

UNIX

CSC 352, Fall 2015
The University of Arizona
William H. Mitchell
whm@cs

What is UNIX?

At Bell Labs in 1969 Ken Thompson created a tiny operating system that came to be known as UNIX.

During the 1970s UNIX gradually grew and evolved, and spread into the computer science community.

In the 1980s and 1990s UNIX became an immensely popular platform for software R&D and later, enterprise computing.

Today, various UNIX-like operating systems run on everything from tiny devices to the most powerful computers made.

UNIX command-line tools are available on just about every platform used by programmers.

What is UNIX?

Some hallmarks of UNIX:

- Pre-emptive multi-tasking of processes
- Full support for multiple simultaneous users
- Utilities work well in combination with others
- APIs that combine simplicity, elegance, and power
- Devices are treated as files
- The system is stable and resilient
- The keyboard is alive and well
- Sophisticated users are not encumbered
- Casual users are frustrated

UNIX Timeline

- 1965** Researchers from Bell Labs and other organizations begin work on Multics, a state-of-the-art interactive, multi-user operating system.
- 1969** Bell Labs researchers, losing hope for the viability of Multics due to performance issues, withdraw from the project.

One of the researchers, Ken Thompson, finds a little-used DEC PDP-7, and in a month implements a simple operating system comprising a kernel, a command interpreter, an editor, and an assembler.

Other Bell researchers, most notably Dennis Ritchie, are attracted to Thompson's system and contribute to it.

- 1970** Peter Neumann suggests the name "Unics" for Thompson's operating system, a pun on "Multics". A DEC PDP-11 is acquired for further development of UNIX.

UNIX Timeline, continued

1971 In addition to supporting research, the PDP-11 running UNIX hosts a word processing project: the preparation of patent applications.

Work begins on the C programming language.

1973 UNIX is rewritten in C.

1975 Ken Thompson takes a sabbatical and teaches at U. C. Berkeley. He gets some students, including Bill Joy, interested in UNIX.

1978 Seventh Edition UNIX (V7), incorporating a goal of portability, is released. (Today: Some say that V7 was the classic UNIX.)

Bill Joy assembles the first Berkeley Software Distribution, featuring a Pascal compiler and Joy's **ex** line editor.

UNIX Timeline, continued

- 1979** Building on Bell Lab's UNIX/32V, UCB produces a version of UNIX that takes advantage of virtual memory on the DEC VAX-11/780. It is released as 3BSD.
- 1981** VAXs running 4.1BSD are the system of choice for computer science departments everywhere.
- 1982** Sun Microsystems is founded; Bill Joy leaves UCB to head Sun's software development. Sun produced the first good UNIX workstation, in my opinion.
- 1983** 4.2BSD is released. Most notable: support for TCP/IP networking.
- Richard Stallman announces the GNU project and later founds the Free Software Foundation. (GNU's Not UNIX.)
- 1984** A federal court decree allows AT&T to get into the computer business; AT&T releases UNIX System V.

UNIX Timeline, continued

- 1984** The X Window System emerges at MIT. It eventually becomes widely used for portable graphics software.
- 1988** IEEE Standard 1003.1-1988 is approved. It came to be known as POSIX.1 (Portable Operating System Interface).
- 1989** AT&T System V R4 (SVR4) is released, merging the System V and BSD development lines.
- 1991** comp.os.minix: "Hello everybody out there using minix – I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. ..."
—Linus Torvalds, a student at the University of Helsinki
- 1993** AT&T sells UNIX System Laboratories to Novell; Novell conveys "UNIX" trademark to X/Open, a standards organization.

Today

- The UNIX brand can be legally applied to any system that has been verified to comply with the "Single UNIX Specification" (SUS). (The SUSv3 (UNIX 03) specification is 3700 pages and covers 1742 interfaces.)
- Only AIX, HP-UX, K-UX, OS X, and Solaris satisfy UNIX 03.
- FreeBSD and NetBSD are the descendants of Berkeley UNIX.
- The IEEE/ISO POSIX standards facilitate writing software that is portable between a wide range of UNIX and non-UNIX systems.
- Linux keeps getting bigger and better.

The shell—Part 1

The shell—basics

Users typically interact with UNIX via a "shell".

A reasonable definition for *shell*:

A command-line based environment for execution and control of programs.

In essence, a shell is a program that is used to run other programs.

There are many different shells but a number of capabilities are common to all popular shells:

- Command execution
- Redirection of input and output
- Piping
- Wildcard expansion
- Process control
- Command recall and editing
- *Turing-complete* (can be used to write "any" program)

Lots of shells

There are many Unix shells. Here are some common ones:

- Bourne Shell (**sh**)
- C Shell (**cs**h)
- **tcsh** ("Enhanced C Shell", "TENEX C Shell")
- Korn Shell (**ksh**)
- Bourne-Again Shell (**bash**)
- Z Shell (**zsh**)

See also https://en.wikipedia.org/wiki/Unix_shell

We'll be using **bash** because it is...

- Widely used
- A typical shell
- Full-featured
- POSIX-compliant

Running **bash** on lectura

Extra credit!

For two points of extra credit on Assignment 1 run the following command in **bash** on lectura:

```
/cs/www/classes/cs352/fall15/bin/i-ran-bash
```

Due date/time: same as Assignment 1.

Details:

(1) Use the following command to confirm that you're running bash.

```
% echo $SHELL  
/bin/bash
```

(2) You'll see something like this: (your prompt may differ from "%")

```
% /cs/www/classes/cs352/fall15/bin/i-ran-bash
```

Give me a moment...

A receipt has been mailed to *YOUR-NETID*@email.arizona.edu. Keep it until you see your points on D2L.

```
%
```

Mail to 352f15@cs.arizona.edu if you have trouble. Don't panic!

Running **bash** on lectura

We'll be using the Linux machine named "lectura" for much of our work.

By virtue of being enrolled in this class you should already have a CS computing account with the same name as your UA NetID.

On the page <http://cs.arizona.edu/computing/services> use "Change my Unix shell (bash/tcsh/ksh)" to be sure that **bash** (/bin/bash) is your shell.

(Note: a quick glance on 8/23 showed that all of you already have **bash** as your login shell.)

If you've forgotten the password for your CS account, use "Reset my forgotten Unix password" on the same page to reset it.

Note that there's no connection between your NetID password and your password on lectura. (And I recommend different passwords for them.)

Running bash on lectura—Macs

If you're on a Mac, start Terminal and use `ssh` to login to lectura:

1. bash prompt on Mac

2. `ssh yourNetID@lec.cs.arizona.edu`

```
whm - ssh - 80x21
Last login: Thu Aug 13 01:42:30 on ttys003
~ 4318 % ssh whm@lec.cs.arizona.edu
whm@lec.cs.arizona.edu's password:
-----
New Password/passphrases rules

At least eight characters (but more is highly recommended)
Both upper and lower case letters
At least one number
At least one special character (e.g., !@#$$%^&*()_+|~-=\`{}[]:~<>?,./)
-----
Last login: Thu Aug 13 01:43:13 2015 from on-campus-115-224.vpn.arizona.edu
whm@lectura ~ 4991 % whoami
whm
whm@lectura ~ 4992 % hostname
lectura.cs.arizona.edu
whm@lectura ~ 4993 %
```

3. No echo/feedback while typing password

4. bash prompt on lectura!

NOTE: Your bash prompts may differ!

Running **bash** on **lectura**—Windows

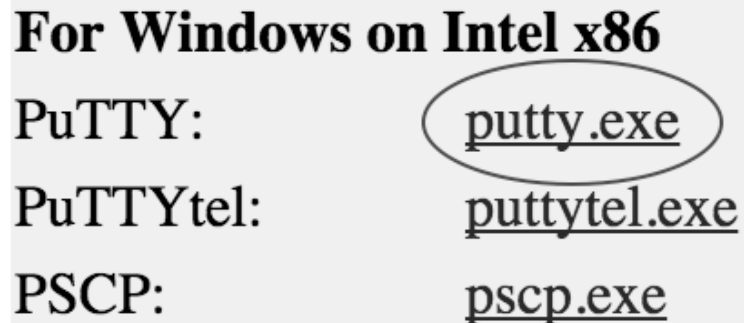
"PuTTY" is a free Telnet/SSH client that I recommend for connecting to **lectura** from a Windows machine.

If you Google for "putty", the first hit should be this:

PuTTY Download Page

- www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Download **putty.exe**:



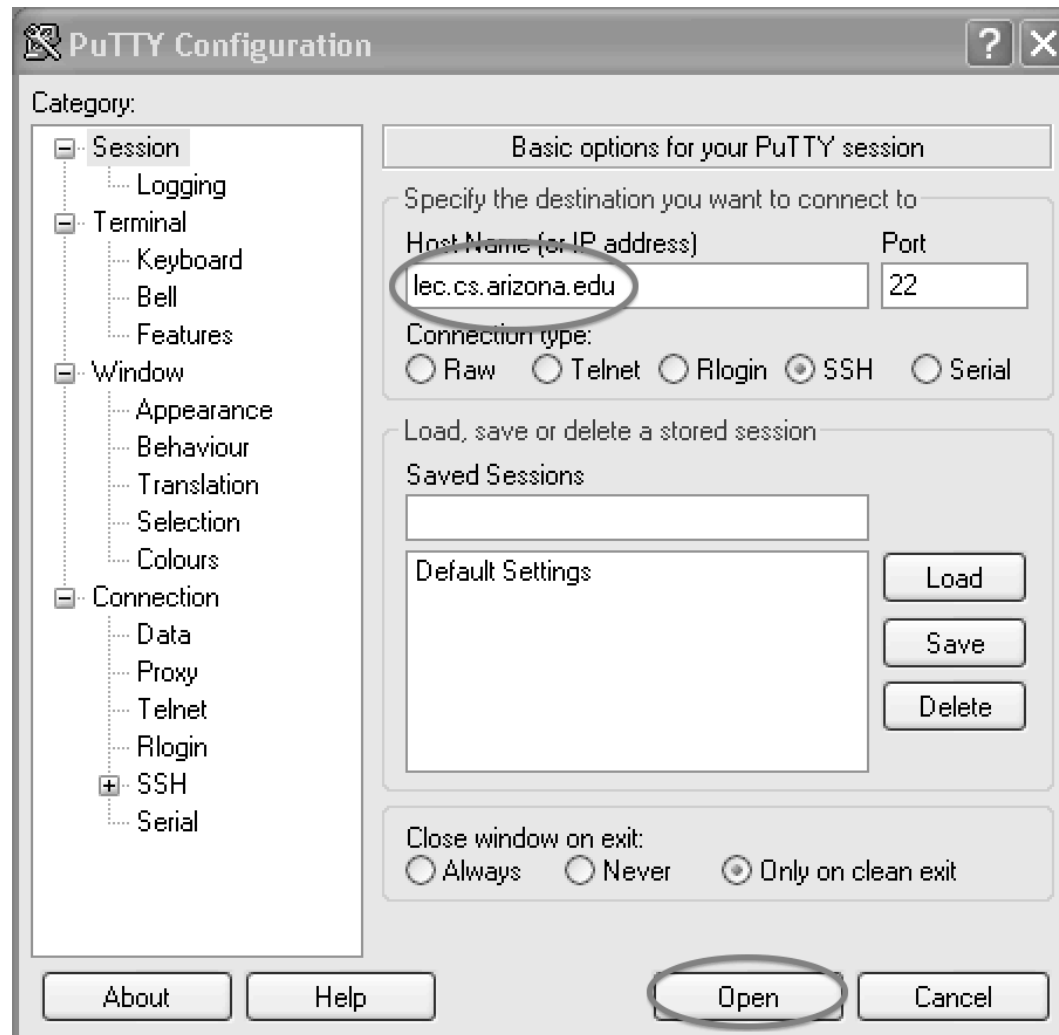
For Windows on Intel x86

PuTTY:	<u>putty.exe</u>
PuTTYtel:	<u>puttytel.exe</u>
PSCP:	<u>pscp.exe</u>

putty.exe is just an executable file; there's no installer. Save **putty.exe** to a convenient place, perhaps your Desktop.

Running **bash** on **lectura**—Windows

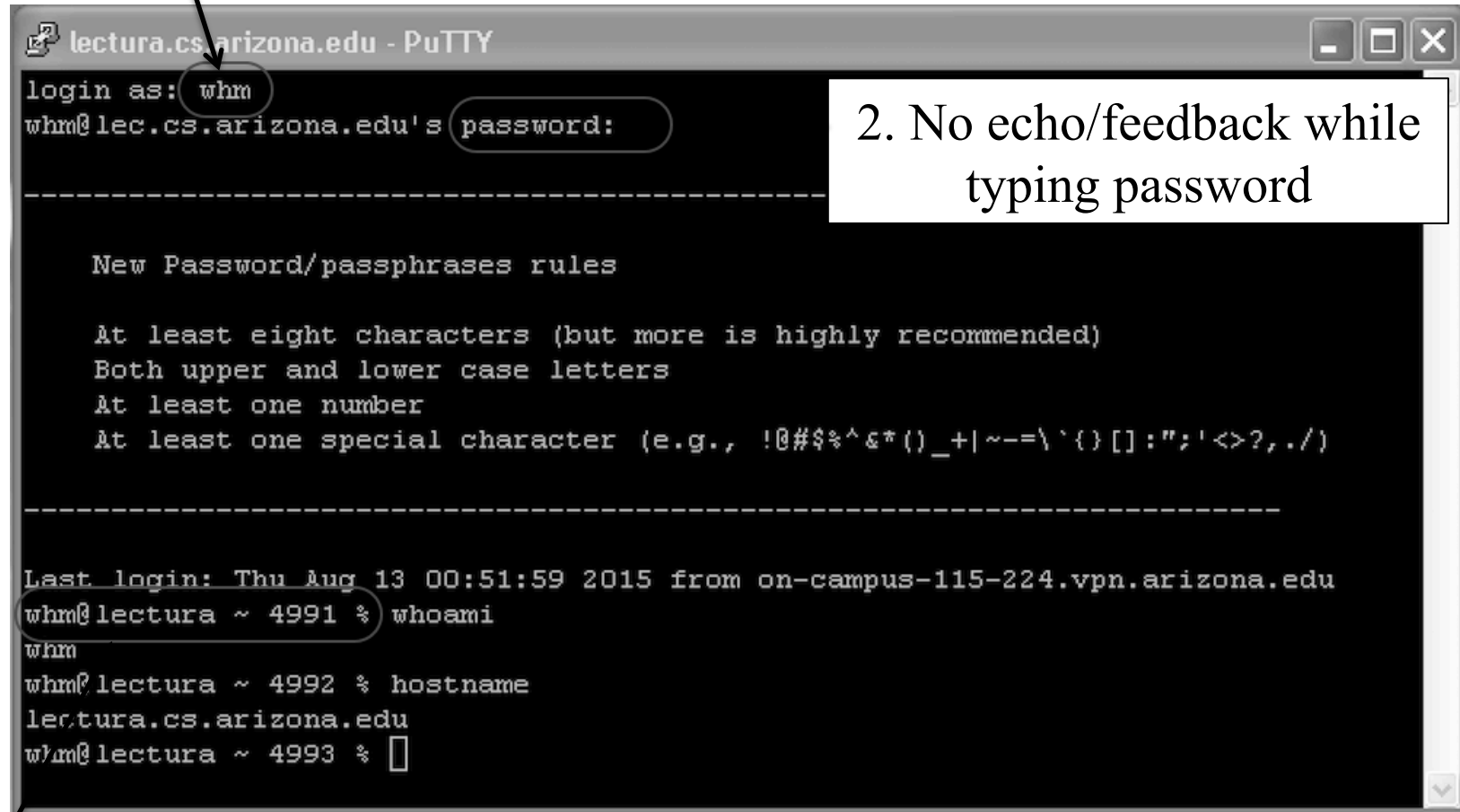
Click on **putty.exe** to run it. In the dialog that opens, fill in **lec.cs.arizona.edu** for Host Name and click **Open**.



Running **bash** on **lectura**—Windows

Enter your NetID and password in the window that PuTTY opens:

1. Your NetID



```
lectura.cs.arizona.edu - PuTTY
login as: whm
whm@lec.cs.arizona.edu's password:
-----
New Password/passphrases rules

At least eight characters (but more is highly recommended)
Both upper and lower case letters
At least one number
At least one special character (e.g., !@#$%^&*()_+|~-=\`{}[]:~<>?,./)
-----

Last login: Thu Aug 13 00:51:59 2015 from on-campus-115-224.vpn.arizona.edu
whm@lectura ~ 4991 % whoami
whm
whm@lectura ~ 4992 % hostname
lectura.cs.arizona.edu
whm@lectura ~ 4993 %
```

2. No echo/feedback while typing password

3. **bash** prompt on **lectura**!

