc.class: No such file or directory
$ echo $?
2
```

The if/else statement (continued):

- The `test(1)` command can be used to check for a variety of conditions such as whether a relation, like equality, holds between two values. It can also be used to test for file attributes such as existence, executability, etc.
 - `test` indicates success or failure by setting the exit status.
- Example: `test -f file` tests whether `file` exists and is a “regular file”:

```
$ test -f args.java
$ echo $?
0
$ test -f zap.java
$ echo $?
1
```

Note: `test` does not print anything to standard output or standard error.

The if/else statement (continued):

- Here is a more general compile-and-run script for use with Java:

```
$ cat runJ1
#!/bin/ksh
if test $# -eq 0
then
    echo "Usage: $0 file (w/o .java!)"
    exit 1
fi
if test -f $1.java
then
    if javac $1.java
    then
        java $*
    fi
else
    echo $1.java: Not found
fi
$ runJ1 args one two three
'one'
'two'
'three'
```

The if/else statement (continued):

- Many shells (including the Korn shell) consider `[` (the left square bracket) to be an alias for the test command.
 - It is common to use `[`, not `test`, in scripts.

- Example:

```
$ cat runJ2
#!/bin/ksh
if [ $# -eq 0 ]
then
    echo "Usage: $0 file (w/o .java!)"
    exit 1
fi
if [ -f $1.java ]
then
    if javac $1.java
    then
        java $*
    fi
else
    echo $1.java: Not found
fi
$ runJ2 lc < args.java
0
```

The while statement:

- General form:

```
while command-line
do
    cmd1
    ...
    cmdN
done
```

- Example: clock-ticks, using an “on-the-fly” while loop.

```
$ while true
> do
>   date
>   sleep 1
> done
Mon Aug 21 09:35:44 MST 2006
Mon Aug 21 09:35:45 MST 2006
Mon Aug 21 09:35:46 MST 2006
Mon Aug 21 09:35:47 MST 2006
Mon Aug 21 09:35:48 MST 2006
^C $
```

The while statement (continued):

- Can use a pipe since the `while` statement wants a command-line. For example, print a message when a specific user logs off:

```
$ cat waitfor
#!/bin/ksh
while who | fgrep -s $1
do
    sleep 10
done
echo $1 is off!
$ waitfor patrick
```

- The `-s` option for `fgrep` is for silent operation; that is, `fgrep` will only provide an exit status indicating success or failure.
- Note: A pipe provides the exit status of the last command of the pipe.
 - The assumption is that failure of an earlier command in the pipe will result in the last command also failing.
 - Thus, the `while` sees only the exit status of `fgrep`, not the exit status of `who`.
 - This behavior is the same for `if` statements as well.

Debugging shell scripts:

- Using `echo` is the most common way.
- The `-v` and/or `-x` options to `ksh` can provide some help.
- `-v` echo's each command before it is executed:
- `-x` causes each command to echo after expansion:

```
$ cat printargs4
#!/bin/ksh
echo There are $# arguments.
echo The arguments are $*
echo 2nd argument is \"$2\"
echo 3rd argument is \"$3\"
$ ksh -v printargs4 one two three
echo There are $# arguments.
There are 3 arguments.
echo The arguments are $*
The arguments are one two three
echo 2nd argument is \"$2\"
2nd argument is 'two'
echo 3rd argument is \"$3\"
3rd argument is 'three'
```

```
$ ksh -v -x runJ1 args one two
if test $# -eq 0
then
    echo "Usage: $0 file (w/o .java!)"
    exit 1
fi
+ test 3 -eq 0
if test -f $1.java
then
    if javac $1.java
    then
        java $*
    fi
else
    echo $1.java: Not found
fi
+ test -f args.java
+ javac args.java
+ java args one two
'one'
'two'
```

Shell Variables:

- Can be created and used interactively:

```
$ pwd
/home/patrick
$ ux=-cs352/fall106/UnixExamples
$ echo $ux
/home/cs352/fall106/UnixExamples
$ cd $ux
$ pwd
/home/cs352/fall106/UnixExamples
```
- The command **set** can be used to see all the current variables. Here is an (edited for length) example:

```
$ set
EDITOR=vi
HOME=/home/patrick
LOGNAME=patrick
OLDPWD=/home/patrick/352/examples
PATH=/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/ccs/bin:/usr/bin/X11/..
PPID=16026
PS1='lectura->'
PS2='>'
PWD=/home/patrick
SECONDS=86535
SHELL=/bin/ksh
TERM=xterm
TZ=US/Arizona
USER=patrick
```

- How can I copy a file from the directory I was previously in to the directory I am currently in?

Shell Variables (continued):

- By default, variables are not transmitted to an executable:

```
$ ux=-cs352/fall106/UnixExamples
$ echo $ux
/home/cs352/fall105/assign2
$ cat variable1
echo The value of ux is $ux
$ variable1
The value of ux is
```
- A variable can be seen by an executable if it is put on the *export list*:

```
$ export ux
$ cat variable1
echo The value of ux is $ux
$ variable1
The value of ux is /home/cs352/fall106/UnixExamples
$ ux="has been changed"
$ variable1
The value of ux is has been changed
```
- With no arguments, **export** will show all variables currently in the export list.

Shell Variables (continued):

- “The environment” is the set of variables and their values that are exported.
- The **env** command will show the current contents of this list.
- “Environment variable” is a common way of referring to a variable that is in the set.
- Environment variables can (and often do) affect the execution of programs. Example:

```
$ date
Mon Aug 21 09:54:21 MST 2006
$ TZ=US/Eastern
$ date
Mon Aug 21 09:54:27 MST 2006
$ export TZ
$ date
Mon Aug 21 12:54:41 EDT 2006
$ export TZ=US/Arizona
$ date
Mon Aug 21 09:55:07 MST 2006
$ TZ=Japan date
Tue Aug 22 01:55:28 JST 2006
$ date
Mon Aug 21 09:55:30 MST 2006
```

- Note: Can both **set** and **export** a value for a variable with the **export** command.

- Changing a variable on the same line with executing a command affects the command

- It does **not** change the variable in the environment.

- Transmission of values is one-way: into the script only.
- Changing a variable’s value in a script affects only that script — it has no effect on the environment.

Shell Variables (continued):

- There are four variables that control the different prompts for the shell: **PS1**, **PS2**, **PS3**, and **PS4**.
- **PS1** is the prompt that is normally displayed.
- You can change the value of **PS1**. Some examples:

```
$ echo $PS1
$
$ PS1="! " ← puts the current command count in the prompt.
538 PS1="! lectura->"
539 lectura->PS1="! !;->" ← Use !! to put an exclamation mark in the prompt.
540 !->
```
- To get the current directory pathname into the prompt:
 - Want **pwd** in the prompt. Can try:

```
$ PS1="$PWD->"
/home/patrick->cd 352/examples
/home/patrick->pwd
/home/patrick/352/examples
OOPS!
```
 - Use single quote marks instead. This puts the **pwd** shell variable in the prompt and evaluates it every time the prompt is printed:

```
$ PS1='$PWD->'
/home/patrick/352/examples->cd ..
/home/patrick/352->cd ..
/home/patrick->
```

Shell Variables (continued):

- PS2 is the prompt used when entering interactive shell commands that need more input. Example:

```
$ echo $PS2
>
$ for i in one two
> do
> echo $i
> done
one
two
$ PS2="more stuff needed ->"
$ for i in one two
more stuff needed ->do
more stuff needed ->echo $i
more stuff needed ->done
one
two
```

Shell Variables (continued):

- PS4 is used during debug sessions with shell scripts:

```
$ echo $PS4
+
$ ksh -x lineCount2 < args.java
+ ls lc.class
+ 2> /dev/null
lc.class
+ :
+ java lc
8
$ PS4="debug output here-> "
$ ksh -x lineCount2 <args.java
+ ls lc.class
+ 2> /dev/null
lc.class
+ :
+ java lc
8
```

- What went wrong? Why did the revised PS4 not show up during the second debug run?

\$PATH:

- How the shell knows which command you really wanted!
- \$PATH lists a series of directories that the shell will search to find the command you specified. By default, on lectura:

```
$ echo $PATH
/usr/bin:/bin:/usr/sbin:/sbin
```

 - This can be expanded to look for executables in additional directories.
 - This line is in my *.profile*:

```
PATH=/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/ccs/bin:.
```
 - There are can be multiple versions of some commands. For example, there is a command called `test` that is located in `/usr/bin/test`. If I create an executable script (or program) named `test` in a local directory, there will be two versions of `test`.
 - In this case, what does the command `test` actually execute?

\$PATH (continued):

- \$PATH determines which `ps` is executed when I do not specify the full path.
 - In my case, `/bin` is in my \$PATH. Thus, `ps` will be `/bin/ps`.
- The order of directories listed in \$PATH is important!
 - When a command is found in more than one of the \$PATH directories, it is the first one found that is executed.
- Quick list of what is in common UNIX directories:
 - `/usr/bin` are the executables supplied by the vendor.
 - `/usr/local/bin` is an example of a directory with additional or optional executables.
 - `.` is the current directory.
 - If present (it does not have to be), `.` should always be the last item in \$PATH.
 - There are both practical and security reasons for this position!
 - See the discussion on page 83 (Section 3.4.2.7) "PATH security considerations".
 - Unlike the recommendation there, I believe putting `.` at the end of \$PATH to be a reasonable compromise. The implicit assumption here is that you know what you are doing when you make this choice. If you are not sure, leave `.` out of your \$PATH.

\$PATH (continued):

- It is common for users to create their own “bin” directory:
 - Put executables here that you have written and wish to use.
 - Put executables here that you gather from elsewhere and which are not installed in any of the “standard” locations.
 - Add this to your \$PATH. For example, I might put the following in my *.profile*:

```
PATH=/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:~/bin:
```
- If you want to know which version of a command will be executed, here are two ways: **which** and **type**.

```
$ which ps
/bin/ps
$ type ps
ps is a tracked alias for /bin/ps
```

\$PATH (continued):

- A final \$PATH note: it is not an error to have a non-existent directory in your \$PATH:

```
$ echo $PATH
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:.
$ ls -l /usr/zap
/usr/zap: No such file or directory
$ PATH="/usr/zap:$PATH"
$ echo $PATH
/usr/zap:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:.
$ ls -l args.java
-rw-r--r--  1 patrick  dept      193 Aug 25 16:15 args.java
```

- Note: I used \$PATH above in re-defining \$PATH.
- Question: Why would I want to put a non-existent directory in my \$PATH?

ksh .profile startup file:

- You can define variables (and aliases) in a *.profile* file located in your home directory.
- This file will be read each time you invoke the Korn shell:
 - This will happen automatically when you login, if you have ksh as your default shell.
 - This will also happen each time you invoke ksh from the command line.
- You can define variables in *.profile* exactly as we have discussed here.
- You can export these variables to make them visible within ksh scripts (or choose not to export them).
- As an example, here is my *.profile*:

```
alias ls="ls -F"
alias rm="rm -i"
alias aux="ps -efa | egrep patrick"
alias lec="ssh -v patrick@lectura.cs.arizona.edu"
export VISUAL=$(whence vi)
PATH=/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:.
export EDITOR=vi
host=$(hostname | cut -f 1 -d .)
export PS1="${host}-> "
```

- You can “test” your *.profile*:

```
. .profile
```

Aliases:

- You can create an alias for commands. Usually done to shorten what you would normally have to type:

```
$ alias jc="javac"
$ jc lc.java
$ alias j="java"
$ j lc < args.java
8
```

- When you type the command **alias** by itself, you will get a list of the current aliases:

```
$ alias
aux='ps -efa | egrep patrick'
j=java
jc=javac
ls='ls -F'
rm='rm -i'
```

- The **type** command will report aliases:

```
$ type ls
ls is an alias for ls -F
$ type rm
rm is an alias for rm -i
```

Command substitution:

- Provides a way to turn the output of a command into command-line arguments:

```
$ cat javafiles
args.java
bad.java
Hello.java
lc.java
tilde.java
$ echo $(cat javafiles)
args.java bad.java Hello.java lc.java tilde.java
```

- `$(command-line)` will run the enclosed *command-line*, and substitute the whitespace-separated words that result for the `$(...)` construct.
- Any number of command substitutions may appear on a command line.
- The enclosed commands may be arbitrarily complex.

```
$ wc $(cat javafiles classfiles)
$ ls -l $(cat javafiles)
-rw-rw-r-- 1 patrick cs352f06 193 Aug 18 13:04 args.java
-rw-rw-r-- 1 patrick cs352f06 191 Aug 18 13:04 bad.java
-rw-rw-r-- 1 patrick cs352f06 144 Aug 18 13:04 Hello.java
-rw-rw-r-- 1 patrick cs352f06 404 Aug 21 10:15 lc.java
-rw-rw-r-- 1 patrick cs352f06 303 Aug 18 13:04 tilde.java
$ wc $(cat javafiles classfiles)
 8   31   193 args.java
 8   31   191 bad.java
 7   19   144 Hello.java
18   47   404 lc.java
 8   27   303 tilde.java
 8   23   594 args.class
 8   22   592 bad.class
 7   20   417 Hello.class
10   32   701 lc.class
 8   22   583 tilde.class
 90  274  4122 total
```

Command-substitution (continued):

- Example:

```
$ wc $(cat javafiles classfiles | sort -k 2 -t.)
 8   23   594 args.class
 8   22   592 bad.class
 7   20   417 Hello.class
10   32   701 lc.class
 8   22   583 tilde.class
 8   31   193 args.java
 8   31   191 bad.java
 7   19   144 Hello.java
18   47   404 lc.java
 8   27   303 tilde.java
 90  274  4122 total
```

- Note: `echo` and command-substitution are inverses:

- `echo` turns arguments into output

- command-substitution turns output into arguments

```
$ echo a b c
a b c
$ echo $(echo a b c)
a b c
```

- What does the following script do?