NOTE: Your **bash prompts may differ!**

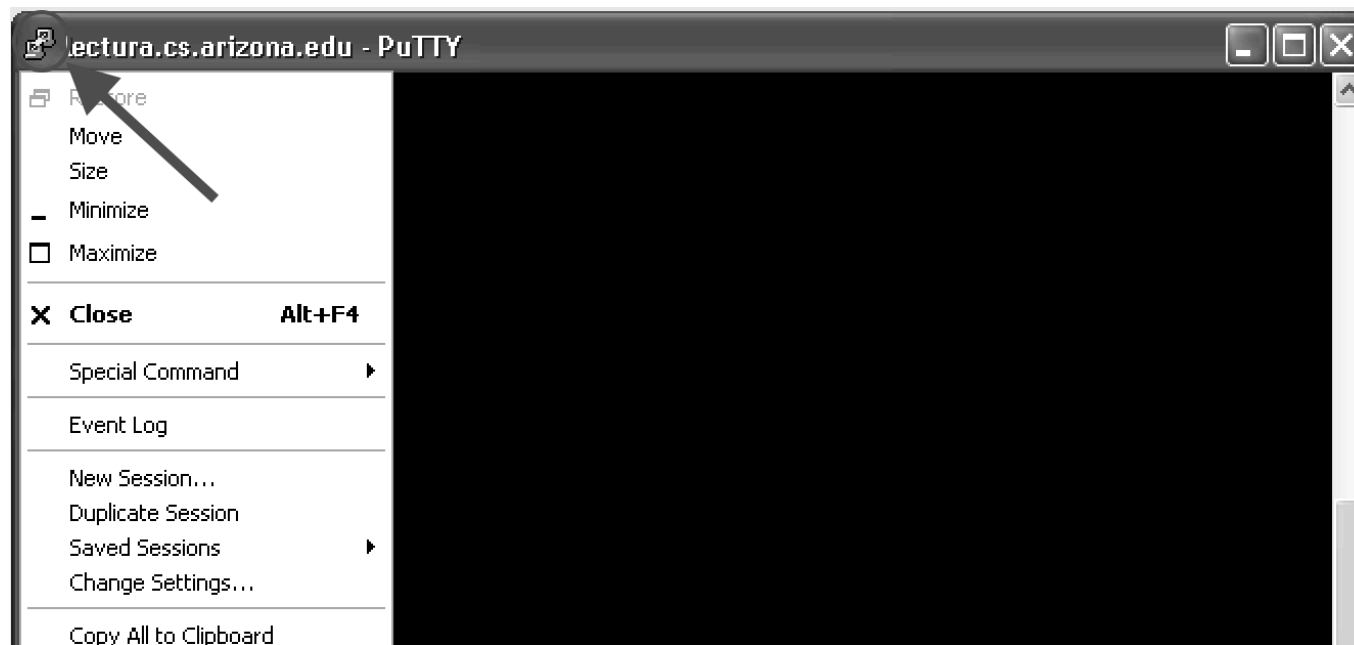
PuTTY notes

To copy text from a PuTTY window to the Windows clipboard, simply click and drag over it, like selecting text in any other Windows application.

Note: Copy is implicit with selection; do not hit ^C!

To paste text from the clipboard into PuTTY, do a right-click.

A number of PuTTY features can be accessed via the "system menu" in the upper left corner of the window.



Running **bash** on lectura--odds and ends

You can have any number of login sessions active at once. It's often handy to run **bash** in one window and keep an editor open in another.

Use Control-C (^C) to kill a currently running command.

Type **exit** or close the window to terminate your lectura login session.

If you're running a different shell, like **tcs**h, and don't want to make **bash** your default, type **bash** at your shell's prompt.

We'll later learn about some improvements:

- Password-less login using **ssh** key pairs
- Adding an **/etc/hosts** entry for lectura so you can type just **lec** instead of **lec.cs.arizona.edu**.
- Handling a NetID that differs from your Mac username.

How can I run **bash** on my machine?

As we've seen, starting Terminal on a Mac opens a window running **bash**.

If you've got a Windows machine, Cygwin(.com) provides a huge number of Unix utilities, including **bash**, that run on Windows. I love Cygwin!

- After Cygwin is installed, use Cygwin Terminal to open a window with **bash**.
- Watch for a Piazza post with some details about installing Cygwin, and remind me if it doesn't appear soon.

On a Linux machine in a CS lab, Terminal opens a window running **bash**.

bash command-line basics

Executing commands

Typing a command name at the **bash** prompt and pressing the **ENTER** key causes the command to be executed.

The command's output, if any, is displayed on the screen. Examples:

```
% hostname
```

```
lectura.cs.arizona.edu
```

```
% whoami
```

```
whm
```

```
% true
```

```
% date
```

```
Sat Aug 15 18:54:39 MST 2015
```

```
% ps
```

PID	TTY	TIME	CMD
22758	pts/18	00:00:00	bash
30245	pts/18	00:00:00	ps



Command-line arguments

Most commands accept one or more *arguments*:

```
% cal 9 2015
```

```
    September 2015
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
```

```
% echo Hello, world!
```

```
Hello, world!
```

```
% factor 223092870
```

```
223092870: 2 3 5 7 11 13 17 19 23
```

Note: These slides will usually show a blank line between commands to improve readability, but `bash` outputs the prompt immediately following the last character of a command's output.

Arguments, continued

For many commands the arguments are file names.

```
% cat Hello.java
public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello, world!");
    }
}
```

```
% javac Hello.java
```

```
% java Hello
Hello, world!
```

```
% rm Hello.class
```

```
% java Hello
Error: Could not find or load main class Hello
```

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

Arguments, continued

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

```
% fgrep Hello Hello.java
```

```
public class Hello {  
    System.out.println("Hello, world!");  
}
```

```
% fgrep Hello Hello.java Test.java
```

```
Hello.java:public class Hello {  
Hello.java: System.out.println("Hello, world!");  
}
```

```
% fgrep Waldo Hello.java Test.java
```

```
%
```

Does **fgrep** exhibit "silence is golden"?

Note: There is a family of **greps**. We'll use **fgrep** to start with because it doesn't interpret the search string as a *regular expression*.

Command-line options

Many commands accept *options* that adjust the behavior of the command.

Options almost always begin with a '-' (minus sign). Options are usually specified immediately following the command.

Examples:

```
% date
```

```
Thu Jan 13 02:19:20 MST 2005
```

```
% date -u
```

```
Thu Jan 13 09:19:22 UTC 2005
```

```
% wc Hello.java
```

```
5      14      127 Hello.java
```

```
% wc -l -w Hello.java
```

```
5      14 Hello.java
```

We can say that **wc -l -w Hello.java** has two options and one *operand*.

Options, continued

Some options have an associated argument. (An "option argument".)

Compile Hello.java with verbose output and put the resulting .class file in the (existing) directory named work:

```
% javac -verbose -d work Hello.java
[parsing started RegularFileObject[Hello.java]]
[parsing completed 13ms]
...lots more....
[wrote RegularFileObject[work/Hello.class]]
[total 286ms]
```

Find files modified in the last 48 hours that are longer than 1024 bytes:

```
% find . -type f -mtime -2 -size +1k
./352.notes
./intro.pptx
./open.notes
./unix.pptx
```

Options, continued

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

```
wc -l -w Hello.java
```

```
wc -lw Hello.java
```

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

```
wc --words --lines Hello.java
```

Options, continued

Whitespace is often significant in command lines. For example, the following commands are all invalid: (Try them!)

```
% date-u
```

```
% wc -l-w Hello.java
```

```
% wc -- notes Hello.java
```

We can think of a command line as a series of "words". The *man page* for `bash` has this definition for "word":

A sequence of characters considered as a single unit by the shell.

Options are sometimes called "flags".

Example: "Run `date` with the `-u` flag."

Oddballs

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

jar, the Java archive tool, requires certain options to come first, and allows them to not be preceded by '-':

```
% jar tvf cloudcoderApp.jar | head -3
    0 Fri Aug 14 18:18:18 MST 2015 META-INF/
   161 Fri Aug 14 18:18:16 MST 2015 META-INF/MANIFEST.MF
    0 Fri Aug 14 15:18:54 MST 2015 org/
...

```

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
```

POSIX guidelines for command-line arguments can be found here:
http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

Sidebar: Java argument handling

When a Java program is run, the shell, the operating system kernel, and the Java run-time system arrange for the command line arguments to appear as an array of strings that is passed to `main`.

Here is a Java program that prints its arguments:

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

Interaction:

```
% java args -a --test x.java
|-a|
|--test|
|x.java|
%
```


Sidebar: Try the examples!

I strongly recommend you try at least one example on every slide!

We'll learn about paths, the `cp` command, *symlinks*, etc. later but for now you can run `args.java` like this on lectura:

```
% cp /cs/www/classes/cs352/fall15/java/args.java .
```

```
% javac args.java
```

```
% java args Hello, world!  
|Hello,|  
|world!|
```

Don't forget the period!

`args.java` is also accessible on the web:

```
http://cs.arizona.edu/classes/cs352/fall15/java/args.java
```

IMPORTANT: In these slides I'll reference files like this:

```
fall15/java/args.java
```

fall15/... means either of these

Metacharacters

Many non-alphanumeric characters have special meaning to shells.

```
% java args :)
```

```
bash: syntax error near unexpected token `:)'
```

Characters that have special meaning are often called *metacharacters*.

Here are the **bash** metacharacters:

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

Metacharacters, continued

If an argument has metacharacters or whitespace we suppress their special meaning by enclosing the argument in quotes.

```
% java args ' :) ' "' '" ' x ' "y " z" 'z"
```

| :) |
| ' ' ' |
| " |
| x |
| y |
| z ' z |

Note that the enclosing quotes are consumed by the shell. args never sees them!

We'll see later that some metacharacters are still interpreted even when surrounded with double quotes. For the time being, always use apostrophes to avoid any surprises.

Metacharacters, continued

An alternative to wrapping with quotes is to use a backslash to "escape" each metacharacter.

If a character is preceded by a backslash, its special meaning, if any, is suppressed.

```
% java args :\)  \'\"\\  x\ y  \x\y\z
| : ) |
| ' " \ |
| x y |
| xyz |
```

Note that it's not an error to escape ordinary characters like **x**, **y** and **z**.

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 '-' and are often single letters; single letter options can often be combined.

Options sometimes have arguments themselves. ("option arguments")

The ordering of options is usually not important.

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.

All-in-all, there are somewhat firm conventions but no hard rules about options and operands.

Reminder: Use `^C` to immediately terminate a command.

Command-line editing and shortcuts

bash supports simple command line recall and editing with the "arrow keys" but many control-key and escape sequences have meaning too. Here are a few:

- `^A/ ^E` Go to start/end of line.
- `^W` Erase the last "word".
- `^U` Erase whole line. (`^C` works, too.)
- `^R` Do incremental search through previous commands.
- `ESC-f/b` Go forwards/backwards a word. (Two keystrokes: **ESC**, then **f**)
- `ESC-.` Insert last word on from last command line. (Very handy!)

Do **bind -p** to see all the bindings. This facility uses the GNU readline mechanism and bindings can be overridden in `~/.inputrc`. (Do **man readline** for lots of details.)

bash also does command and filename completion with **TAB**:

Hit **TAB** to complete to longest unique string.

If a "beep", hit **TAB** a second time to see alternatives.

The man command

The man command

The **man** command displays documentation for commands (and more). Here is an abridged example—the "man page" for **cat**:

```
% man cat
```

```
CAT(1)
```

```
User Commands
```

```
CAT(1)
```

```
NAME
```

```
cat - concatenate files and print on the standard output
```

```
SYNOPSIS
```

```
cat [OPTION]... [FILE]...
```

```
DESCRIPTION
```

```
Concatenate FILE(s), or standard input, to standard output.
```

```
-A, --show-all  
    equivalent to -vET
```

```
...
```

```
With no FILE, or when FILE is -, read standard input.
```

man uses **less** to display pages. Type space to go forwards, **b** to go backwards. Type **/STRING**<ENTER> to search for a string, then **n** to search for the next occurrence. **h** (for help) shows lots more **less** commands. (Try it!)

Manual sections

The UNIX "manual" is divided into these sections: (from `man man`)

- 1 User commands
- 2 System calls (functions provided by the kernel)
- 3 Library calls (functions within program libraries)
- 4 Special files (usually found in `/dev`)
- 5 File formats and conventions eg `/etc/passwd`
- 6 Games
- 7 Miscellaneous (including macro packages and conventions), e.g. `man(7)`, `groff(7)`
- 8 System administration commands (usually only for root)
- 9 Kernel routines [Non standard]

Recall that `man cat` showed `CAT (1)`. That "`(1)`" tells us that `cat` is a user command.

`man malloc` shows `MALLOC (3)`. That "`(3)`" tells us that `malloc` is a library function.

man -k

A very handy **man** option is **-k**, which specifies a keyword to search for in the "**NAME**" entries for all man pages.

Example: ("What was that calendar printing command??")

```
% man -k calendar
```

```
cal (1) - displays a calendar and the date of Easter
calendar (1) - reminder service
difftime (3posix) - compute the difference between two
                    calendar time values
ncal (1) - displays a calendar and the date of Easter
zshcalsys (1) - zsh calendar system
```

Some man page names appear in more than one section of the manual. For example, **printf** appears in sections 1 and 3. The **-s** option selects the entry in the specified section.

```
man -s 1 printf
man -s 3 printf
```

Most manual sections have an **intro** page that provides an overview of the section. For example, try **man -s 2 intro**

Built-in help for commands

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

```
% wc --help
```

```
Usage: wc [OPTION]... [FILE]...
```

```
or: wc [OPTION]... --files0-from=F
```

```
Print newline, word, and byte counts for each FILE,  
and a total line if more than one FILE is specified.
```

```
With no FILE, or when FILE is -, read standard  
input. A word is a non-zero-length sequence of  
characters
```

```
...
```

Some commands don't support **--help**, but...

```
% cal --help
```

```
cal: invalid option -- '-'
```

```
Usage: cal [general options] [-h jy] [[month] year]
```

```
cal [general options] [-hj] [-m month] [year]
```

```
...
```

What are "commands"?

Many things can be run as a command:

- Machine-code executables, like a compiled and linked C program
- Shell scripts, functions, builtins, and aliases
- A program source file with a "shebang" line

The **type** command can be used to see what a command really is.

```
% type date
```

```
date is /bin/date
```

```
% type gcc
```

```
gcc is aliased to `c99 -Wall -g`
```

```
% type type
```

```
type is a shell builtin
```

For a shell builtin, don't use **man**; use **help**:

```
% help type
```

I/O Redirection with **bash**

I/O Redirection

There are several possible destinations for the output of a command:

- The screen
- A file
- Another command
- A hardware device
- A command on another machine
- and more!

Similarly, input may come from a variety of sources in addition to the keyboard.

UNIX has a notion of standard input and standard output.

It is common for programs to read from standard input and/or write to standard output.

By default, when the shell starts a program, standard input is associated with the keyboard, and standard output is associated with the screen.

Standard input and standard output are sometimes called *streams* or *I/O streams*.

I/O Redirection, continued

Here is a Java program that reads lines from standard input and writes the line count to standard output:

```
import java.io.*;
public class lc { // "LC" (in fall15/java/lc.java)
    public static void main(String args[]) throws IOException {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in)); // standard input in Java

        String line; int count = 0;
        // Read lines from standard input
        while ((line = in.readLine()) != null)
            count++;

        // Write count to standard output
        System.out.println(count);
    }
}
```

Interaction:

```
% java lc
one
two
three
^D (control-D)
3
%
```

I/O Redirection, continued

It is possible to *redirect* standard input, so that instead of reading characters from the keyboard, the data comes from a file.

Input redirection is indicated by adding `< file` to a command:

```
% java lc < lc.java
```

```
13
```

```
% wc < /etc/passwd
```

```
2903 10754 198109
```

```
% sha1sum < lc.class
```

```
6e3f76a2f16edb8e8b688a5938c72885cef93b13 -
```

```
% java lc < lcount.java
```

```
-bash: lcount.java: No such file or directory
```

What's interesting about the last example?

The error is from bash, not the JVM or lc.java.

I/O Redirection, continued

Output redirection is similar:

```
% java lc > count
just
testing
^D
```

```
% cat count
2
```

If the target file does not exist, it is created. If it exists, it is overwritten.

Speculate: What's the result of the following?

```
% java lc > /etc/passwd
-bash: /etc/passwd: Permission denied
```

I/O Redirection, continued

Both input and output can be redirected:

```
% java lc < /etc/passwd > pwlines
```

```
% cat pwlines
```

```
2903
```

What are some "what-ifs" we can try?

- Must input redirection appear before output redirection?
- Is whitespace needed around < and >?
- What happens if we specify two input or output redirections?
- What does `java lc < /etc/passwd > java do`?
- What does `java lc > lc.java do`?

I/O Redirection, continued

Consider this claim:

"The shell completely consumes any text used to specify redirections. For example, given `wc < lc.java`, the `wc` program does not see the `< or lc.java`."

How could we test that claim using only what we've seen thus far?

My "proof":

```
% java args a b c < lc.java > out
```

```
% cat out
```

```
| a |
```

```
| b |
```

```
| c |
```

Only `a`, `b`, and `c` ended up in the string array passed to `main`. There's no trace of `< lc.java or > out`.

I/O Redirection, continued

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

```
% wc Hello.java  
6 14 128 Hello.java
```

```
% wc < Hello.java  
6 14 128
```

What's a difference in output between the two?

Why is there a difference?

If you have an idea about why there's a difference, post it on Piazza!

I/O Redirection, continued

Consider this:

```
% wc Hello.java < args.java  
      5      14      127 Hello.java
```

Why isn't `args.java` processed, too?

Challenge: Try writing `x.java` that behaves as follows:

```
% java x a b c      # Prints a, b, c
```

```
% java x < y        # Prints the contents of the file y
```

```
% java x a b c < y # Prints a, b, c, and then contents of file y
```

```
% java x            # Prints lines read on standard input
```

I/O Redirection, continued

There is a third standard I/O stream: *standard error*.

By convention, programs send "normal" output to standard output, and "exceptional" output to standard error.