```
for i in $*
do
mv -i $i $(echo $i | tr A-Z a-z)
done
```

Command-substitution (continued):

- What does the following do?

```
$ yes "x" | head -n $(cat lc.java | wc -l) > lc.x
```

- Problem: Run `more` on the files in the current directory that contain the word "if". (Hint: `fgrep -l` prints only the names of files that contain a match.)

- Problem: Copy files in the current directory that contain the words "if" and "while" to a subdirectory named *both*.

- Problem: Write a script `mkprefix pfx N` with this behavior:

```
$ ls
mkprefix
$ mkprefix x 5
$ ls
mkprefix x.1 x.2 x.3 x.4 x.5
```

- Hint: `seq N` prints integers from 1 to N, one per line.

Command-substitution (continued):

- Can use command substitution to assign a value to a variable:

```
$ f=$(fgrep -l Reader *.java)
$ echo $f
lc.java
$ wc $f
 18   47   404 lc.java
$ ls -l $f
-rw-rw-r-- 1 patrick cs352f06 404 Aug 21 10:15 lc.java
```

- Here is a script that produces the sum of its arguments:

- `eval.java` takes three arguments: `operand opcode operand` and returns the result of the operation.

```
$ cat sum
sum=0
for i in $*
do
sum=$(java eval $sum + $i)
done
echo $sum
```

- Problem: Use `sum` to compute the sum of the integers from 1 through 100.

Command-substitution (continued):

- Command substitution is performed inside double-quoted literals.
 - Two examples that might be used inside a script:

```
echo "Hostname: $(hostname | cut -f1 -d.)"
echo "Users:    $(echo $(who | wc -l))"
```
 - What is the purpose of the second `echo` in the second line?
- An older, but still quite common, form of command substitution is back quotes:

```
wc `cat javafiles classfiles | sort +1 -t.`
```

 - The back-quote form may be easier to type. May also be harder to read.
 - The back-quote form does not nest:

```
$ echo $(echo $(echo x))
x
$ echo `echo `echo x``
echo x
```

Assorted (useful) Utilities — diff:

Reading: The man page for `diff`; *Unix Power Tools*, Section 11.1.

- Compares two files. The default output can be cryptic:

```
$ diff args.java args1.java
4c4
<   for (int i = 0; i < args.length; i++)
---
>   for (int i = 0; i<args.length; i++)
6d5
<   } // main
7a7,8
>   } // main method
>
```

```
$ cat args.java
public class args {
    public static void main(String args[] ) {
        for (int i = 0; i < args.length; i++)
            System.out.println("" + args[i] + "");
    } // main
} // class args

$ cat args1.java
public class args {
    public static void main(String args[] ) {
        for (int i = 0; i<args.length; i++)
            System.out.println("" + args[i] + "");
    } // main method
} // class args
```

Assorted (useful) Utilities — diff (continued):

- The `-c` option (context) is generally more readable:

```
$ diff -c args.java args1.java
*** args.java Thu Aug 25 16:15:13 2005
--- args1.java Thu Sep 22 09:48:19 2005
*****
*** 1,8 ****
public class args {
    public static void main(String args[] ) {
1         for (int i = 0; i < args.length; i++)
            System.out.println("" + args[i] + "");
-     } // main
    } // class args
--- 1,9 ----
public class args {
    public static void main(String args[] ) {
1         for (int i = 0; i<args.length; i++)
            System.out.println("" + args[i] + "");
+     } // main method
+     } // class args
```

 - First two lines show the info about the two files.
 - A row of asterisks separates each section of differences.
 - `*** 1,8 ****` indicates that lines 1-8 of the first file immediately follow.
 - The `***` correspond to the `*`'s used in the file names.
 - `--- 1,9 ----` indicates that lines 1-9 of the second file come next.
 - The `---` correspond to the `-`'s used in the file names.

Assorted (useful) Utilities — diff (continued):

- ```
$ diff -c args.java args1.java
*** args.java Thu Aug 25 16:15:13 2005
--- args1.java Thu Sep 22 09:48:19 2005

*** 1,8 ****
public class args {
 public static void main(String args[]) {
1 for (int i = 0; i < args.length; i++)
 System.out.println("" + args[i] + "");
- } // main
 } // class args
--- 1,9 ----
public class args {
 public static void main(String args[]) {
1 for (int i = 0; i<args.length; i++)
 System.out.println("" + args[i] + "");
+ } // main method
+ } // class args
```
- `!` indicates a line that differs between the two files.
  - `-` indicates a line present in the first file but not in the second.
  - `+` indicates a line present in the second file but not in the first.
  - Another way to think about the report is in terms of modifications to the first file that would produce the second file:
    - `!` change this line to match the line in the second file.
    - `-` delete this line.
    - `+` add this line.

Assorted (useful) Utilities — diff (continued):

- Here is a **diff** example with several differences. The **-c1** option limits the context to one line (the default with **-c** is 3 lines). Note that a change that is only an insertion or deletion does not show the section from the other file.