```
% cal 2016 1 > out
```

```
cal: 2016 is neither a month number (1..12) nor  
a name
```

```
% cat out
```

Standard output and error output can be combined with the '>&' redirection operator:

```
% cal 2016 1 >& out
```

```
% cat out
```

```
cal: 2016 is neither a month number (1..12) nor  
a name
```

Java's `System.err` is associated with standard error. (Try it!)

I/O Redirection, continued

A great benefit of I/O redirection is that a program doesn't need to include any file-handling code. A program can be written in terms of reading/writing standard input/output; opening files (and handling potential failures) is done by the shell.

Consider the additional code that would be required for an alternative interface for the line counter:

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

Problem: Write it! Don't forget to handle errors, too!

Contrast: Once upon a time, users of DEC's VMS operating system did output redirection like this,

```
% assign/user sys$output out  
% run program
```

which has a lot of "ceremony" compared to the UNIX equivalent:

```
% program > out
```

Sidebar: standard input and output in other languages

Here's a Ruby line counter, fall15/misc/lc.rb:

```
count = 0
while line = gets
  count += 1
end
puts count
```

Execution:

```
% ruby lc.rb < /usr/share/dict/words
99171
```

Oops! line is not needed!

fall15/misc/lc.py is a Python 3 version:

```
import sys
count = 0
while True:
    line = sys.stdin.readline()
    if len(line) == 0:
        break
    count += 1
print(count)
```

Execution:

```
% py3 lc.py < lc.rb
5
```


Pipes

Pipes

A *pipe* is an IPC (interprocess communication) mechanism supported by the UNIX kernel.

A pipe connects two processes such that data written into the pipe by the sending process can be read by the receiving process.

Let's "pipe" the output of `who` into `wc -l`, to see how many login sessions are active on `lectura`. The "or" bar is used to designate a pipe.

```
% who  
dmr pts/3 2015-08-23 01:08 (ox2.cs.arizona.edu)  
ken pts/4 2015-08-24 09:54 (imy.cs.arizona.edu)  
...lots more...
```

```
% who | wc -l  
98
```

A series of commands connected with pipes is a "pipeline".

Pipes, continued

At hand, a "pipeline":

```
% who | wc -l  
98
```

The command **who | wc -l** pipes the standard output of **who** into the standard input of **wc**. **who** and **wc** run simultaneously.

Data always flows from left to right in a pipeline.

How could I find out how many login sessions I've got active?

```
% who | fgrep whm  
whm pts/18 2015-08-17 19:00 (...)  
whm pts/28 2015-08-17 12:17 (...)
```

```
% who | fgrep whm | cat -n  
1 whm pts/18 2015-08-17 19:00 (...)  
2 whm pts/28 2015-08-17 12:17 (...)
```

Pipes, continued

Q: How many JavaScript files are in the Java archive cloudcoderApp.jar?

jar(1) output looks like this:

```
% jar tf cloudcoderApp.jar | head -15
log4j.properties
org/cloudcoder/webserver/CloudCoderDaemon$1.class
org/cloudcoder/webserver/CloudCoderDaemon.class
...
war/cloudcoder/ace/snippets/abap.js
war/cloudcoder/ace/snippets/actionscript.js
...
```

Nearly-correct solution:

```
% jar tf cloudcoderApp.jar | fgrep .js | wc -l
932
```

Fully correct, using **grep** and a *regular expression*:

```
% jar tf cloudcoderApp.jar | grep /\.js$ | wc -l
932
```

Pipes, continued

Next, we want to see the names of those JavaScript files. One approach is to let `jar tf cloudcoderApp.jar | fgrep .js` run to completion and scroll back, looking for the start of the output.

```
[900+ lines scrolled off]
war/cloudcoder/ace/worker-php.js
war/cloudcoder/ace/worker-xquery.js
war/cloudcoder/cloudcoder.devmode.js
war/cloudcoder/cloudcoder.nocache.js
%
```

Alternative: `jar tf cloudcoderApp.jar | fgrep .js | less`

```
war/WEB-INF/classes/edu/ycp/cs/dh/acegwt/public/ace/ace-compatible-noconflict.js
war/WEB-INF/classes/edu/ycp/cs/dh/acegwt/public/ace/ace-compatible-uncompressed.js
war/WEB-INF/classes/edu/ycp/cs/dh/acegwt/public/ace/ace-compatible.js
war/WEB-INF/classes/edu/ycp/cs/dh/acegwt/public/ace/ace-uncompressed.js
:
```

The colon is the `less(1)` prompt. Recall that `man` uses `less`. Type space to go forwards, `b` to go backwards. `h` (for help) shows lots more `less` commands.

Sidebar: more or less

No kidding...

% **man more**

NAME

more — file perusal filter for crt viewing

DESCRIPTION

more is a filter for paging through text one screenful at a time.

% **man less**

NAME

less - opposite of more

DESCRIPTION

Less is a program similar to more (1), but which allows backward movement in the file as well as forward movement. [And lots more!]

"more" got wired into my fingers so when **less** came along I simply did **alias more='less'** to switch to **less**. You'll see me type ... | **more** but that actually runs **less**. If I say "**more**", you should hear "**less**".



~~UNIX~~ Developers use **more** (or **less**) a lot

Browsing my **bash** history shows some examples of piping into **more** (**less**):

- `svn diff -r44:45 | more`
- `certtool d USERTrust_RSA_Certification_Authority-bad.cer | more`
- `javap -v ./target/classes/edu/arizona/cs/practice/QuickStart.class | more`
- `find | xfield -d/ -1 | sort | uniq -c | more`
- `git show master | more`
- `grep pancakes $s/*/table.out | xfield 1 3 | xfield -d/ 10 -1 | xfield 1 -1 | sort -rn -k2 | cat -n | more`
- `egrep ": xfield|FAIL" g/out | more`
- `t switched.rb | cat -A | more`
- `man -k postscript | more`

Sidebar: Where should pagination be done?

Here is a proposal for a POSIX.2 standards addition:

"All programs that produce more than one screen of output should have a **--page** option that indicates output should be paginated with functionality equivalent to **less(1)**. This will be both more convenient for users and avoid the overhead of starting **less** as a separate process."

Let's vote!

Sidebar, continued

Points to consider:

- Are there issues with the size of executables and memory usage?
- In how many different languages would the paging functionality need to be implemented?
- How many lines are on a "screen"?
- Is there much overhead in running a second process?
- What if a better pager was devised? Would we call it `--pagev2` and keep `--page` for those who like the old version?
- Should there be a `--wc` standard option, too?

Bottom line: Terrible idea!

The UNIX way: Programs have well-focused responsibilities and can be combined in various ways.

If **evenless** comes along one day, I can easily start using it.

Computing with pipes

A key element of the UNIX philosophy is to use *pipelines* to combine programs to solve a problem, rather than writing a new program.

Problem: How many unique users are on lectura?

```
v1: Get login names
% who | cut -f1 -d " "
```

ken
dmr
ken
francis
rob
walt24
dmr
rob
wnj
dmr
ken

```
v2: Sort login names
% who | cut -f1 -d" " | sort
```

dmr
dmr
dmr
francis
ken
ken
ken
rob
rob
walt24
wnj

```
v3: Get unique login names
% who | cut -f1 -d" " | sort | uniq
```

dmr
francis
ken
rob
walt24
wnj

```
v4: Get the count
% who | cut -f1 -d" " | sort |
uniq | wc -l
```

6

Pipes, continued

Write pipelines to answer these questions:

- What user has the greatest number of login sessions on `lectura`? Don't worry about ties. (Helpers: `uniq -c`, `head`, `sort -rn`)
- What words in `/usr/share/dict/words` contain all the vowels?
- Which lowercase letter occurs most often in `lc.java`? Don't worry about ties. (Helpers: `fold`, `tr -dc`)

Problem: Confirm that all processes in a pipeline are running at the same time.

Problem: How could we implement `bash`-like piping on an operating system that doesn't allow for the output of process to be connected to the input of another process? (MS-DOS did this.)

Trivial shell scripts

Shell script basics

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

Here is a two-line script that prints the number of current login sessions:

```
% cat ucount  
echo -n "Current logins: "  
who | wc -l
```

Lets make the script *executable* with `chmod`. (We'll talk about permissions soon!)

```
% chmod +x ucount
```

Depending on your search path settings for `bash` you might be able to type just **ucount** to run it:

```
% ucount  
Current logins: 44
```

If instead you see **ucount: command not found**, prefix the script name with dot-slash:

```
% ./ucount  
Current logins: 44
```

We'll learn about the search path soon!

Scripts and I/O streams

Redirecting a script's standard output produces a concatenation of standard output of all the commands in the script.

For reference:

```
% cat ucount  
echo -n "Current logins: "  
who | wc -l
```

The output of `ucount` can be redirected:

```
% ucount > out  
% cat out  
Current logins: 44
```

The file `out` ends up with the output of each command in turn.

Scripts and I/O streams, continued

Programs that are run inside a script "inherit" the standard input stream of the script.

```
% cat countbytes
```

```
wc -C
```

```
% date | countbytes
```

```
29
```

Above, the standard input of `countbytes` becomes the standard input of `wc`.

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

```
% cat lc
```

```
# Note: assumes lc.class is in this directory
```

```
java lc
```

```
% cal | lc
```

```
8
```

Script parameters

Command line arguments can be passed to scripts. An argument can be referenced using `$N`, where `N` is the 1-based position of the argument. Example:

```
% cat printargs
```

```
echo The first argument is $1
```

```
echo Arg 2: \'$2\'
```

```
echo "Third arg: >$3<"
```

```
% printargs Dots " ... " and more dots
```

```
The first argument is Dots
```

```
Arg 2: ' ... '
```

```
Third arg: >and<
```

If there is no `N`th argument, `$N` expands to nothing.

```
% printargs just testing
```

```
The first argument is just
```

```
Arg 2: 'testing'
```

```
Third arg: ><
```

Experiment: Change the quotes to apostrophes in the third `echo`.

Script parameters, continued

Imagine a verbose `wc`:

```
% ./vwc lc.java
```

```
Lines: 13
```

```
Words: 39
```

```
Chars: 361
```

Problem: write it!

Solution:

```
% cat vwc
```

```
echo -n "Lines: "
```

```
wc -l < $1
```

```
echo -n "Words: "
```

```
wc -w < $1
```

```
echo -n "Chars: "
```

```
wc -c < $1
```

Editing Files

Prominent UNIX Editors

There have been five prominent and widely popular UNIX editors:

- `ed` is the original UNIX editor. It is line-oriented and terse, but elegant. `ed`, or a lookalike, is on most UNIX systems.
- `vi` ("vee-eye") was created by Bill Joy in 1976. It is screen oriented and "modal". It was an extension of Joy's `ex`, that was essentially an improved version of `ed`. `vi` is arguably the fastest plain-text editor for touch-typists.
- In 1981, James Gosling created "UNIX Emacs", a C implementation that was similar to Richard Stallman's Emacs for the PDP-10. Important difference: Gosling's version provided "Mock Lisp", not a true Lisp.
- In 1984-1985 Stallman created GNU Emacs—the first tangible result of the GNU project.
- In 1991 Bram Moolenaar released Vim (`vi` IMproved). He started with source from Stevie, a `vi` clone. Both were originally for the Amiga.

The editing landscape

wikipedia.org/wiki/Comparison_of_text_editors lists 78 text editors that run on Windows, OS X, and/or some version of UNIX. Nine are installed on lectura.

Which one(s) should you learn?

Here are some of the things I value most in an editor:

- All common editing tasks can be done from the keyboard
- Multiple files open at once, and multiple views of a file
- Keys can be rebound to suit my preferences
- Programmable with a full-featured language I know
- Runs on OS X, Windows, and Linux

What you value may differ!

I use:

- Aquamacs for day-to-day editing
- vim for quick editing and browsing
- Eclipse or IntelliJ for Java EE projects
- PowerPoint for these slides
- WordPerfect for assignments, exams, and papers.

The editing landscape, continued

Other popular editors on Windows and/or OS X:

- Sublime
- Notepad++
- BBEdit (related: Text Wrangler)
- TextMate
- nano (a pico clone)
- And there are lots of IDE-based editors

It's silly for programmers to use editors like nano and (plain old) Notepad!

- They just doesn't do much!
- They're editors for someone who doesn't want to be proficient.
- Programmers need to be proficient when editing.
- Don't let me catch you using nano or Notepad!

The end goal: your code needs to run on lectura

All programming assignments will be graded on lectura. Therefore, here is good advice: Test your code on lectura!

Here are some ways to get source files onto lectura:

- Use an editor on lectura like `emacs` or `vim` (but not `nano`!)
- Edit on your laptop and have changes automatically propagated to lectura:
 - On Windows, WinSCP's "Keep Remote Directory up to Date" works great!
 - For OS X, Drop Sync 3 from mudflatsoftware.com looks promising but I've had trouble making it work. Yummy FTP Watcher also looks promising but I haven't tried it.
- Use remote editing on your laptop to edit files that reside on lectura:
 - Sublime has remote editing packages (more on Piazza).
 - Emacs has well integrated remote editing.
 - Flow from fivedetails.com allows remote editing with your OS X editor of choice (\$4.99). Cyberduck is similar; it's donate-ware.

A good way to irritate your instructor during office hours is to use a file transfer app that you must interact with after each save in order to get your file to lectura.

Our approach for editing

If you know or want to learn a good UNIX editor like Vim or Emacs, I encourage you to do that.

If not, I encourage you to edit on your own machines and use an SFTP client like WinSCP or Cyberduck to get your code onto lectura.

If you don't have a laptop and are working on the machines in the CS or OSCR labs and don't know where to get started, come for help during office hours.

Emacs, Vim, and your suggestions

I'll say a few things about Emacs and Vim in discussion sections but if you want to learn about them on your own, here are some resources:

- My Emacs slides straight from 2005 are here: [fall15/emacs.pdf](#)
- Emacs on lectura is **emacs**. The initial screen shows that **ctrl-h**, then **t** starts a tutorial. Exit Emacs with **ctrl-x** then **ctrl-c**.
- Vim on lectura is **vim**. The **vimtutor** shell command starts a tutorial. Exit the tutorial with **ZZ** or **:q!** .
- I learned `vi` from <http://docs.freebsd.org/44doc/usd/12.vi/paper.pdf>

There are lots of tutorials for Emacs, Vim, and other editors on the net. If you find one you like, recommend it on Piazza.

If there's an editor you really like, tell us about it on Piazza, but please no long debates about editors on Piazza—find some other place for that.

Potential for trouble

Using WinSCP and other clients create some possibility for trouble.

Example:

1. You create **x.java** on your home desktop and WinSCP syncs it to lectura.
2. On campus you edit **x.java** on lectura.
3. Back home in the evening you start up WinSCP and it overwrites your changes to **x.java** on lectura with an old copy from your desktop.

~~It's important to~~

I recommend being very consistent about where the "master" copies of files are. For example, if your masters are on your laptop, avoid the temptation to make "quick changes" in the copies on lectura.

Also, synchronizing tools sometimes stop working. If you make a change on your machine that seems to have no effect, use **cat** or **ls -lt** to check the file on lectura.

Low-tech backups for code

Along with understanding your tools I recommend frequent backups. Here's a one-line, low-tech backup that you can run on *lectura*:

```
% pr *.java | mail -s 352 your-netid@email.arizona.edu
```

- The **pr** command writes the content of each of your **.java** files in turn to standard output, and that output is piped into **mail**, a command line mailer. (Try **man mail**.)
- You'll get a message with the subject "352".
- **pr** generates some page headers that you'll have to hack out if you need to recover a file, but your source code will all be there.
- Experiment with it before you start counting on it.

Automatic file sync with WinSCP (for Windows users)

WinSCP installation

Hit <https://winscp.net/download/winscp575setup.exe> and watch out for all the spammy Download Now! buttons.

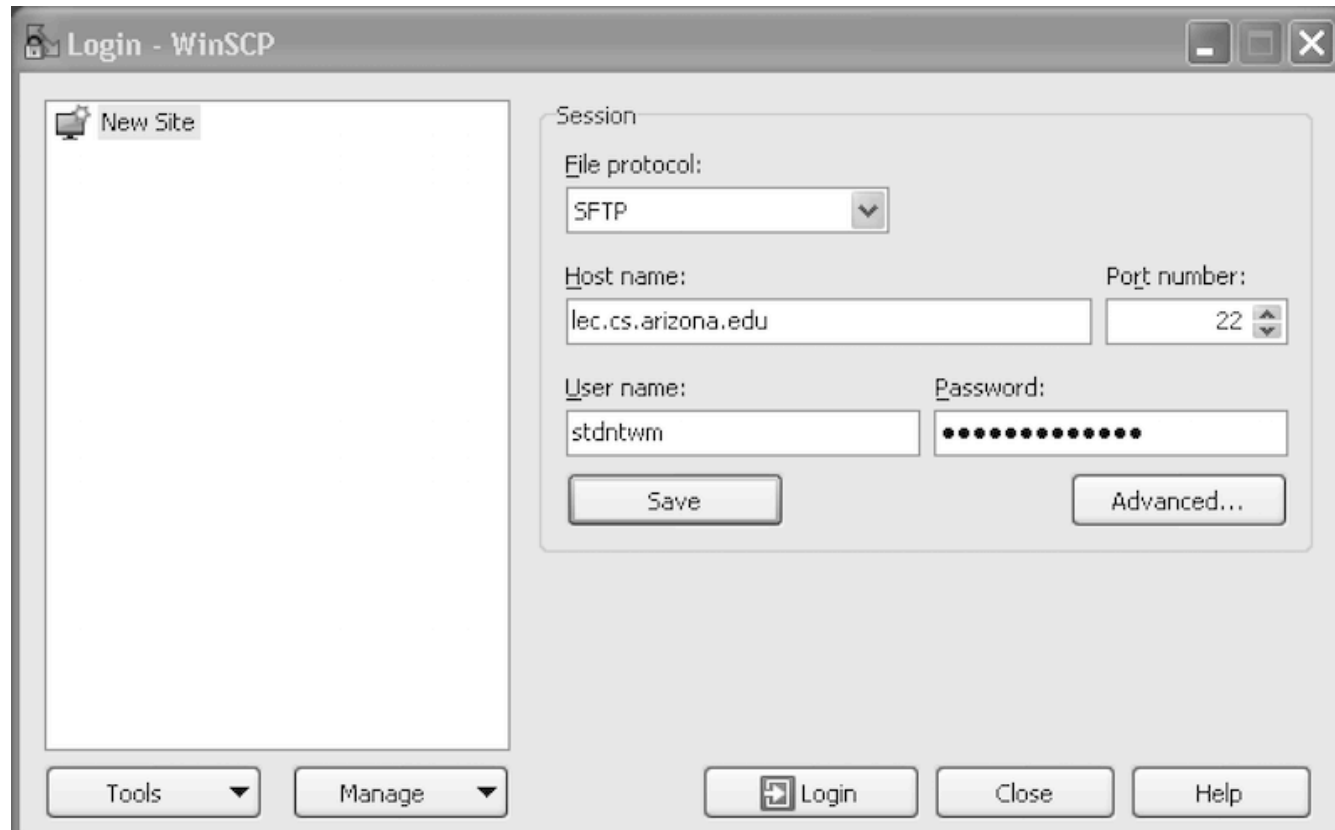
Run `winscp575setup.exe` to start an installer.

These instructions assume selecting "Typical installation" and "Commander" for the user interface style.

Start up WinSCP when the installation is complete.

Login to lectura

Use lec.cs.arizona.edu for the Host name. Leave Port number and File Protocol at their defaults. Click Login.

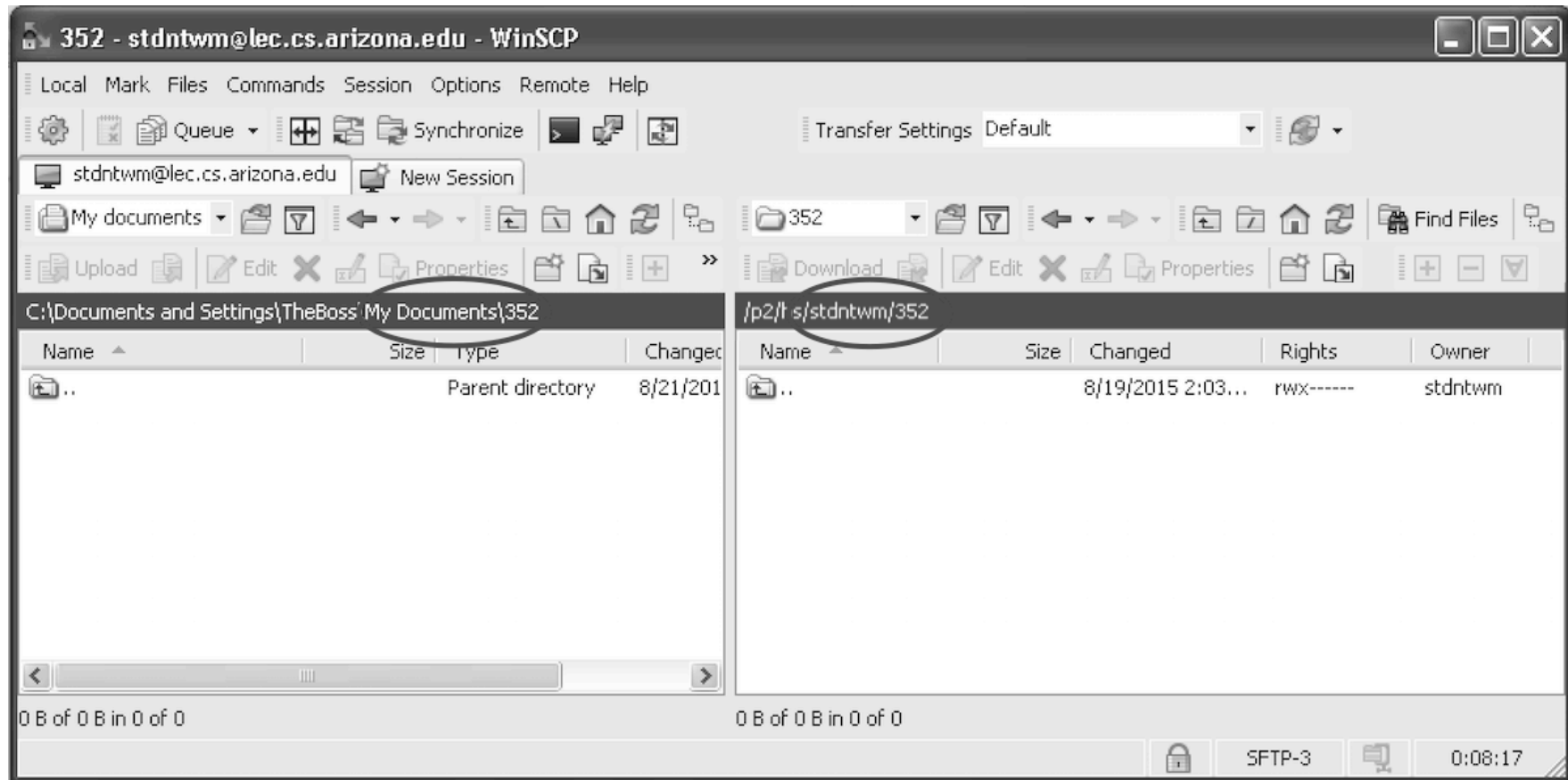


Note that you can Save connection settings. (Try it next time.)

For this demo, I'm logging into my student test account, stdntwm.

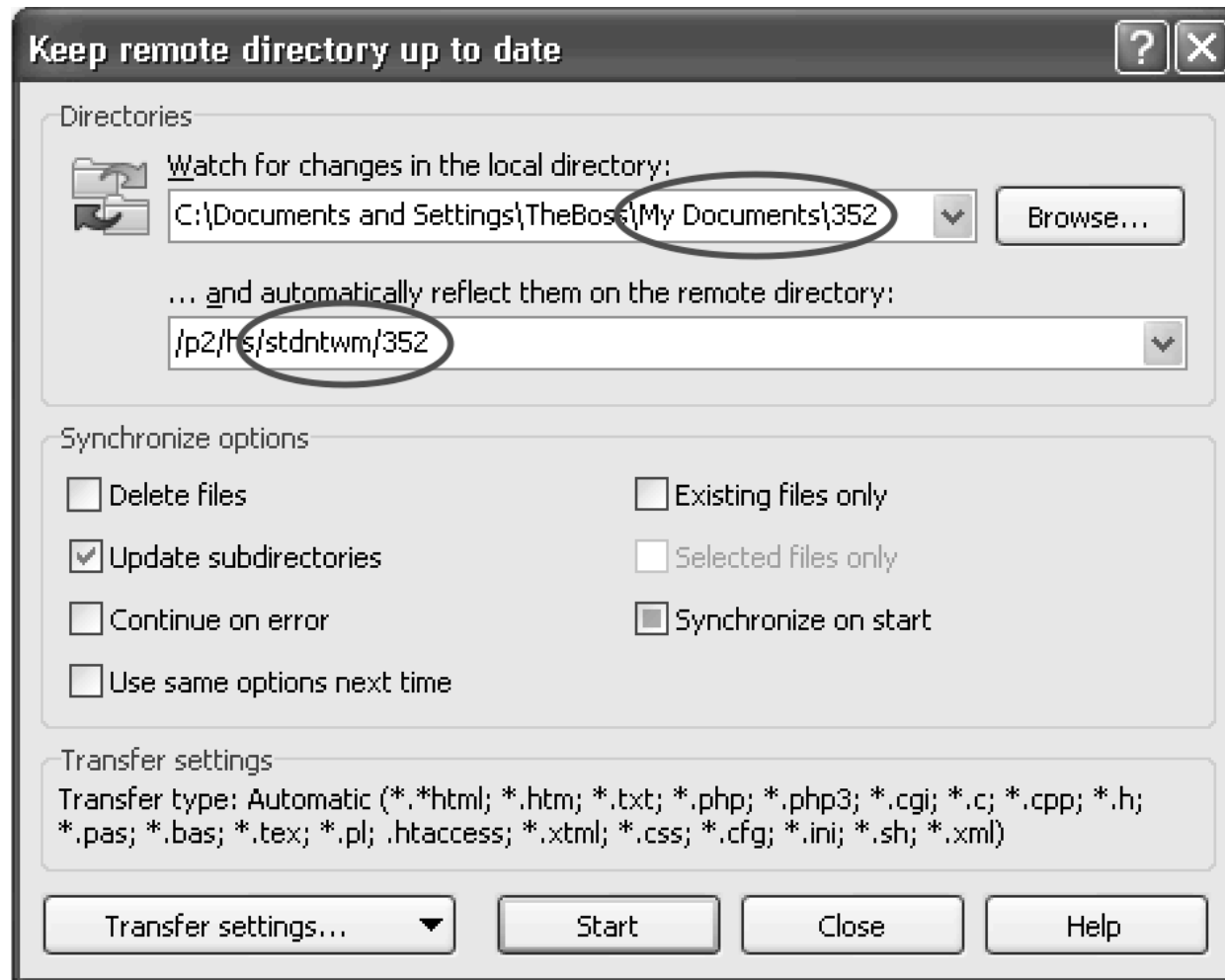
Navigate to your 352 directories on both machines

Navigate to your 352 directory (folder) in both the left panel, which shows files on your machine, and in the right panel, which shows your files on lectura. Use WinSCP to make those directories (with right-clicks) if you haven't already made them.



Synchronize and start watching

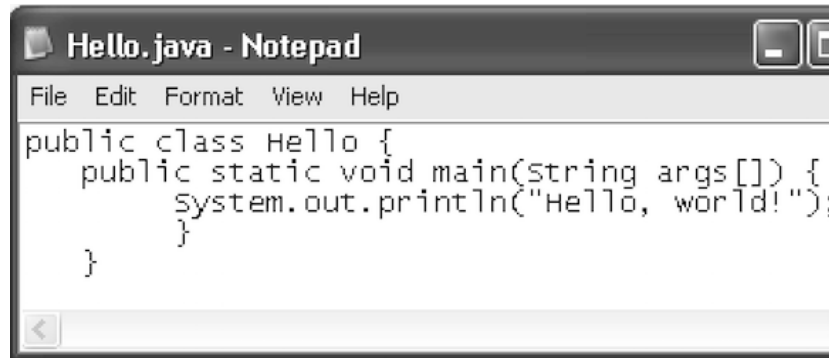
Click Commands > Keep Remote Directory up to Date... producing this:



BE SURE that the local and remote directories are correct. The Synchronize options shown are good for beginners. Click Start.

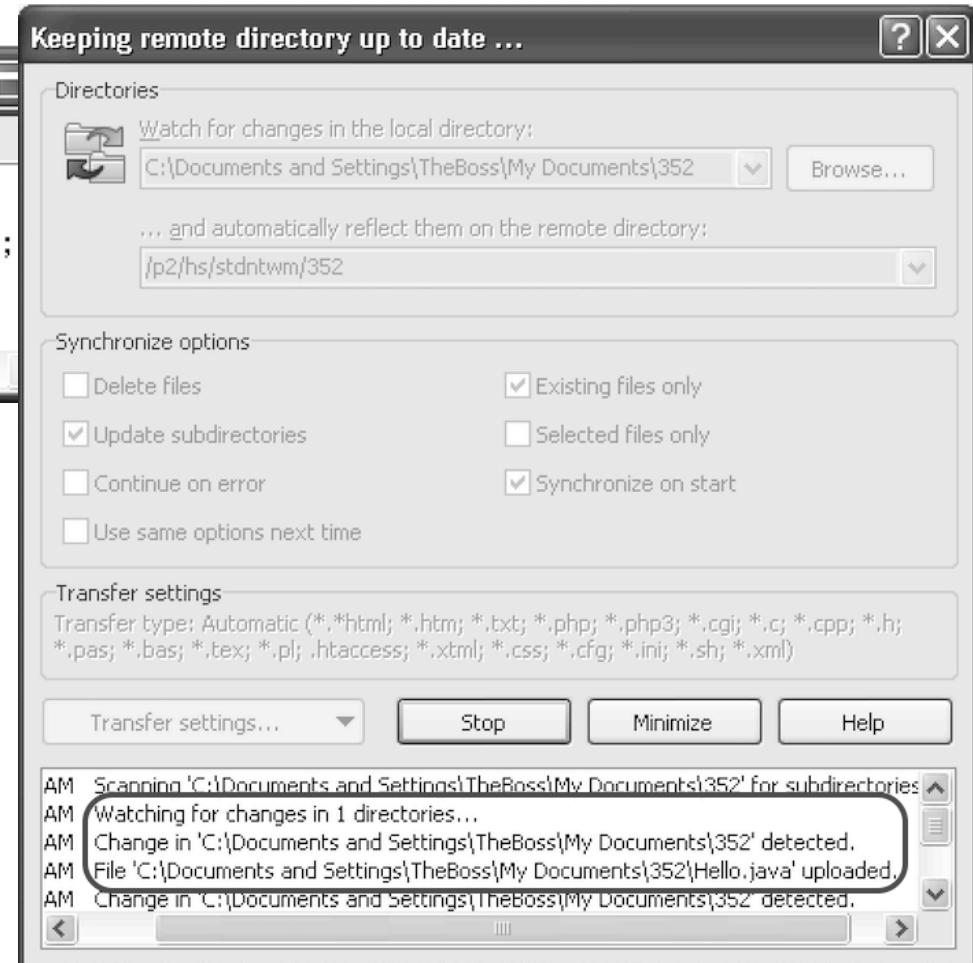
Edit, observing synchronization

Create the file Hello.java in the 352 directory on your windows machine using your favorite editor (which should not be Notepad!) When you do, you'll see activity in the "Keeping..." dialog that Start brought up.



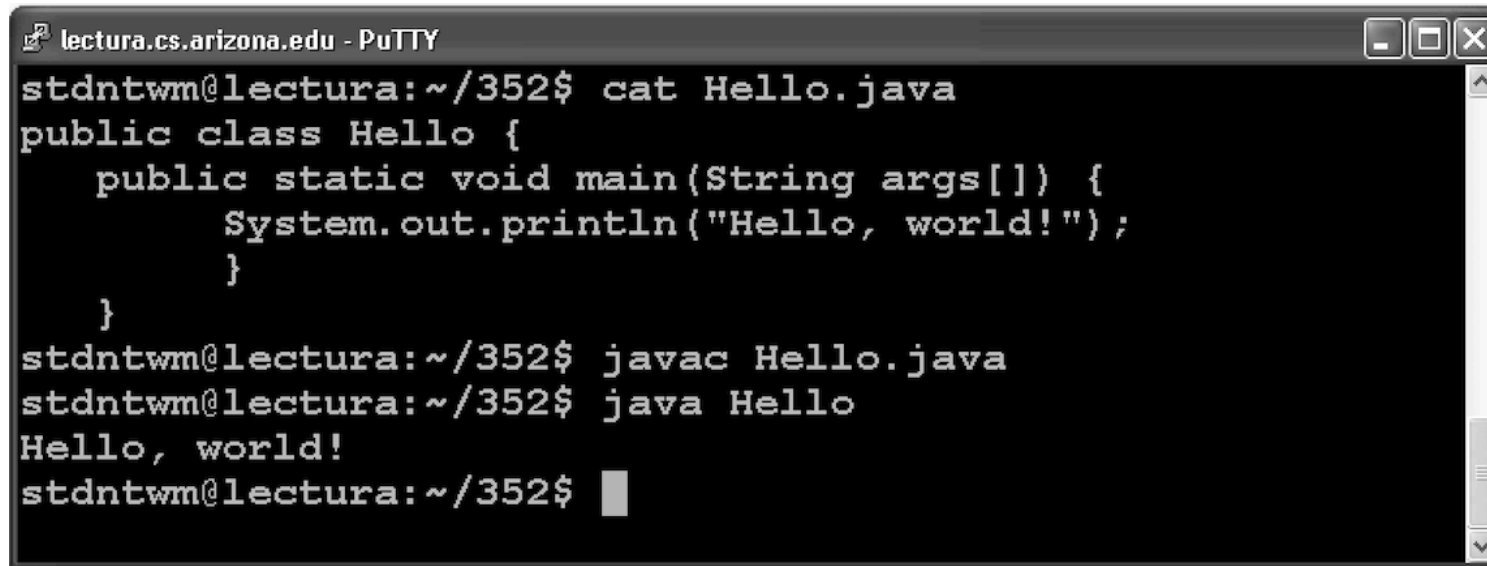
```
File Edit Format View Help
public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello, world!");
    }
}
```

Suggestion: Keep a bit of the activity pane in view in a corner of your screen, to watch for expected activity on saves.



Compile and run on lectura

Use PuTTY to connect to lectura and use the command `cd 352` to change to your 352 directory. Confirm that **Hello.java** looks good, then compile and run it.

A screenshot of a PuTTY terminal window titled "lectura.cs.arizona.edu - PuTTY". The terminal shows a user named "stdntwm" at the "lectura" machine in the directory "~/352". The user enters the command "cat Hello.java", which displays the contents of the file: a Java class named "Hello" with a "main" method that prints "Hello, world!". The user then enters "javac Hello.java" to compile the program, followed by "java Hello" to run it. The output of the program is "Hello, world!". The terminal prompt is now "stdntwm@lectura:~/352\$" with a cursor.