```
$ diff -c1 100.txt 100mod.txt
*** 100.txt Thu Sep 22 09:57:53 2005
--- 100mod.txt Thu Sep 22 09:58:17 2005

*** 9,10 ****
--- 9,11 ----
9
+ a
10

*** 30,32 ****
30
- 31
32
--- 31,32 ----

*** 55,59 ****
55
- 56
- 57
- 58
59
```

Assorted (useful) Utilities — diff (continued):

- By default, **diff** is very strict — two files are identical only if character-by-character their contents are the same.
  - diff** produces no output if the files are identical:  
\$ diff args.java args.java  
\$

- diff** has many options to relax this strict requirement. **-b** ignores differences in runs of whitespace. **-w** ignores differences in any amount of whitespace:

```
$ diff -w -c args.java args1.java
*** args.java Thu Aug 25 16:15:13 2005
--- args1.java Thu Sep 22 09:48:19 2005

*** 3,8 ****
public static void main(String args[]) {
 for (int i = 0; i < args.length; i++)
 System.out.println("" + args[i] + "");
- } // main
} // class args
--- 3,9 ----
public static void main(String args[]) {
 for (int i = 0; i < args.length; i++)
 System.out.println("" + args[i] + "");

+ } // main method
+ } // class args
```

These two lines are not reported as being different.

Assorted (useful) Utilities — diff (continued):

- If one **diff** operand specifies a file and the other specifies a directory, **diff** looks in the directory for a file of the same name.
  - diff args.java v1**, where **v1** is a directory, is equivalent to **diff args.java v1/args.java**
- The **-r** option to **diff** will recursively compare all the files in two directories.
  - Very handy in determining differences among two versions of a project!
- The **-i** option will ignore case differences.
- The **-e** option will produce a set of change commands for the **ed** editor that will change the first file into the second.

Assorted (useful) Utilities — patch:

- Makes changes to a file based on a “diff” — the output of a **diff** run.
- Can be used to provide a short list of updates to a file, instead of supplying the whole new file.
- Can be applied recursively to a directory:
  - Use **-r** with **diff** to compare the two directories.

```
$ diff -C 1 args1.java args.java
*** args1.java Tue Sep 27 07:18:15 2005
--- args.java Tue Sep 27 07:16:31 2005

*** 3,9 ****
public static void main(String args[]) {
! for (int i = 0; i < args.length; i++)
 System.out.println("" + args[i] + "");
- } // main method
- } // class args
--- 3,8 ----
public static void main(String args[]) {
! for (int i = 0; i < args.length; i++)
 System.out.println("" + args[i] + "");
+ } // main
} // class args
$ diff -C 1 args1.java args.java > args1-args.diff
$ patch < args1-args.diff
Looks like a new-style context diff.
File to patch: args1.java
done
$ diff -C 1 args1.java args.java
No differences encountered
```

### Assorted (useful) Utilities — **find**:

Reading: *Unix Power Tools*, Chapter 9.

- Recursively searches a directory tree (or trees) for files matching specified criteria.
- Simplest use: specify a directory name.
  - **find** prints the name of every directory entry in the tree.

```
$ find /cs/www/classes/cs352/fall106
/cs/www/classes/cs352/fall106
/cs/www/classes/cs352/fall106/notes
/cs/www/classes/cs352/fall106/notes/1-Unix-large.pdf
/cs/www/classes/cs352/fall106/notes/1-Unix.pdf
/cs/www/classes/cs352/fall106/index.html
/cs/www/classes/cs352/fall106/programs
/cs/www/classes/cs352/fall106/programs/Assignment4v1.1.html
/cs/www/classes/cs352/fall106/programs/Assignment3v1.1.html
/cs/www/classes/cs352/fall106/programs/Assignment1v1.0.html
/cs/www/classes/cs352/fall106/programs/Assignment4v1.0.css
/cs/www/classes/cs352/fall106/programs/Assignment3v1.0.css
/cs/www/classes/cs352/fall106/programs/Assignment2v1.1.html
/cs/www/classes/cs352/fall106/programs/Assignment3v1.1.pdf
/cs/www/classes/cs352/fall106/programs/Assignment2v1.0.html
/cs/www/classes/cs352/fall106/programs/Assignment2v1.0.css
... lots more
```
  - Hidden files are shown (there were none in the above directory), but `.` and `..` are not shown (special cases).
  - If no directory is specified, the default is the current directory.

### Assorted (useful) Utilities — **find** (continued):

- A number of “tests” are available to constrain the results.
- **-name** requires that files match a wildcard specification:

```
$ pwd
/home/patrick/352/examples
$ find . -name "*.java"
./Hello.java
./args.java
./lc.java
./today/Hello.java
./today/args.java
./today/lc.java
./tilde.java
./eval.java
./args1.java
./args2.java
./both/lc.java
```

- The following (often, but not always) fails to work, why?

```
$ pwd
/home/patrick/352/examples
$ find . -name *.java
find: paths must precede expression
Usage: find [-H] [-L] [-P] [path...] [expression]
$
```

- Can be useful as a process substitution:

```
$ more $(find . -name "*.java")
```

- **find** can take a long time:

```
$ pwd
/home/patrick
$ time find . | wc -l
9161

real 0m15.14s
user 0m0.02s
sys 0m0.17s
```

### Assorted (useful) Utilities — **find** (continued):

Access and modification times:

- Search for files that were modified in the last two “days”:

```
$ pwd
/home/patrick/352/examples
$ find . -mtime -2
.
./args.java
./args
./args2-args.diff
./args1.java
./args2.java
./zapscrip
./args1-args.diff
./args.java.rej
./args.java.orig
```
- Note that both ordinary files and directories (i.e., `.` in the output above) are produced.
- **-mtime -2** means “modified in the last 48 hours”
- **-mtime +3** means modified more than 72 hours ago;
- **-mtime 2** means last modified 48 to 72 hours ago.

### Assorted (useful) Utilities — **find** (continued):

- Search for files above a certain size:

```
$ ls -l *.java */*.java
-rw----- 1 patrick dept 144 Aug 22 17:24 Hello.java
-rw-r--r-- 1 patrick dept 193 Sep 27 07:22 args.java
-rw-r--r-- 1 patrick dept 199 Sep 27 07:22 args1.java
-rw-r--r-- 1 patrick dept 199 Sep 27 07:04 args2.java
-rw-r--r-- 1 patrick dept 357 Sep 22 16:13 both/lc.java
-rw-r--r-- 1 patrick dept 831 Sep 22 09:10 eval.java
-rw-r--r-- 1 patrick dept 357 Sep 5 14:59 lc.java
-rw-r--r-- 1 patrick dept 291 Sep 5 15:44 tilde.java
-rw----- 1 patrick dept 144 Sep 1 16:44 today/Hello.java
-rw-r--r-- 1 patrick dept 193 Sep 1 16:44 today/args.java
-rw-r--r-- 1 patrick dept 357 Sep 1 16:44 today/lc.java
```

```
$ find . -size +300c -name "*.java"
./lc.java
./today/lc.java
./eval.java
./both/lc.java
$ find . -size 300c -name "*.java"
$
```

- **+300** uses blocks (of 512 bytes each) for the size.
- **+300c** uses bytes for the size.
- **+300k** uses bytes \* 1024 for the size (300k = 300 kilobytes).

- Why did the last **find** command come up empty?
- What would be found by the following command?