```
lectura.cs.arizona.edu - PuTTY
stdntwm@lectura:~/352$ cat Hello.java
public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello, world!");
    }
}
stdntwm@lectura:~/352$ javac Hello.java
stdntwm@lectura:~/352$ java Hello
Hello, world!
stdntwm@lectura:~/352$
```

Repeat the cycle a few times, editing and saving changes on your machine, observing activity in the Keeping... window, and confirming those changes are reflected on lectura.

Remote editing with Cyberduck (for Mac users)

Install and run Cyberduck

It appears that Cyberduck is \$23.99 in the App Store but if you download it from <https://cyberduck.io/>, you get a version that's donate-ware. This demo uses the latter.

Start Cyberduck and click Open Connection to get the connection dialog shown at right.

Select SFTP. For Server use lec.cs.arizona.edu.

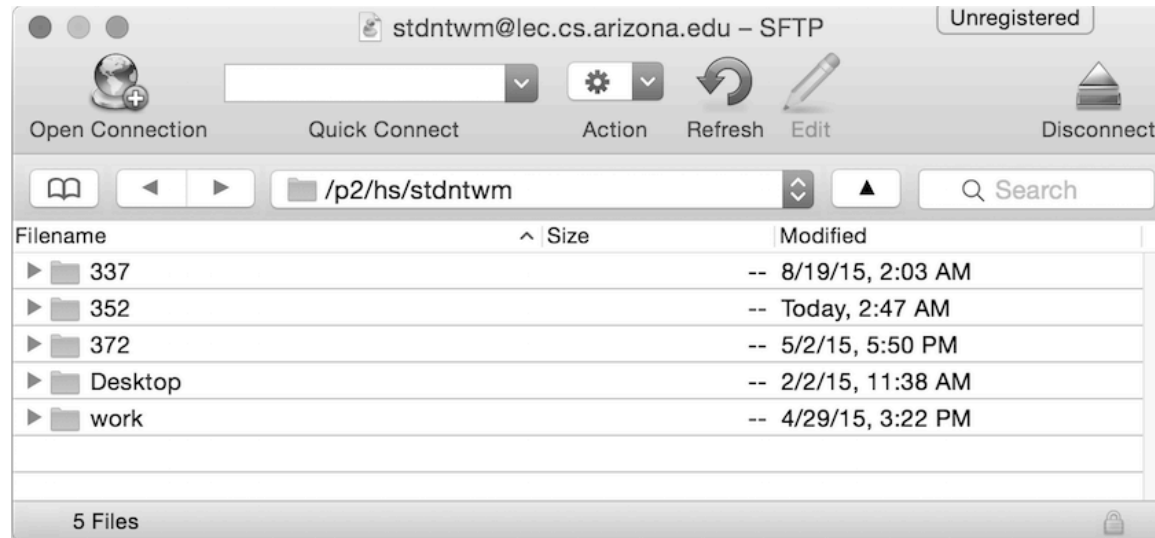
For this demo, I'm logging into my student test account, stdntwm.

Click Connect.

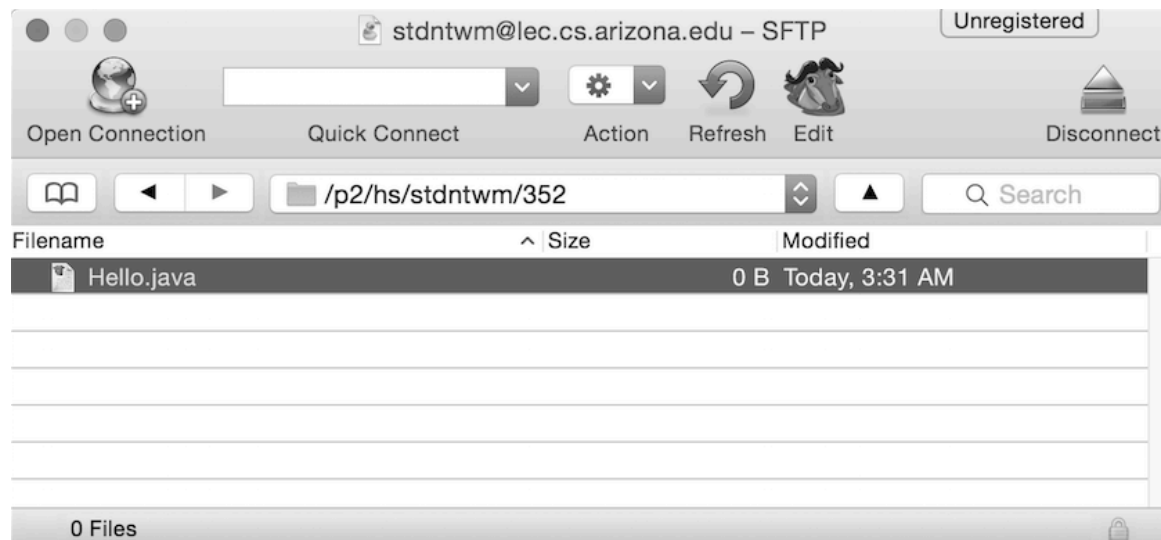


Create an empty Hello.java in 352 directory

Using a right-click, create a 352 directory (folder) if needed.

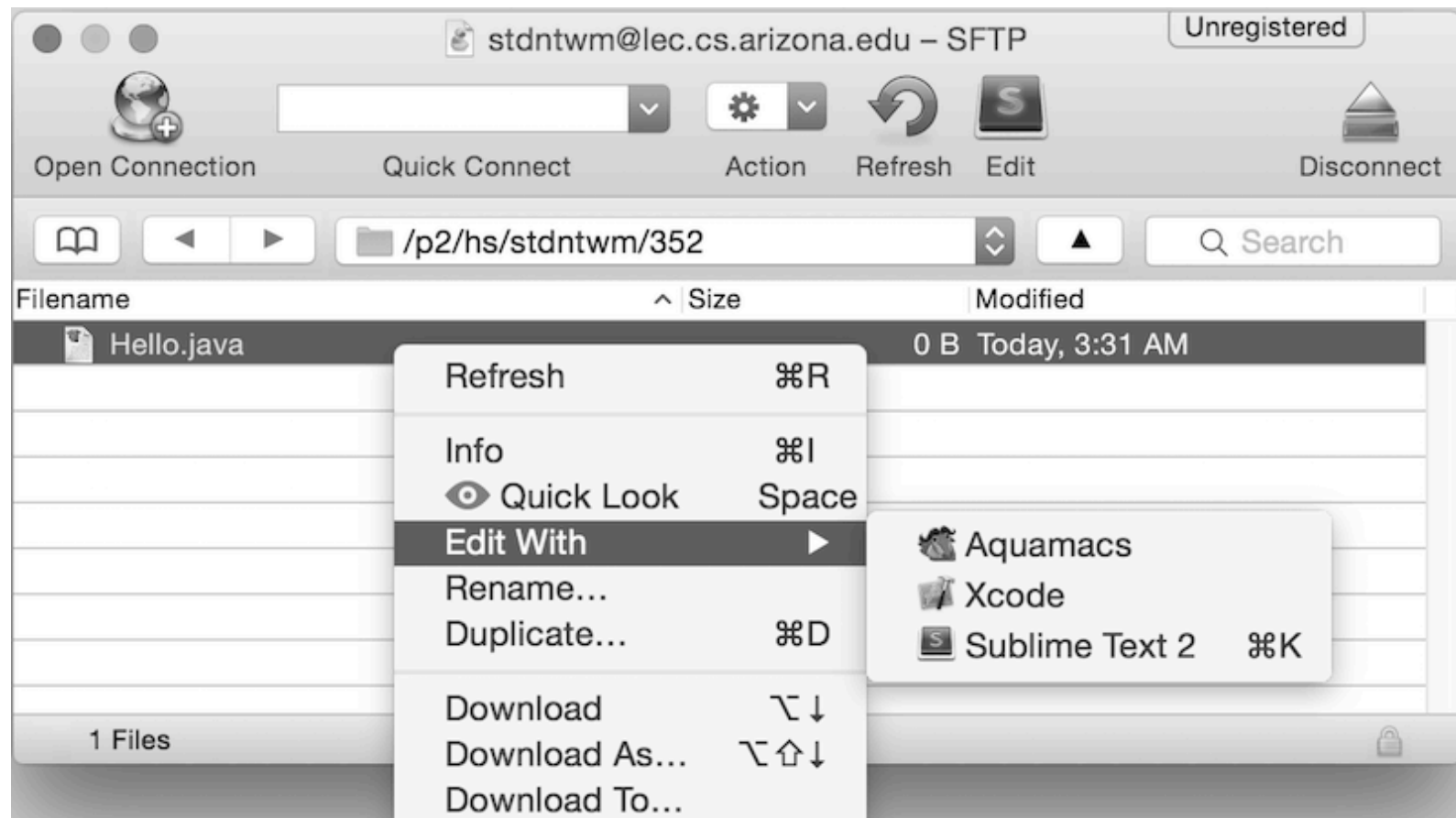


Navigate to 352 and use a right-click or cmd-F to make Hello.java, an empty file.



Open Hello.java with an editor on your Mac

With a right-click I'll indicate that I want to remotely edit the just-created and empty Hello.java on lectura using Sublime on my Mac.



Note: I needed to use Preferences and add Sublime as a default to make it appear alongside Aquamacs and Xcode.

Code up Hello with Sublime

Sublime now opens up with a empty window for Hello.java.



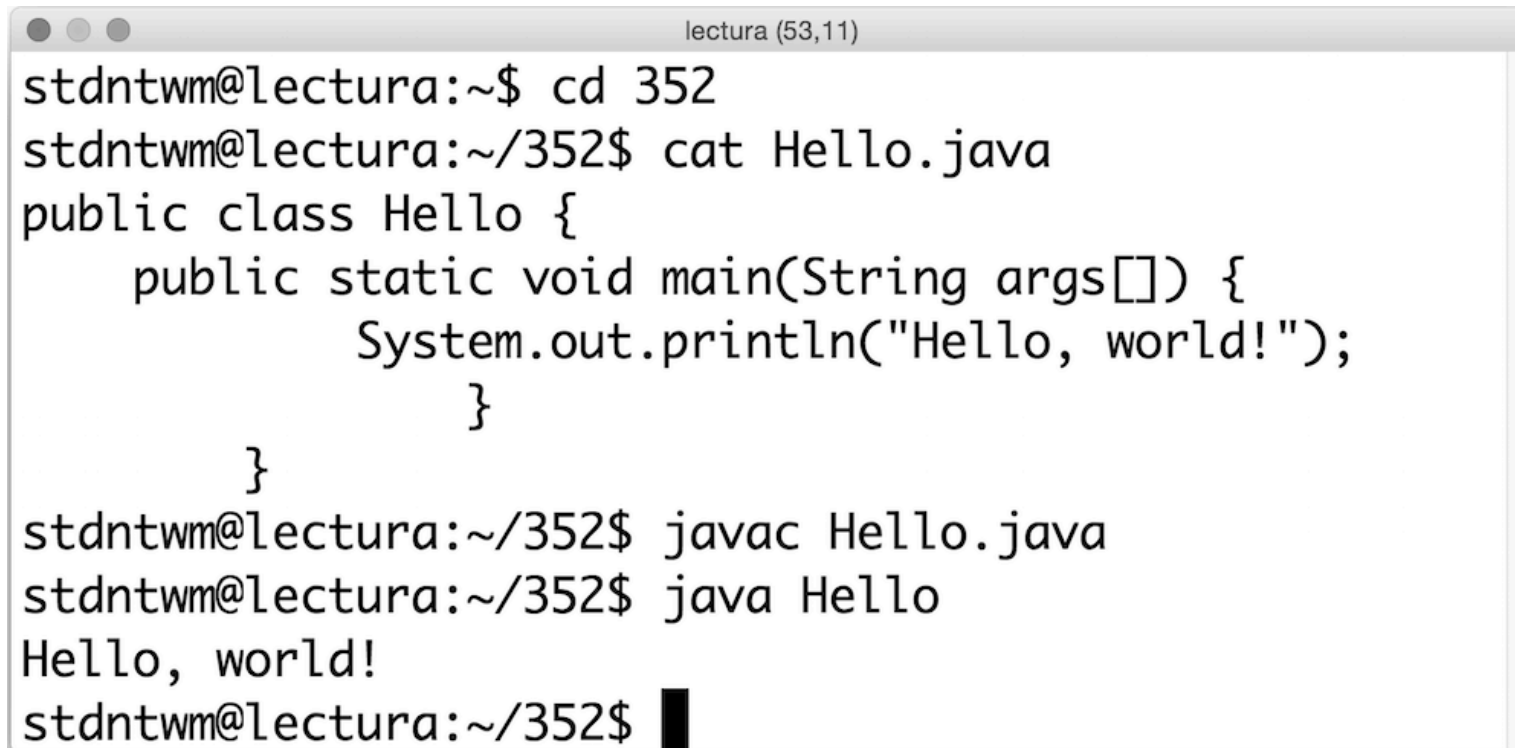
I'll write some code. Whenever I hit Save, the contents of the buffer will be written to `/home/stdntwm/352/Hello.java` on lectura. Here's the final result:



Note the long path (`/private/...`) where Sublime saved Hello.java. Whenever Cyberduck sees a change in that file, it's copied to `352/Hello.java` on lectura.

Compile and run Hello.java on lectura

After saving on the Mac, I can switch to a Terminal window where I've used ssh to login on lectura, cd to my 352 directory, and compile and run Hello.java:

A screenshot of a terminal window titled "lectura (53,11)". The terminal shows the following commands and output:

```
stdntwm@lectura:~$ cd 352
stdntwm@lectura:~/352$ cat Hello.java
public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello, world!");
    }
}
stdntwm@lectura:~/352$ javac Hello.java
stdntwm@lectura:~/352$ java Hello
Hello, world!
stdntwm@lectura:~/352$
```

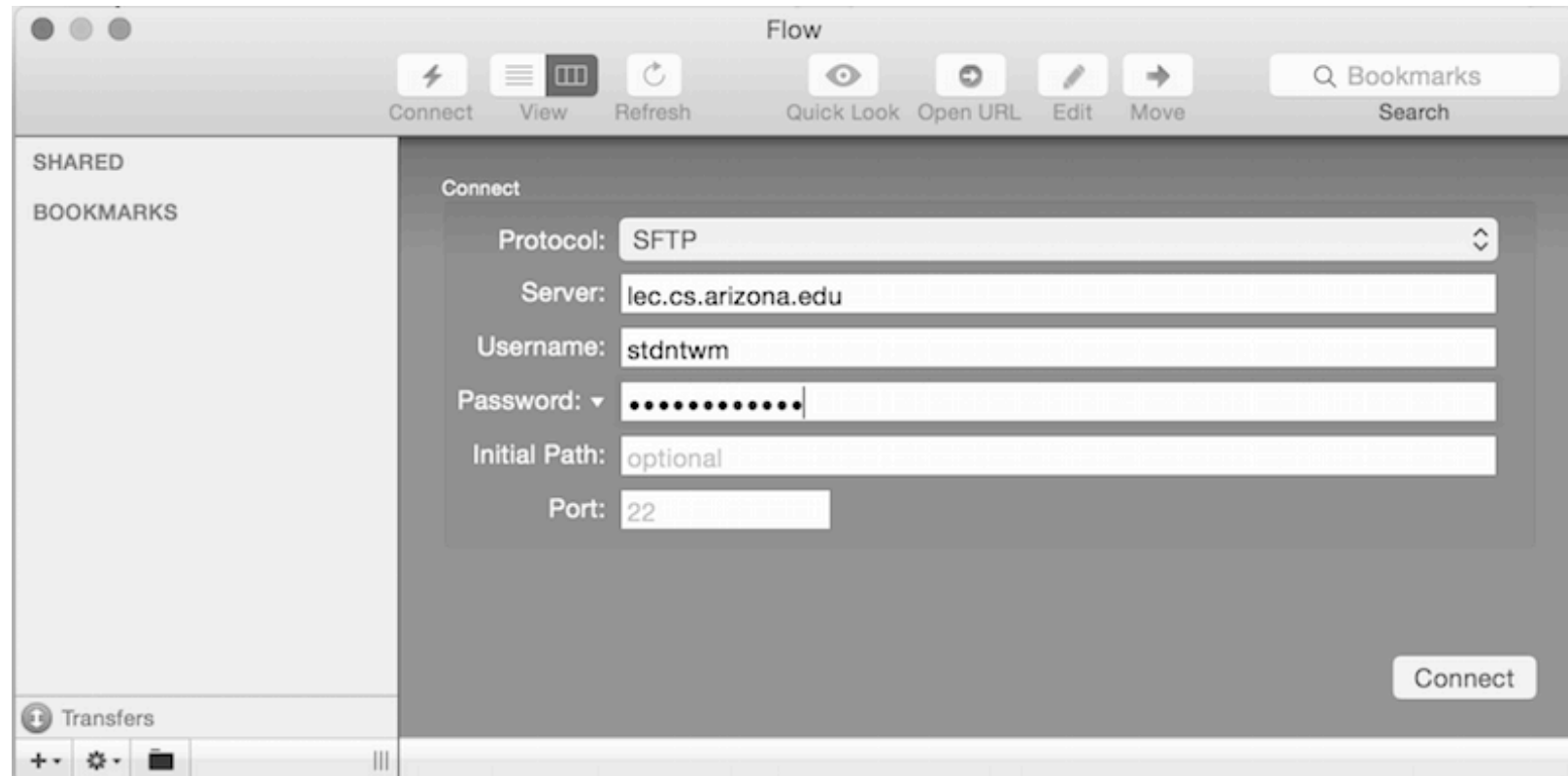
The big picture: I can use Cyberduck to open a file that resides on lectura and edit it using my favorite Mac editor.

Remote editing with Flow (for Mac users)

Remote editing with Flow

I include this section in case you don't like Cyberduck. Flow costs \$4.99.

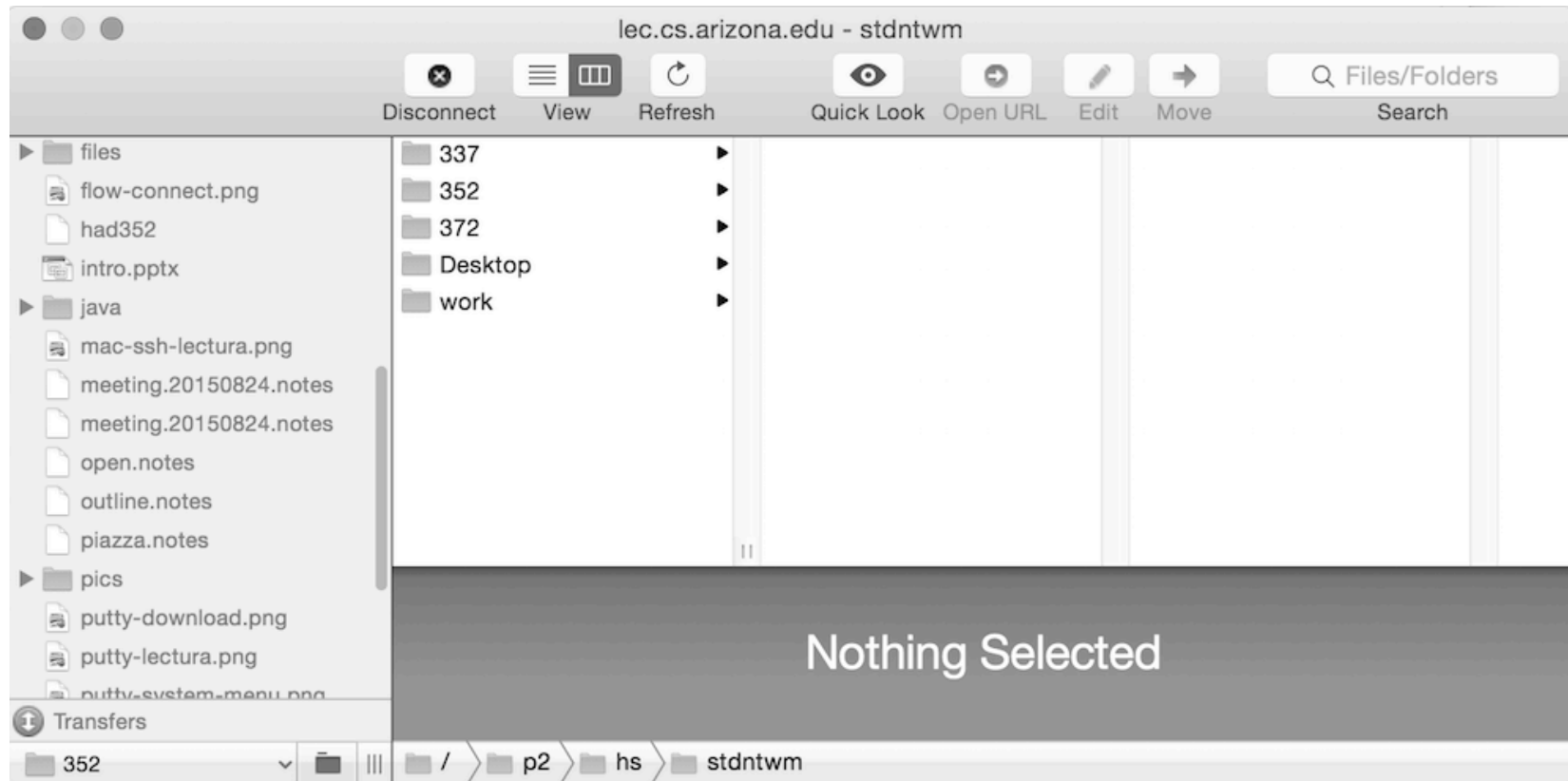
When you start Flow you'll get a connect dialog:



Leave the Protocol as SFTP and the port as 22. If you don't specify an Initial Path you'll start in your home directory. For this demo, I'm logging into my student test account, stdntwm.

Remote editing with Flow, continued

Once logged in, I see this:

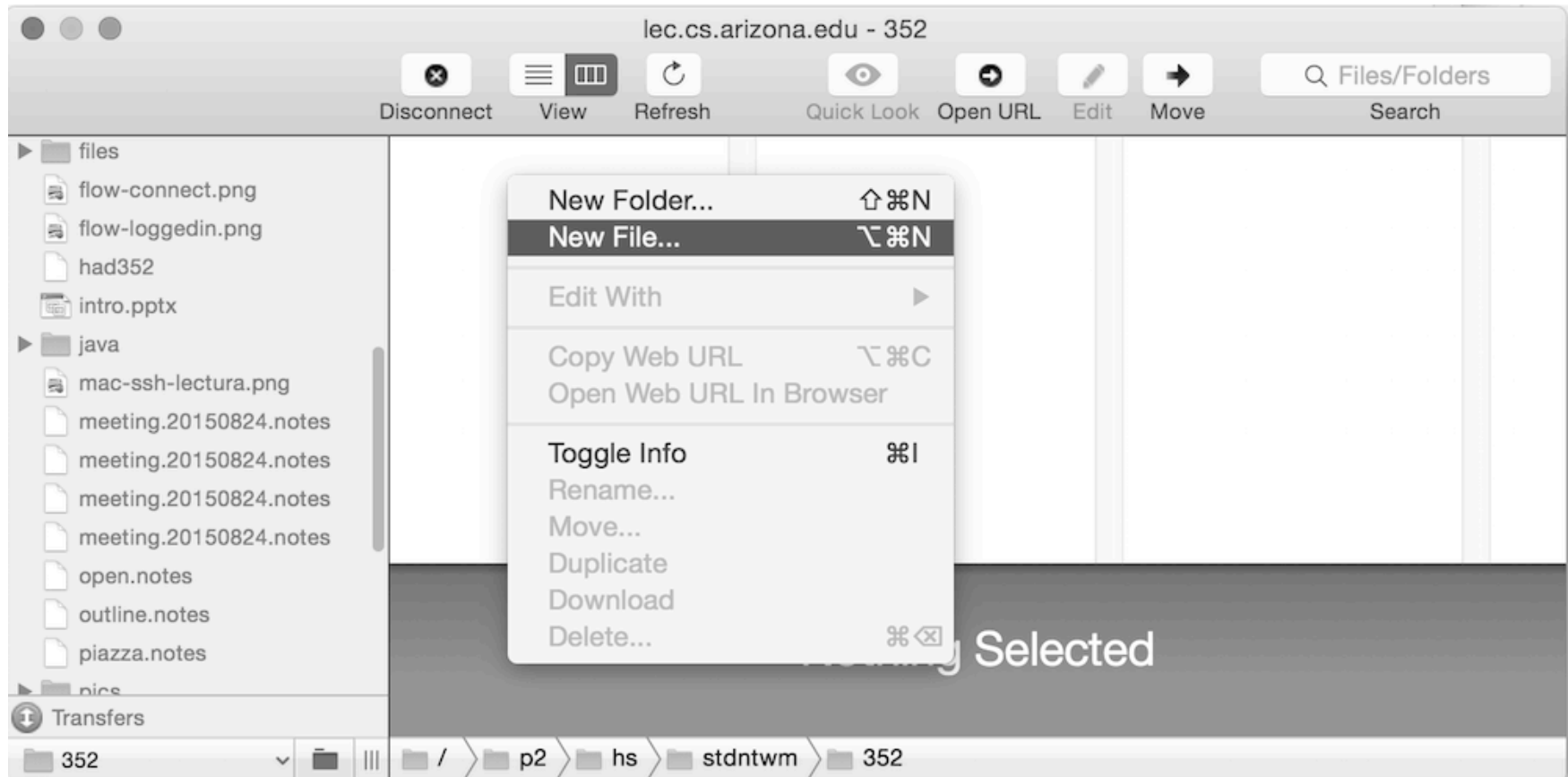


My Mac folders are on the left but I don't care about them. I simply want to create a file named Hello.java in my 352 directory on lectura, which is the second of the five directories shown on the right. I double-click 352.

Note: The *breadcrumbs* at the bottom show that my lectura home directory is /p2/hs/stdntwm, but another name for it is /home/stdntwm.

Remote editing with Flow, continued

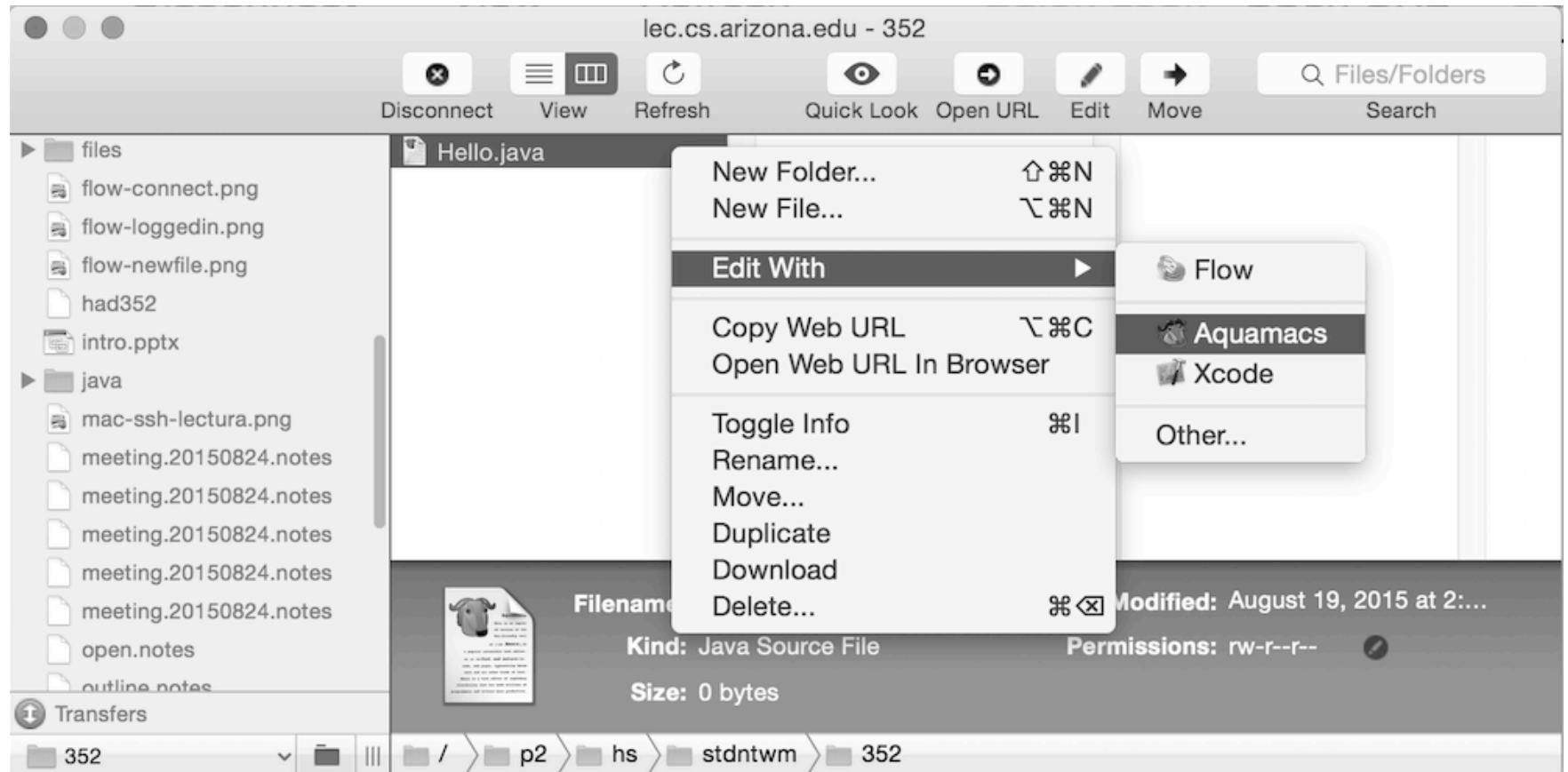
A right-click in the empty pane on the left brings up a dialog that I'll use to create a new file, Hello.java, that will reside on lectura.



Note that the breadcrumbs show I'm now in my 352 directory on lectura.

Remote editing with Flow, continued

With a right-click on Hello.java I'll indicate that I want to edit it with Aquamacs on my Mac.



Remote editing with Flow, continued

Aquamacs now opens up with a *buffer* for Hello.java.

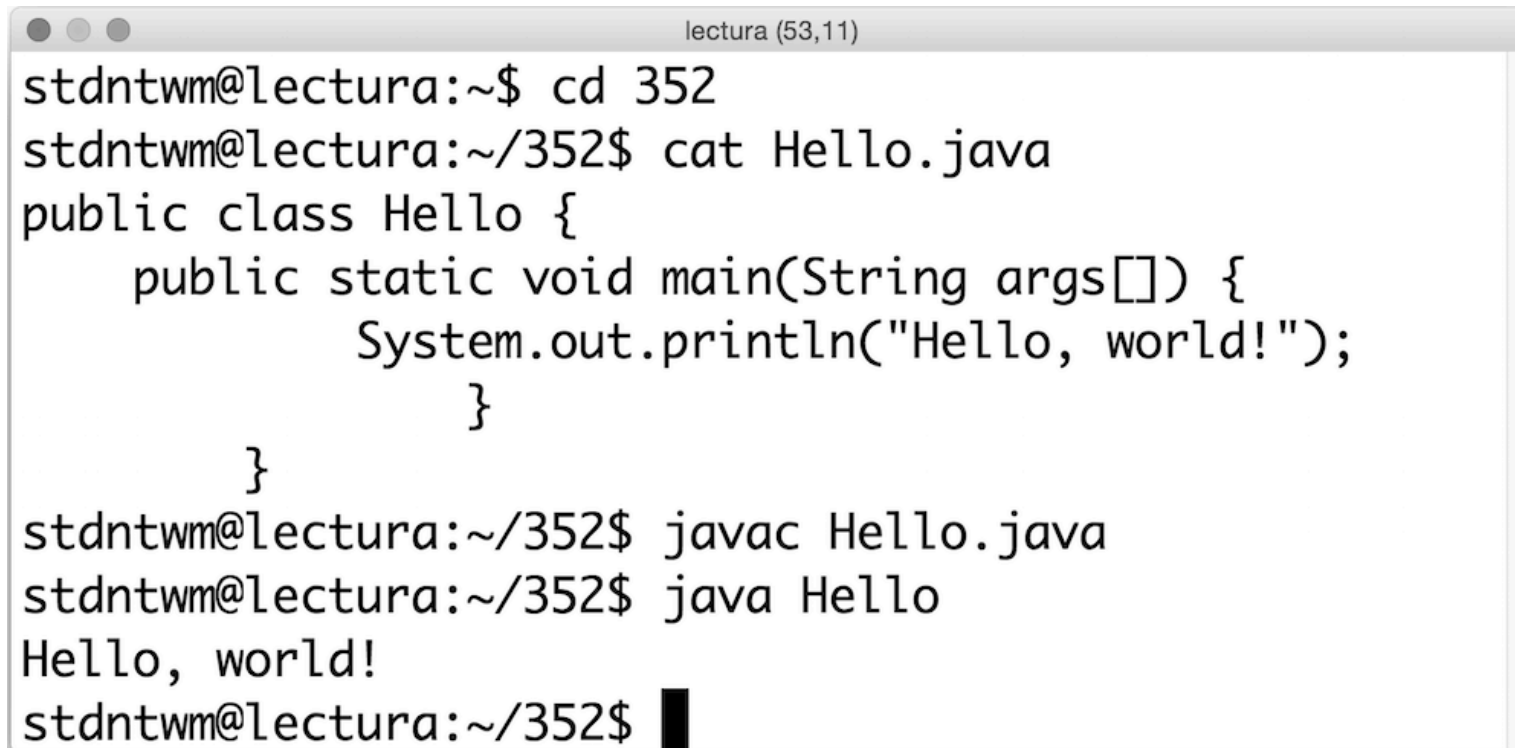


I'll write some code. Whenever I hit Save, the contents of the buffer will be written to `/home/stdntwm/352/Hello.java` on lectura. Here's the final result:



Remote editing with Flow, continued

After saving on the Mac, I can switch to a Terminal window where I've used ssh to login on lectura, cd to my 352 directory, and compile and run Hello.java:

A terminal window titled 'lectura (53,11)' showing a sequence of commands and their output. The user navigates to the directory ~/352, displays the contents of Hello.java, compiles it with javac, and runs it with java, resulting in the output 'Hello, world!'.