```
$ find . -size -300c -name "*.java"
```

Assorted (useful) Utilities — find (continued):

- Find directories:

```
$ pwd
/cs/www/classes/cs352/fall06
$ find . -type d
.
./notes
./programs
```
- Negate a condition:

```
$ pwd
/cs/www/classes/cs352/fall05
$ find . ! -type f
.
./notes
./programs
./gdbdocs
./Syllabus_files
```

  - If more than one test is specified, **find** does an AND of the test results:

```
$ pwd
/home/patrick/352/examples
$ find . -name "*.java" -mtime -2
./args.java
./args1.java
./args2.java
```
- Use **-o** for OR:

```
$ find . -size +100k -o \(-name *.java -mtime -1 \)
./lgDictionary.txt
./bad.java
./text-files/sawyer.txt
./tilde.java
```

  - Why are the **\**'s needed?

Assorted (useful) Utilities — find (continued):

- Find files newer than a specified file:

```
$ pwd
/home/cs352/fall06/UnixExamples
$ ls -l *java
-rw-rw-r-- 1 patrick cs352f06 199 Aug 18 13:04 args1.java
-rw-rw-r-- 1 patrick cs352f06 193 Aug 18 13:04 args.java
-rw-rw-r-- 1 patrick cs352f06 191 Sep 13 22:16 bad.java
-rw-rw-r-- 1 patrick cs352f06 948 Aug 18 13:04 eval.java
-rw-rw-r-- 1 patrick cs352f06 144 Aug 18 13:04 Hello.java
-rw-rw-r-- 1 patrick cs352f06 404 Aug 21 10:15 lc.java
-rw-rw-r-- 1 patrick cs352f06 303 Sep 13 22:16 tilde.java
$ find . -newer lc.java -name *.java
./bad.java
./tilde.java
```

Assorted (useful) Utilities — find (continued):

- Using **-ls** causes **ls -lids** to be performed on each file found:

```
$ pwd
/home/cs352/fall06/UnixExamples
$ find . -name *.java -ls
16793891 4 -rw-rw-r-- 1 patrick cs352f06 199 Aug 18 13:04 ./args1.java
16793893 4 -rw-rw-r-- 1 patrick cs352f06 193 Aug 18 13:04 ./args.java
16793895 4 -rw-rw-r-- 1 patrick cs352f06 191 Sep 13 22:16 ./bad.java
16793905 4 -rw-rw-r-- 1 patrick cs352f06 948 Aug 18 13:04 ./eval.java
16793909 4 -rw-rw-r-- 1 patrick cs352f06 144 Aug 18 13:04 ./Hello.java
20953474 4 -rw-rw-r-- 1 patrick cs352f06 404 Aug 21 10:15 ./lc.java
9166717 4 -rw-rw-r-- 1 patrick cs352f06 303 Sep 13 22:16 ./tilde.java
```

Assorted (useful) Utilities — find (continued):

- Using **-exec** runs a specified command.
  - The command starts with **-exec** and continues to the next semicolon.
  - The semi-colon has to be escaped so the shell will not intercept it.
  - For each match that **find** locates, the path to that match is substituted for **{}** in the command.

```
$ pwd
/home/patrick/352/examples
$ find . -name "*.java" -exec wc -l {} \;
9 ./args1.java
8 ./args.java
8 ./bad.java
33 ./eval.java
7 ./Hello.java
18 ./lc.java
8 ./tilde.java
```

    - How many times was the **wc** command executed above? How can you tell?
  - Shell aliases do not get substituted when using **-exec** with **find**. The following will not use my **rm -i** alias:

```
$ find . -name "print*" -exec rm {} \;
```

### Assorted (useful) Utilities — tar:

- Used to create an “archive” that contains all files in a tree, or create a tree from a tar archive.
  - Takes its name from “tape archiver”.

- A small directory tree to use in these examples:
- To create an archive (“tar file”) of arExample:

```
$ ls -R arExample
arExample:
count/ java-src/ zapscript*

arExample/count:
countBoth* countBoth2* countBoth3* countBoth4*

arExample/java-src:
args.java args1.java args2.java

c = create the archive.
v = verbose output.
f filename = name of the archive that will be created.

$ tar cvf arExample.tar arExample
a arExample/ OK
a arExample/zapscript 1K
a arExample/count OK
a arExample/count/countBoth 1K
a arExample/count/countBoth2 1K
a arExample/count/countBoth3 1K
a arExample/count/countBoth4 1K
a arExample/java-src/ OK
a arExample/java-src/args.java 1K
a arExample/java-src/args1.java 1K
a arExample/java-src/args2.java 1K
$ ls -l arExample.tar
-rw-r--r-- 1 patrick dept 10752 Sep 27 09:29 arExample.tar
```

### Assorted (useful) Utilities — tar (continued):

- Can view the “table of contents” of an archive using the t option:

```
$ tar tf arExample.tar
arExample/
arExample/zapscript
arExample/count/
arExample/count/countBoth
arExample/count/countBoth2
arExample/count/countBoth3
arExample/count/countBoth4
arExample/java-src/
arExample/java-src/args.java
arExample/java-src/args1.java
arExample/java-src/args2.java
$ tar tvf arExample.tar
drwxr-xr-x 122/46 0 Sep 27 09:16 2005 arExample/
-rwxr--r-- 122/46 105 Sep 27 09:16 2005 arExample/zapscript
drwxr-xr-x 122/46 0 Sep 27 09:28 2005 arExample/count/
-rwxr--r-- 122/46 156 Sep 27 09:16 2005 arExample/count/countBoth
-rwxr--r-- 122/46 120 Sep 27 09:16 2005 arExample/count/countBoth2
-rwxr--r-- 122/46 160 Sep 27 09:16 2005 arExample/count/countBoth3
-rwxr--r-- 122/46 159 Sep 27 09:16 2005 arExample/count/countBoth4
drwxr-xr-x 122/46 0 Sep 27 09:15 2005 arExample/java-src/
-rw-r--r-- 122/46 193 Sep 27 09:15 2005 arExample/java-src/args.java
-rw-r--r-- 122/46 199 Sep 27 09:15 2005 arExample/java-src/args1.java
-rw-r--r-- 122/46 199 Sep 27 09:15 2005 arExample/java-src/args2.java
```

### Assorted (useful) Utilities — tar (continued):

- The x option is used to extract the contents of an archive, reproducing the original tree:

```
$ mkdir v2
$ cd v2
$ tar xvf ../arExample.tar
x arExample, 0 bytes, 0 tape blocks
x arExample/zapscript, 105 bytes, 1 tape blocks
x arExample/count, 0 bytes, 0 tape blocks
x arExample/count/countBoth, 156 bytes, 1 tape blocks
x arExample/count/countBoth2, 120 bytes, 1 tape blocks
x arExample/count/countBoth3, 160 bytes, 1 tape blocks
x arExample/count/countBoth4, 159 bytes, 1 tape blocks
x arExample/java-src, 0 bytes, 0 tape blocks
x arExample/java-src/args.java, 193 bytes, 1 tape blocks
x arExample/java-src/args1.java, 199 bytes, 1 tape blocks
x arExample/java-src/args2.java, 199 bytes, 1 tape blocks
$ ls -R
.:
arExample/
./arExample:
count/ java-src/ zapscript*
./arExample/count:
countBoth* countBoth2* countBoth3* countBoth4*
./arExample/java-src:
args.java args1.java args2.java
```

#### Caution:

If the directory is not empty:

- tar will add to the contents of the directory.
- tar will overwrite any files/directories of the same name!

### Assorted (useful) Utilities — tar (continued):

- Individual files and/or directories can be extracted:

```
$ pwd
/home/patrick/352/examples/v2
$ ls
$ tar xvf ../arExample.tar arExample/zapscript arExample/java-src/args1.java
x arExample/zapscript, 105 bytes, 1 tape blocks
x arExample/java-src/args1.java, 199 bytes, 1 tape blocks
$ ls -R
.:
arExample/
./arExample:
java-src/ zapscript*
./arExample/java-src:
args1.java
```

- Some thought has to be given to how an archive is “rooted”. arExample.tar is rooted at arExample.
  - Extracting files from arExample.tar causes the directory arExample to be created.
  - Extracting individual files/directories (as above) requires arExample to be stated in the file name.

Assorted (useful) Utilities — tar (continued):

- Can “root” the archive at the current directory:

```
$ pwd
/home/patrick/352/examples/arExample
$ tar cvf ../arExample2.tar .
a ./ OK
a ./zapscrip1 1K
a ./count/ OK
a ./count/countBoth 1K
a ./count/countBoth2 1K
a ./count/countBoth3 1K
a ./count/countBoth4 1K
a ./java-src/ OK
a ./java-src/args.java 1K
a ./java-src/args1.java 1K
a ./java-src/args2.java 1K
$ cd ..

$ mkdir v3
$ cd v3
$ tar xvf ../arExample2.tar
x ., 0 bytes, 0 tape blocks
x ./zapscrip1, 105 bytes, 1 tape blocks
x ./count, 0 bytes, 0 tape blocks
x ./count/countBoth, 156 bytes, 1 tape blocks
x ./count/countBoth2, 120 bytes, 1 tape blocks
x ./count/countBoth3, 160 bytes, 1 tape blocks
x ./count/countBoth4, 159 bytes, 1 tape blocks
x ./java-src, 0 bytes, 0 tape blocks
x ./java-src/args.java, 193 bytes, 1 tape blocks
x ./java-src/args1.java, 199 bytes, 1 tape blocks
x ./java-src/args2.java, 199 bytes, 1 tape blocks
$ ls -R
.:
count/ java-src/ zapscrip1*

./count:
countBoth* countBoth2* countBoth3* countBoth4*

./java-src:
args.java args1.java args2.java
```

Assorted (useful) Utilities — tar (continued):

- Can use **tar** to make a checkpoint of work in progress. For example:

```
$ pwd
/home/patrick/352
$ tar cvf ../352.Sep27.tar .
a ./ OK
a ./examples/ OK
a ./examples/Hello.java 1K
...
```

- Or, you might do:

```
$ cd
$ tar cvf 352.Sep27.tar 352
a 352/ OK
a 352/examples/ OK
a 352/examples/Hello.java 1K
...
```

- Mistakes to avoid. Why?

```
$ tar cvf 352.tar .
$ tar cvf ~/352.tar ~/352
```

It is (almost) always a mistake to create an archive that has absolute pathnames in it!!

Assorted (useful) Utilities — tar (continued):

- Can provide a specific list of names to **tar**. We can use this to archive only the Java files:

```
$ tar cvf jfiles.tar $(find . -name *.java)
a ./Hello.java 1K
a ./args.java 1K
a ./lc.java 1K
a ./today/Hello.java 1K
a ./today/args.java 1K
a ./today/lc.java 1K
a ./tilde.java 1K
a ./eval.java 1K
a ./args1.java 1K
a ./args2.java 1K
a ./both/lc.java 1K
a ./arExample/java-src/args.java 1K
a ./arExample/java-src/args1.java 1K
a ./arExample/java-src/args2.java 1K
```

- **tar** can read from standard input and output to standard output. Here is an example that uses both to make a copy of a directory tree:

```
$ pwd
/home/patrick/352
$ tar cf - . | (mkdir ../352.2; cd ../352.2; tar xf -)
```

Assorted (useful) Utilities — tar (continued):

- There are more options for **tar**.
- Those covered here comprise a very useful, and common, subset.
- There are multiple versions of **tar** (unfortunately).
  - **tar** on Solaris is `/usr/bin/tar`
  - GNU **tar** is the default on Fedora and OS X (and there are manpages on each for GNU **tar**).
- GNU **tar** has additional options, the most useful of which is:

- **z, Z** to compress (or uncompress) files. For example:

```
$ tar cf 352.tar 352
$ ls -l 352.tar
-rw-r--r-- 1 patrick dept 8648704 Sep 27 10:38 352.tar
$ /usr/local/bin/tar czf 352.tz 352
$ ls -l 352.tz
-rw-r--r-- 1 patrick dept 3448747 Sep 27 10:38 352.tz
```

- **z** invokes the **gzip** (or **gunzip**) compression, **Z** invokes **compress** (or **uncompress**).

### Assorted (useful) Utilities — **compress, gzip**:

- Standard Unix utility.
  - Not the best compression, but is fairly fast.

```
$ ls -l 352.tar
-rw-r--r-- 1 patrick dept 8648704 Sep 27 10:38 352.tar
$ time compress 352.tar
real 0m1.17s
user 0m0.97s
sys 0m0.08s
$ ls -l 352.tar.z
-rw-r--r-- 1 patrick dept 3802107 Sep 27 10:38 352.tar.z
$ time uncompress 352.tar.Z
real 0m0.83s
user 0m0.47s
sys 0m0.12s
```
- Most Unix systems have additional compression utilities. Example: **gzip, gunzip**:

```
$ time gzip 352.tar
real 0m3.50s
user 0m3.23s
sys 0m0.05s
$ ls -l 352.tar.gz
-rw-r--r-- 1 patrick dept 3448808 Sep 27 10:38 352.tar.gz
$ time gunzip 352.tar.gz
real 0m1.04s
user 0m0.35s
sys 0m0.05s
```

## The grep Family

Reading: *Learning the Korn Shell*, section 4.5.2 entitled “Patterns and Regular Expressions”  
*Unix Power Tools*, Chapter 13 (grep tools), Chapter 32 (regular expressions)

### **fgrep**:

- Review: **fgrep** prints lines that match a pattern.