```
stdntwm@lectura:~$ cd 352
stdntwm@lectura:~/352$ cat Hello.java
public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello, world!");
    }
}
stdntwm@lectura:~/352$ javac Hello.java
stdntwm@lectura:~/352$ java Hello
Hello, world!
stdntwm@lectura:~/352$
```

The big picture: I can use Flow to open a file that resides on lectura and edit it using my favorite Mac editor.

Files, directories and paths

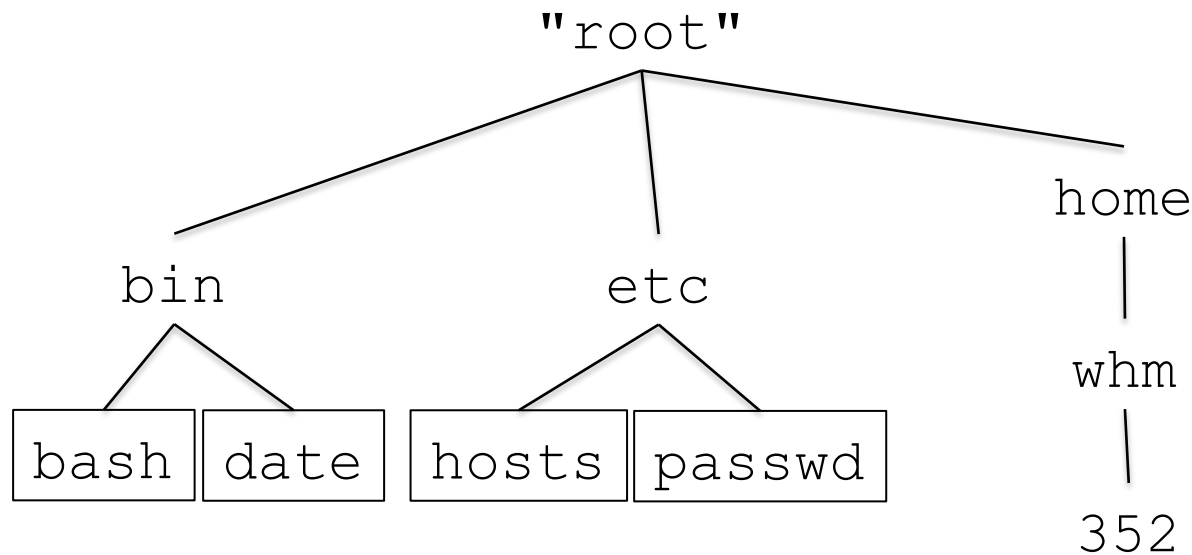
File systems

A UNIX machine stores data and provides access to it using one or more *file systems*.

- There are many different types of UNIX file systems.
- Some file systems maintain a data structure stored on a permanent medium such as a rotating disk or an SSD (solid-state drive).
- Other file systems are essentially protocols for accessing files over a network.
- The files associated with your CS account are stored on a server that uses the ZFS filesystem. (See <https://en.wikipedia.org/wiki/ZFS>)
- A ZFS filesystem can hold about 281 quadrillion files, with a total capacity of about 3×10^{23} bytes.
- Lectura accesses the files on ZFS servers using NFS (Network File System).

Directories

No matter what type of file system is being used, UNIX users see a tree of *directories*. Here's a tiny portion of the tree we see on lectura:



Directories are said to have *entries*. How many entries do each of the directories above have?

Directory entries reference "files" but a "file" can be one of many things: a *regular file*, *directory*, *special file*, *named pipe*, *socket*, or *symbolic link*. Regular files are shown boxed above; the other entries are directories.

The term "directory" is synonymous with "folder", but we'll prefer "directory".

Directory entries

Let's assume we're in an empty directory, like 352 on the previous slide.
Let's make two zero-length files using `touch`.

```
% touch one two
```

The `ls` ("LS") command lists the entries that are in a directory.

```
% ls  
one two
```

Let's now use `mkdir` to make an empty directory, `d1`.

```
% mkdir d1
```

```
% ls  
d1 one two
```

The `-F` option of `ls` causes directory names to be shown with a `/`:

```
% ls -F  
d1/ one two
```

Directory entries, continued

We can also use the `-l` ("L") option of `ls` to distinguish directories:

```
% ls -l
total 3
drwxrwxr-x 2 whm whm 2 Aug 25 19:07 d1
-rw-rw-r-- 1 whm whm 0 Aug 25 19:05 one
-rw-rw-r-- 1 whm whm 0 Aug 25 19:05 two
```

"total 3" shows that three blocks of disk space are used by these entries. *I'll sometimes not show this line, to save space.*

The "d" in the first column indicates that `d1` is a directory; the "-" shows that `one` and `two` are *regular files*.

`ls -l` shows other *metadata* (data about data) for the three entries, too:

- The `rw...` string shows *permissions* (soon...)
- The second column is the *link count* (later...)
- The next two columns (`whm whm`) show user and group ownership (later...)
- Following the ownerships is the file size, for the regular files.
- The date and time of the last modification are shown.

Sidebar: Entry names

There is a single set of rules for valid entry names for all types of entries.

Here are two simple rules for entry names:

- All ASCII characters except NUL (all bits zero) and / (slash) can be used.
- The maximum length is platform-dependent; it's 255 on lectura.

Here are some valid entry names, three per line:

```
Hello.java      a.out          core
.bashrc         a.b.c.d.      !@#$%^&*()_-=
:)              \_ \          ...
```

Here are two more entry names, one per line.

```
A collection of assorted(!) notes about UNIX
      (three blanks and two tabs!)
```

Entry names, continued

In classic UNIX, and on *lectura*, entry names are case-sensitive. For example, `hello.java` and `Hello.java` name two different files.

Some file systems are case-insensitive; some are case-configurable.

If a filename contains shell metacharacters or whitespace characters, the characters must be often be escaped when the file is specified on the command line:

```
% wc -c '[123]' Version\ 2 'This|That'
    359 [123]
    688 Version 2
    417 This|That
   1464 total
```

Due to the word-oriented nature of command-line parsing, entry names with whitespace cause a lot of headaches, especially in shell scripts. Programmers often avoid putting blanks and other problematic characters in entry names.

Current working directory

Every UNIX process (a running program) has a *current working directory*.

A very strong convention is that when a program operand specifies the name of a file, the file is assumed to be in the current working directory.

When we ran "touch one two" earlier, we were saying "Touch files one and two in the current working directory."

When we ran "mkdir d1", we were saying, "Make a directory d1 in the current working directory."

How is the current working directory used by following command?

```
% java lc < words.txt > count
```

- 1) *bash opens words.txt and count in the current working directory and passes those streams to java as standard input and standard output.*
- 2) *java reads lc.class in the current working directory and runs it.*

Note: "current working directory" is often shortened to either "current directory" or "working directory".

Current working directory, continued

Like all other processes, bash has a current working directory, too.

When bash starts a program, like touch, mkdir, or java, the process inherits bash's current working directory.

cd is a builtin command of bash:

```
% type cd  
cd is a shell builtin
```

cd changes the working directory of bash. Let's try it:

```
% ls -F  
d1/  one  two
```

```
% cd d1
```

```
% ls
```

```
%
```

Here's what happened:

- cd changed the working directory of bash to d1.
- bash ran ls, and ls inherited d1 as its c.w.d.
- Since d1 is empty, ls shows no entries.

Key question: Why must cd be a builtin?

If cd were a program, changing its c.w.d. would have no effect on the c.w.d. of bash!

Current working directory, continued

Let's do some things in my 352 directory.

```
% ls -F
```

```
d1/  one  two
```

```
% cd d1
```

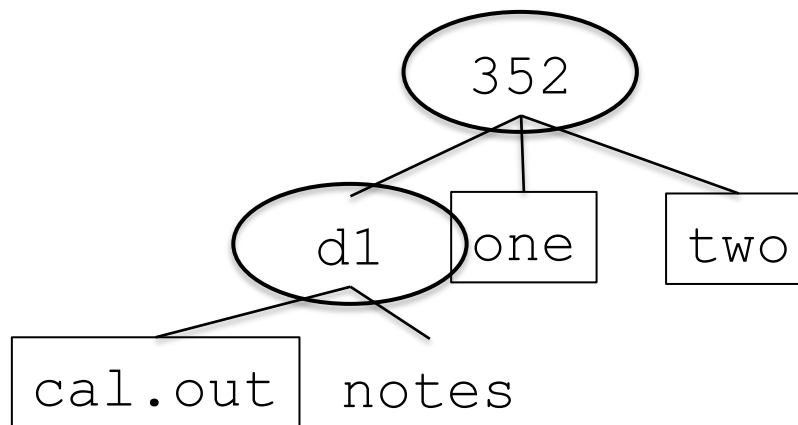
```
% ls
```

```
% cal > cal.out
```

```
% mkdir notes
```

```
% ls -F
```

```
cal.out  notes/
```



`.. ("dot dot")`

`ls -l` doesn't tell the whole truth. Let's add `-a`, which shows "hidden" entries, too. Recall that our working directory is `d1`.

```
% ls -la
total 8
drwxrwxr-x 3 whm whm 4 Aug 26 23:22 .
drwxrwxr-x 3 whm whm 5 Aug 25 19:14 ..
-rw-rw-r-- 1 whm whm 188 Aug 26 23:21 cal.out
drwxrwxr-x 2 whm whm 2 Aug 26 23:22 notes
```

By definition, "hidden" entries are entries that start with a dot.

The entry `.. ("dot dot")` is the parent directory of `d1`. We can use it to go up a level.

```
% cd ..
```

```
% ls -F
```

```
d1/ one two
```

Examining entries with `ls`

We can specify one or more directory entries as operands for `ls`.

```
% ls -l one two
```

```
-rw-rw-r-- 1 whm whm 0 Aug 25 19:05 one  
-rw-rw-r-- 1 whm whm 0 Aug 25 19:05 two
```

```
% ls -la d1
```

```
total 8  
drwxrwxr-x 3 whm whm 4 Aug 26 23:28 .  
drwxrwxr-x 3 whm whm 5 Aug 25 19:14 ..  
-rw-rw-r-- 1 whm whm 188 Aug 26 23:21 cal.out  
drwxrwxr-x 2 whm whm 2 Aug 26 23:28 notes
```

```
% cd d1
```

```
% ls -l ..
```

```
total 3  
drwxrwxr-x 3 whm whm 4 Aug 26 23:28 d1  
-rw-rw-r-- 1 whm whm 0 Aug 25 19:05 one  
-rw-rw-r-- 1 whm whm 0 Aug 25 19:05 two
```

Paths

Instead of giving `ls` an entry, we can give it a *path* to an entry. Example:

```
% ls
```

```
d1 one two
```

```
% ls -l d1/cal.out
```

```
-rw-rw-r-- 1 whm whm 188 Aug 26 23:21 d1/cal.out
```

A series of directory entries separated by slashes is called a *path*.

Almost all programs accept paths for file or directory operands.

```
% mkdir d1/notes/langs d1/notes/platforms
```

```
% cat d1/cal.out
```

```
August 2015
```

```
...
```

```
% cd d1/notes
```

```
% touch langs/java
```

Paths, continued

We can find out where we are with `pwd` (print working directory).

```
% pwd  
/home/whm/352
```

The path printed by `pwd` shows that we're in the `352` directory in the `whm` directory in the `home` directory of the "root" directory.

A directory in a directory is often called a *subdirectory*. `352` is a subdirectory of `whm`. `whm` is a subdirectory of `home`.

Let's ascend to the root. To save space we'll use a semicolon to put two commands on the same command line.

```
% cd ..; pwd  
/home/whm
```

```
% cd ..; pwd  
/home
```

```
% cd ..; pwd  
/
```

Paths, continued

Let's see what's in the root of the file system.

```
% pwd  
/
```

```
% ls -F
```

```
bin/          extensions/   local/        scratch/  
boot/         gems/        lost+found/   selinux/  
build/        home/        media/        specifications/  
build_info/   homeauto/    mnt/          srv/  
cache/        initrd.img@  opt/          sys/  
cdrom/        initrd.img.old@ p1/          tmp/  
cs/           lhome/       p2/           usr/  
dev/          lib/         proc/         var/  
doc/          lib32/       root/         vmlinuz@  
etc/          lib64/       run/          vmlinuz.old@  
etc-lect/     libnss3.so@  sbin/
```

We see mostly directories and a few symbolic links (the @'s).

Paths, continued

With "root" as our current directory, let's use `find` to show paths to all the files and directories in my 352 directory.

```
% pwd
```

```
/
```

```
% find home/whm/352
```

```
home/whm/352
```

```
home/whm/352/two
```

```
home/whm/352/d1
```

```
home/whm/352/d1/notes
```

```
home/whm/352/d1/notes/platforms
```

```
home/whm/352/d1/notes/langs
```

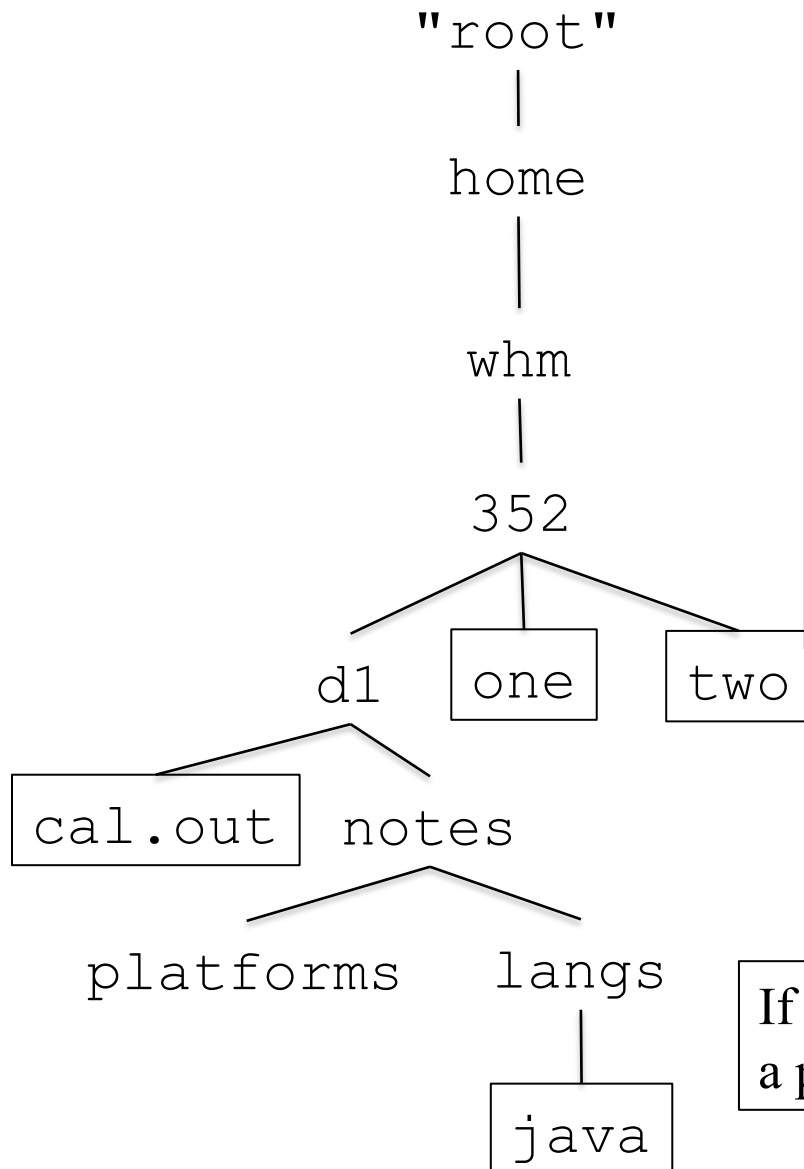
```
home/whm/352/d1/notes/langs/java
```