```
$ fgrep println *java
Hello.java: System.out.println("Hello, world!");
args.java: System.out.println("" + args[i] + "");
args1.java: System.out.println("" + args[i] + "");
args2.java: System.out.println("" + args[i] + "");
eval.java: System.out.println(value);
lc.java: System.out.println(count);
tilde.java: System.out.println(
tilde.java: System.out.println(new File("~/352/examples/args.java").exists());
```
- Each **.java** file is processed line-by-line. If a line contains **println**, it is printed (along with the file name).
- Can take input from stdin:

```
$ cat *java | fgrep println
System.out.println("Hello, world!");
System.out.println("" + args[i] + "");
System.out.println("" + args[i] + "");
System.out.println("" + args[i] + "");
System.out.println(value);
System.out.println(count);
System.out.println(
System.out.println(new File("~/352/examples/args.java").exists());
```

### **fgrep** (continued):

- ```
$ fgrep -l println *.java
Hello.java
args.java
args1.java
args2.java
eval.java
lc.java
tilde.java
```
- The **-v** flag causes inversion — lines that do not match are printed:

```
$ fgrep -v println args.java
public class args {

    public static void main(String args[]) {
        for (int i = 0; i < args.length; i++)
        } // main

} // class args
```
 - Problems:
 - Find lines initializing an **Object** array with “**new**” of some sort.
 - Find **println** calls that do not contain a double-quote but only consider Java files that contain the string “**Test**”.

fgrep (continued):

- Other handy options (among many!):
 - **-w** searches for whole “words”. (GNU **fgrep** only)
 - **-c** prints a count of matching lines.
 - **-C num** prints surrounding lines (GNU **fgrep** only).
 - **-n** prints line numbers.
 - **-e** is used to escape a search string that starts with a hyphen: **fgrep -e -xray ...**
 - **-f file** reads search patterns from a file.

Regular expressions:

- Definition: language is a set of strings. The set may be infinite. (This is a CS definition :-).
- The Chomsky hierarchy of languages looks like this:
 - Type 0: Unrestricted languages.
 - Type 1: Context-sensitive languages.
 - Type 2: Context-free languages.
 - Type 3: Regular languages.
- Natural languages are unrestricted languages (roughly speaking) that can only be specified by unrestricted grammars.
- Programming languages are (usually) context-free languages — Can be specified by a context-free grammar.
 - Which has very restrictive rules.
 - Every Java program is a string in the context-free language that is specified by the Java grammar.

Regular expressions (continued):

- A *regular language* is a very limited kind of context free language that can be described by a *regular grammar*.
 - A regular language can also be described by a *regular expression*.
- A regular expression is simply a string that may contain metacharacters. Example:
`^[A-Z].*[ab]\?\|[a-fA-F]\{3,5\}`
- Regular expressions have some similarities to filename wildcards
 - But the two facilities are used in different contexts and
 - Behave very differently in most cases.

Regular expressions (continued):

- **grep** behaves much like **fgrep** but:
 - Assumes its first argument is a regular expression instead of literal text.
- Alphanumeric characters, and some special characters, match themselves.
 - Thus, **fgrep abc123 filename** and **grep abc123 filename** find the same matches.
 - **abc123** is a regular expression that describes a language that contains exactly one string: "abc123".
- Simplest RE (regular expression) metacharacter is the period (or dot). It matches any one character.

```
$ grep "a.b.c" -cs352/spring06/UnixExamples/words
albacore
barbecue
canvasback
drawback
iambic
playback
snapback
$ grep "a.b.c.e" -cs352/spring06/UnixExamples/words
barbecue
```
- What is matched by the following: **grep "....." -cs352/spring06/UnixExamples/words**
- Note: As a matter of habit, enclose regular expressions in double quotes.

Regular expressions (continued):

- A caret (^) at the beginning of a regular expression requires that the following RE appear at the start of a line.

```
$ grep "^dw" -cs352/fall106/UnixExamples/words
dwarf
dwarves
dwell
dwelt
dwindle
```
- A dollar sign (\$) requires that the line end with the preceding RE:

```
$ grep "roads$" -cs352/fall106/UnixExamples/words
abroad
broad
byroad
crossroad
highroad
inroad
railroad
road
```
- Problems:
 - Print words that are exactly four characters long.
 - How many words have an **a** as their third character?
 - Does the file **1c.java** contain any empty lines?

Regular expressions (continued):

- **R*** matches zero or more occurrences of the regular expression **R**.
\$ **grep "rail*" -cs352/fall106/UnixExamples/words | more**
afraid
algebraic
appraisal
appraise
arraign
braid
Braille
...
• Problems:
 - Find words that start with **a** and end with **b**.
 - Describe lines matched by **^a*b.*c..\$**
 - Describe lines matched by **if.*f(.*)**
 - Find words that have all vowels in order.
 - What does **a*.\.** match?
- Tip: To experiment with **grep**, run it with no filename and type lines of test input. Lines that match are echoed.
 - Example: What does the RE ***.c** match? Try:
grep "*.c"

Regular expressions (continued):

- There are precedence rules for regular expression metacharacters.
- The ***** metacharacter has higher precedence than juxtaposition (one character next to another).
 - Example: **a.*b** is interpreted as **a(.*)b** rather than **(a.)*b**.
- In part, this means that ***** is greedy. It will try to match as many characters as possible first. If the pattern then fails, the ***** will match one less character and try again, etc.
 - Example: **ab*b** on the string **abbbb** will
 - **a** matches **a**
 - **b*** matches **bbbb**
 - fail to match the last **b** in the RE.
 - backtrack: **b*** matches **bbbb**
 - **b** matches **b**
 - Pattern succeeds.

Regular expressions (continued):

- Sets: (We have seen these before)
 - **[characters]** is a RE that matches any one of the characters in the set.
 - **[^characters]** matches any character that is not in the set. (It matches the *complement* of the set.)
 - **^[AETIOU].*[0-9]\$** matches any line that starts with a capital letter and does not end in a digit.
- Problems:
 - Find words that contain a capital letter in a position other than the first.
 - Strings that match **[A-Za-z_][A-Za-z_0-9]*** commonly occur in programs. What are they?
 - Find lines that have **if** and one of these comparisons: **!=, ==, <=, >=**
 - Are there any non-alphabetic characters in **-cs352/fall106/UnixExamples/words**?

Regular expressions (continued):

- Many programs (not just the **grep** family) have at least some support for regular expressions.
- But, the set of metacharacters can vary from program to program (ugh!). Examples:
 - **grep** on Fedora and OS X considers **+, ?, |** and **()**'s to have special meaning only when escaped.
 - **grep** on Solaris does not consider **+, ?, |** and **()**'s to have any special meaning.
 - **egrep** considers the characters **+, ?, |** and **()**'s to have special meaning unless escaped.
- There are three classic members in the **grep** family: **grep**, **fgrep**, and **egrep**.
 - Performance and memory limitations at the time they were created drove the development of the three.
 - **grep** patterns are limited RE's. **grep** uses a compact nondeterministic algorithm.
 - **egrep** patterns are full regular expressions. **egrep** uses a fast deterministic algorithm that sometimes needs exponential space.
 - **fgrep** patterns are fixed strings. **fgrep** is fast and compact.
- There are additional **grep** variants, plus programs that use **grep**-like algorithms. Try **man -k grep**.
 - **agrep** is one that can do approximate matches. (And was developed here!)
- Pick one of: **grep**, **egrep**, **agrep**, ...and get familiar with the RE's that it accepts.
 - Keep **fgrep** as a tool when searching for fixed strings; especially when the strings contain characters that would otherwise be metacharacters!

Regular expressions (continued):

- These may not work with **grep** (see previous slide).
- If **R** is a regular expression:
 - **R+** matches one or more instances of **R**.
 - **R?** matches zero or one instance of **R**.
 - Example: **[a-z][0-9]?** matches one or more lower case letters and (possibly) one digit.
- Like *****, **+** and **?** have higher precedence than juxtaposition.
- RE's can be grouped with parentheses to override precedence rules.
 - Example: **(ab)+** matches strings like: **ab**, **abab**, **ababab**, etc.
- Describe the input lines that would be printed by this **egrep** invocation:
`$ egrep "^(ab)+c?(xyz)*? $"`
- **|** provides an "or" capability. Example: **egrep "(one|two|three) (apple|biscuit)s?"**
- There are (many) more regular expression patterns that (various) utilities support. See for example: the **java.util.regex** package and **perl**.
- If you know the following, you can accomplish a lot:
`. * + ? [...] [^...] ^ $`

sed — stream editor:

- Designed to perform one or more transformations on a stream of input text.
- The **s/pattern/replacement/flags** command of **sed** performs simple textual substitution.

```
$ date
Wed Sep 13 22:44:26 MST 2006
$ date | sed "s/O/zero/"
Wed Sep 13 22:44:32 MST 2zero06
$ date | sed "s/O/zero/g"
Wed Sep 13 22:45:zerozero MST 2zerozero6
```

g is the "global" flag. It causes the pattern to be substituted everywhere in the *pattern* space, not just the first occurrence.
- Can use regular expressions with **sed**:

```
lectura-> cal | sed "s/[0-9]\+/<num>/g"
September <num>
Su Mo Tu We Th Fr Sa
<num> <num> <num> <num> <num> <num> <num>
<num> <num> <num> <num> <num> <num> <num>
<num> <num> <num> <num> <num> <num> <num>
<num> <num> <num> <num> <num> <num> <num>
```

sed — stream editor (continued):

- Another regular expression example:

```
lectura-> cat args.java
public class args {

    public static void main(String args[] ) {
        for (int i = 0; i < args.length; i++)
            System.out.println("'" + args[i] + "'");
    } // main

} // class args
lectura-> sed "s/(.*)/()/ < args.java
public class args {

    public static void main() {
        for ()
            System.out.println();
    } // main

} // class args
```
- Problem: Use **sed** to strip **//** comments from a Java source file.
 - Take a naive approach. For example, do not worry about a string literal such as **"some // are here"**.

sed — stream editor (continued):

- In the "pattern" field of **sed**'s **s** command, an **ampersand** specifies the text matched by the "replacement" field.

```
$ date
Wed Sep 13 22:48:44 MST 2006
$ date | sed "s/[0-9]\+/(&)/g"
Wed Sep (13) (22):(49):(03) MST (2006)
```
- Another flag option: Use a number. Indicates the "replacement" is done on only the *n*th item returned by the "pattern":

```
lectura-> cal 2 2006 | sed "s/[12]/Z/3"
February 2006
Su Mo Tu We Th Fr Sa
    1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28
```

Replaces the 3rd occurrence of the pattern (a 1 or a 2) on each line with **Z**.

sed — stream editor (continued):

- Elements of the “replacement” field grouped with parentheses can be plugged into the “pattern” field with \1, \2, etc.
 - The ()’s must be escaped.
 - There can be up to 9 patterns.

```
lectura-> sed "s/\([abc]*\):\([abc]*\)/\2 came after \1/"
aaaaa:ccc
ccc came after aaaaa
aa:cccba
cccba came after aaa
aa:cccddd
ccc came after aaaddd
```

| | | |
|---|---|--|
| lectura-> cat sed-file1 one:two first:second alpha:bravo alpha:bravo:charlie alpha:bravo:charlie:delta alpha:bravo:charlie:delta:echo | lectura-> cat sed-file1 sed "s/(.*):(.*)/\2:\1/" two:one second:first bravo:alpha charlie:alpha:bravo delta:alpha:bravo:charlie echo:alpha:bravo:charlie:delta | lectura-> cat sed-file1 sed "s/([A-Za-z]*):\([.]*\)/ \2:\1/" two:one second:first bravo:alpha bravo:charlie:alpha bravo:charlie:delta:alpha bravo:charlie:delta:echo:alpha |
|---|---|--|

sed — stream editor (continued):

- Any character can be used as the delimiter in separating the “pattern” and “replacement” fields. The character that follows the **s** is assumed to be the delimiter character.

```
lectura-> sed "s\([abc]*\):\([abc]*\)+\2 came after \1+"
abcabc:abcabc:why
abcabc came after abcabc:why
```

- Several **sed** commands may be specified on the command line.

- Each **sed** command uses **-e**.

Note: Use of ; instead of / to separate fields. Necessary when using \1, etc.

- Example: Deletes whole-line comments and strips intra line comments:

```
lectura-> sed -e "/^\\//d" -e "s;///  
;>";" < eval.java
```

d deletes the line that matches the pattern in the “from” field.

Using ;’s here means we do not have to escape the /’s in the pattern. Compare with:
-e "s/\//.*/"

sed — stream editor (continued):

- A sequence of **sed** commands can be put into a file:

```
lectura-> cat stripcmt
/^\\//d
s;///  
;>";
lectura-> sed -f stripcmt eval.java
```

- **sed** has many more capabilities than are discussed here.

- For example, print the first and last lines of standard input:

```
lectura-> cat eval.java | sed -n -e 1p -e '$p'
// Simple operand evaluation. Takes three command-line arguments,
} // class eval
```

- For example, print the fourth and last lines of standard input:

```
lectura-> cat eval.java | sed -n -e 4p -e '$p'
public class eval {
} // class eval
```