```
home/whm/352/d1/cal.out
```

```
home/whm/352/one
```

Paths, continued

Here's the current tree.



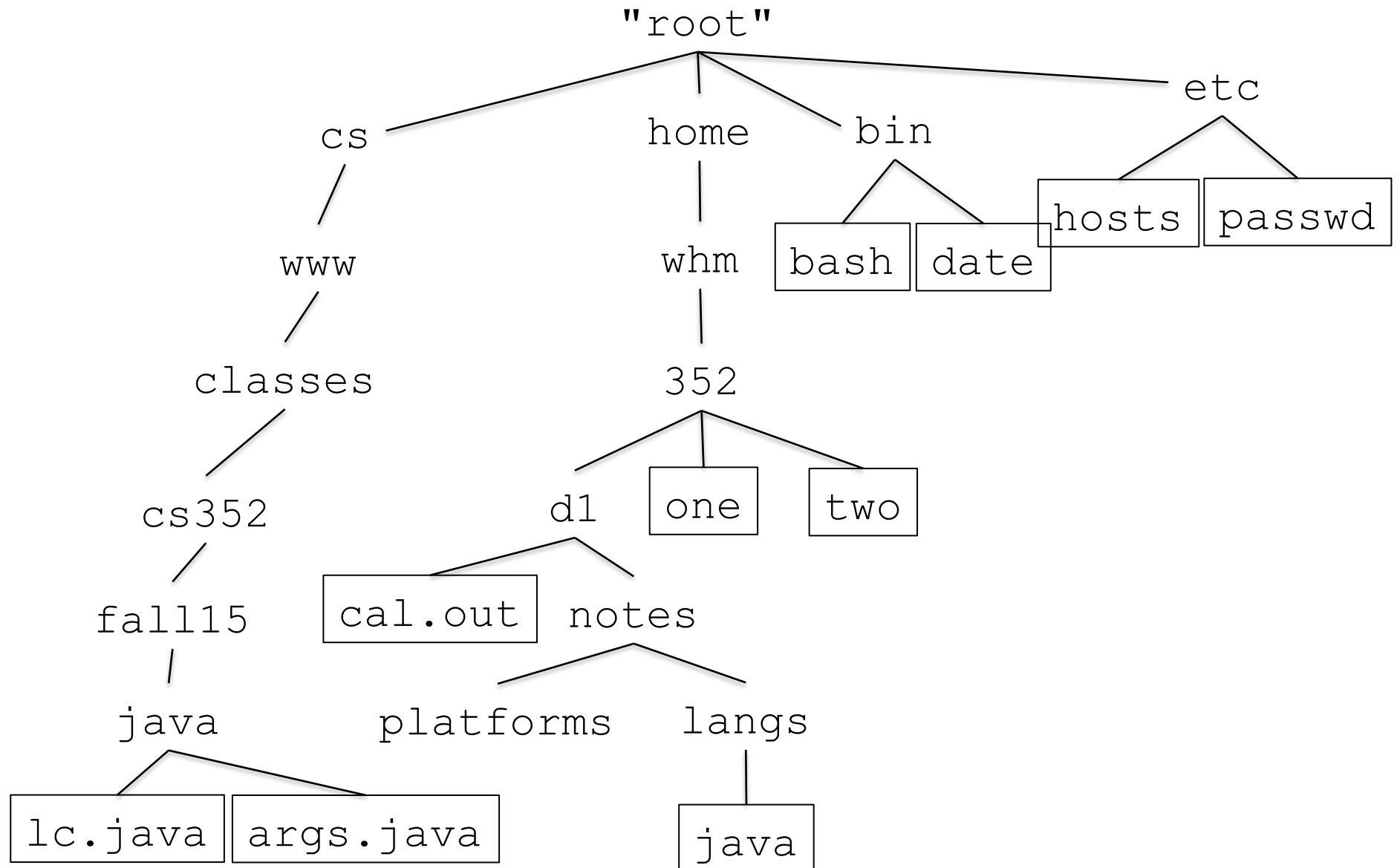
```
% find home/whm/352  
home/whm/352  
home/whm/352/two  
home/whm/352/d1  
home/whm/352/d1/notes  
home/whm/352/d1/notes/platforms  
home/whm/352/d1/notes/langs  
home/whm/352/d1/notes/langs/java  
home/whm/352/d1/cal.out  
home/whm/352/one
```

Exercise: Using the `find` output, draw the tree from scratch.

If you've had data structures...Is `find` performing a pre-order, in-order, or post-order traversal?

A bigger picture

Let's add some other files and directories referenced in these slides.



Navigation with paths

352

Let's try some problems with this tree:

If we're in `work`, how can I `cd` to ...

`352`?

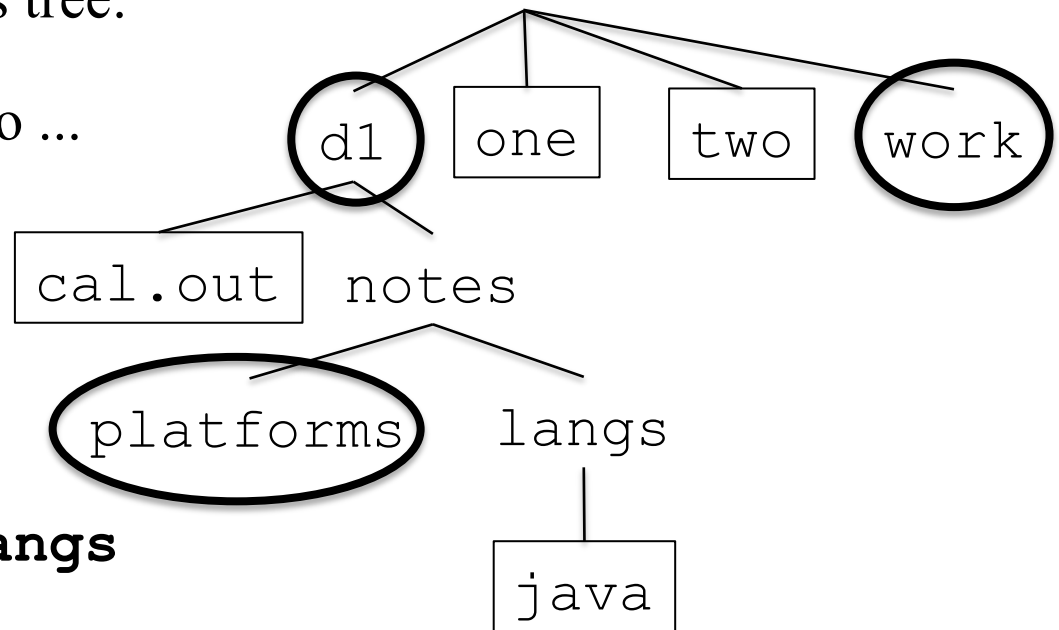
`cd ..`

`d1`?

`cd ../d1`

`langs`?

`cd ../d1/notes/langs`



If in `platforms`, how can I `cat`...

`cal.out`?

`cat ../../cal.out`

`java`?

`cat ../langs/java`

`two`?

`cat ../../../two`

If in `d1`, which of these work?

`cd ../d1`

`cat d1/cal.out`

`cat ../../two`

`cd notes/langs/java`

`cd notes/platforms/..`

`cd langs/notes`

Practice!

Exercise:

- Draw a small tree with nine directories and five files.
- Using `mkdir` and `touch`, create the tree.
Instead of `touch`, you might use `echo` to provide a little file content:

```
% echo This is x.java > x.java
```
- Write and solve seven navigation problems like on the previous slide.
- Try `find` (no arguments) and `ls -laR` at various points in the tree.
- Experiment with TAB completion to build a path piece by piece. Work through your navigation problems one directory at a time, using TAB completion to look around.

Handy: "`cd -`" goes back to the last directory you were in.

The `cp` command

The `cp` command copies files and directories. A simple case is a file to file copy:

```
% cp List.java List.java.bak
```

If `List.java.bak` already exists, it is silently overwritten.

Paths can be used with `cp`:

```
% cp ../cal.out langs/x
```

`cp` can copy whole trees:

```
% cp -r notes/langs langs.bak
```

The cp command, continued

This SYNOPSIS from the cp man page shows another way to use cp:

```
cp [OPTION]... FILE1 FILE2 ... FILEN DIRECTORY
```

[OPTION]... indicates that cp accepts some options.

The all-caps strings like FILE1 and DIRECTORY are placeholders that describe what's expected.

Speculate: What does this synopsis tell us about a possible way to use cp?
cp can copy any number of files specified on the command line into a specified directory.

Let's copy three files to ../work. They'll retain their names.

```
% cp cal.out ../one ../two ../work
```

```
% ls ../work
```

```
cal.out  one  two
```

Practical problem

Problem: I want to copy `../t-e9c3dc762.xml` into my current directory, `352/d1`.

Bad solution--just hammer it out (error prone and not DRY!):

```
cp ../t-e9c3dc762.xml t-e9c3dc762.xml
```

Here's a creative answer:

```
cp ../t-e9c3dc762.xml ../d1
```

We could use filename completion with TABs:

```
cp ../t-TAB ../t-TAB
```

producing

```
cp ../t-e9c3dc762.xml ../t-e9c3dc762.xml
```

Then use command-line editing to remove `../` from the second path:

```
cp ../t-e9c3dc762.xml t-e9c3dc762.xml
```

But there's a better way!

Practical problem, continued

Recall that along with "dot dot", `ls -la` shows "dot":

```
% ls -la
drwxrwxr-x 3 whm whm 5 Aug 28 09:20 .
drwxrwxr-x 4 whm whm 7 Aug 28 09:30 ..
-rw-rw-r-- 1 whm whm 188 Aug 26 23:21 cal.out
...more...
```

The hidden entry . ("dot") is the current directory. We can do this:

```
cp ../t-e9c3dc762.xml .
```

We've seen dot earlier, too:

```
find . -type f -mtime -2 -size +1k
cp /cs/www/classes/cs352/fall15/java/args.java .
./ucount
```

What do these commands do?

```
cd .
cd ../.
cd ../.././../---
```

Relative vs. absolute paths

Here are some examples of a *relative path*:

```
x.java  
352/ucount  
./out  
../../  
etc/passwd  
./etc/passwd
```

Here are some examples of an absolute path:

```
/home/whm/x.java  
/Users/whm/352/ucount  
/out  
/etc/..bin  
/etc/passwd  
/home  
/cs/www/classes/cs352/fall15/syllabus.pdf  
/.
```

What do the absolute paths have in common?

They start with a slash.

Relative vs. absolute paths, continued

Two simple rules:

Any path that starts with a slash is an absolute path.

If a path doesn't start with a slash, it is a relative path.

Relative or absolute?

`x.java`

`../../x`

`/home/whm/x.java`

`352/ucount`

`./etc/passwd`

`/etc/passwd`

Relative vs. absolute paths

When the UNIX kernel opens a file specified by an absolute path, it starts at the root of the file system and then works through the path one entry at a time.

If I execute this Java expression,

```
new FileReader("/home/whm/352/x")
```

the system...

- ① Looks in the root (/) for home, then...
- ② Looks in home for whm, then...
- ③ Looks in whm for 352, then...
- ④ Looks in 352 for x

How would the steps differ for the following?

```
new FileReader("home/whm/352/x")
```

The system...

- ① Looks in the current working directory for home, then...
- Steps 2-4 are the same!

Relative vs. absolute paths

What's a key benefit of absolute paths?

Absolute paths work no matter what your current directory is.

What's a key element that relative paths depend on?

The concept of a current directory.

(Without a current directory, there's nothing to be relative to!)

What are key benefits of relative paths?

Brevity:

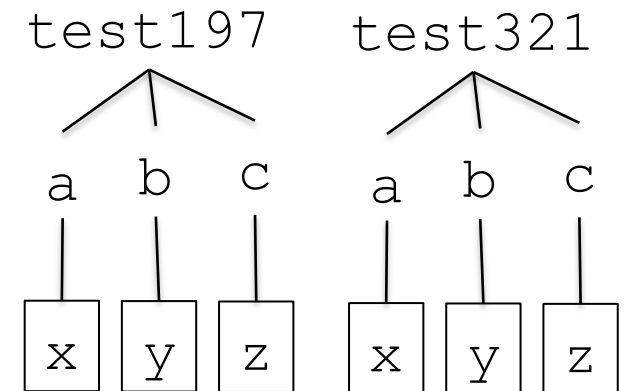
If I'm in /home/whm/352 I can say `cat java/lc.java`.

Location independence for programs using them.

Imagine many identically structured trees holding test results:

An analyzer might `cd` into each tree in turn and analyze `a/x`, `b/y`, and `c/z`.

Most of the path stuff, including `.` and `..` has worked on all Microsoft operating systems.



Your home directory

The sixth field of your `/etc/passwd` entry specifies the path to your home directory. Here's my entry:

```
% fgrep whm: /etc/passwd
whm:x:3086:8086:William H. Mitchell:/home/whm:/bin/bash
```

Whenever you login to `lectura`, your current directory is set to your home directory.

If `cd` is run with no arguments, it takes you to your home directory.

```
% pwd
/home/whm/352/a2
```

```
% cd
```

```
% pwd
/home/whm
```

Do `man -s 5 passwd` for a description of the `/etc/passwd` fields.

Tilde expansion

bash expands ~/ in a command line to the absolute path of your home directory.

```
% echo ~/352/d1  
/home/whm/352/d1
```

```
% cd ~/info
```

```
% pwd  
/home/whm/info
```

```
% cat ~/3bin/lec-mpage
```

```
...
```

Tilde expansion saves some typing but what's another benefit?

Scripts can use ~ to keep paths system-independent.

On my Mac, ~ is /Users/whm but a script using ~/lib/x works on both my Mac and lectura.

Is ~/lib/x an absolute path or a relative path?

Tilde expansion, continued

It's important to understand that tilde expansion is done by bash (evidenced by `echo`), not the operating system.

Library routines typically do not do tilde expansion. Let's try Ruby:

```
% irb
>> open("/home/whm/x")
=> #<File:/home/whm/x>
>> open("~/x")
Errno::ENOENT: No such file or directory - ~/x
```

Some programs recognize `~/` as being a shorthand for your home directory.

In Vim things like `:vi ~/x` and `:r ~/352/notes` work.

Emacs handles paths like `~/x`. In Emacs Lisp, `(find-file "~/x")` works.

Some (all?) file selection dialogs on OS X recognize `~/`. (Do `File>Open` in some app and type `~` or `/`, and you'll get a "Go to the folder:" slide-down.)

Try this: (use `~whm` as shown, not your login name)

```
% java args ~ ~whm /x/~y /x/~ ~mysql
```

Symbolic links

Motivation:

Test files for a2 are in `/cs/www/classes/cs352/fall15/a2`, but that's a long path to type!

The a2 write-up says to do this,

```
% cd ~/352/a2
```

```
% ln -s /cs/www/classes/cs352/fall15/a2 .
```

then take a look at what `ls` shows: (output is split across lines)

```
% ls -l a2
```

```
lrwxrwxrwx 1 whm whm 31 Aug 29 21:42 a2 ->  
/cs/www/classes/cs352/fall15/a2
```

That lowercase "L" at the start of the line indicates that `a2` is a *symbolic link*, often shortened to "*symlink*".

The `a2 -> /cs/.../fall15/a2` indicates that `a2` references (or "points to") that entry.

The symlink, an entry in the current directory, was named `a2` because the destination, the second operand of `ln -s`, was dot, the current directory.

Symbolic links, continued

Let's temporarily set the bash prompt to show the current working directory and then see what's in a2.

```
% PS1="\n\w % "
```

```
~/352/a2 % ls a2
```

```
amj-hints  days.2  isleap-hints  timeline.txt  
args.out   delivs  lengths.1     tomorrow-hints
```

```
...
```

The a2 symlink creates the illusion that 352/a2 has an a2 subdirectory but in fact we're looking at /cs/www/classes/cs352/fall15/a2!

```
~/352/a2 % ls /cs/www/classes/cs352/fall15/a2
```

```
amj-hints  days.2  isleap-hints  timeline.txt  
args.out   delivs  lengths.1     tomorrow-hints
```

```
...
```

It is very important to understand this:

```
~/352/a2      is /home/whm/352/a2
```

```
~/352/a2/a2  is /cs/www/classes/cs352/fall15/a2
```

Symbolic links, continued

At hand:

```
~/352/a2 % ls -l a2  
lrwxrwxrwx 1 whm whm 31 Aug 29 21:42 a2 ->  
/cs/www/classes/cs352/fall15/a2
```

Files in `/cs/www/classes/cs352/fall15/a2` can now be referenced concisely:

```
~/352/a2 % cat a2/rev.1  
one  
two  
three
```

```
~/352/a2 % java rev < a2/rev.1  
eno  
owt  
eerht
```

The tester will assume the presence of an `a2` symlink!

Symbolic links, continued

A symbolic link can reference any type of "file", including a regular file.

```
~/352/a2 % ln -s a2/lengths.1 x
```

```
~/352/a2 % ls -l x
```

```
lrwxrwxrwx 1 whm whm 12 Sep  1 17:34 x -> a2/lengths.1
```

```
~/352/a2 % cat x
```

```
just
```

```
a
```

```
test
```

```
here
```

```
~/352/a2 % java lengths < x
```

```
4
```

```
1
```

```
4
```

```
0
```

```
4
```

Symbolic links, continued

Key point:

Symbolic links are handled by the operating system.

Benefit:

A program doesn't have to do anything special to follow a symlink to its destination.

In 352/a2, `new FileReader("a2/days.1")` works. The `FileReader` constructor has no clue that `a2` is a symlink!

Sidebar: Windows shortcuts

I often describe a symlink as a "Windows shortcut done right."

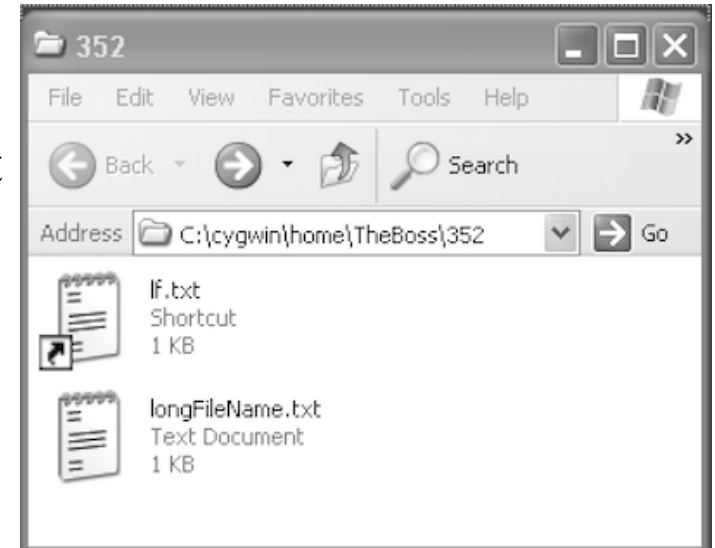
I've made a Windows shortcut named `lf.txt` that references `longFileName.txt`.

I can open either with Explorer but watch what type, the Windows analog of `cat (1)`, does:

```
C:>type longFileName.txt
Tue, Sep 01, 2015  5:50:25 PM
```

```
C:>type lf.txt.lnk
```

```
L\F [y$_ [ ] P O :i+00/C:\:1Bcygwin
$<A"Gcygwin41<L home <"Ghome<1! ...
```



If a Windows program wants to handle shortcuts, it's got to have special code to do it!

Symbolic links, continued

File-related utility programs often have special handling for symbolic links.

One example is `ls`, whose `-L` option says to "follow" the link.

```
~/352/a2 % ls -l x  
lrwxrwxrwx 1 whm whm 12 Sep  1 17:34 x -> a2/  
lengths.1
```

```
~/352/a2 % ls -lL x  
-rw-r--r-- 1 whm whm 18 Aug 29 21:21 x
```

```
~/352/a2 % ls -l a2/lengths.1  
-rw-r--r-- 1 whm whm 18 Aug 29 21:21 a2/lengths.1
```

Try `ls -l /usr/share/dict/words`. Try it again with `-lL`.

Symbolic links

I've got a lot of symbolic links. Here are some on my Mac:

```
% ls -l /w
```

```
lrwxr-xr-x 1 root wheel 9 Jun 7 2013 /w -> Users/whm
```

```
% ls -l /e
```

```
lrwxr-xr-x 1 whm admin 11 Dec 9 2009 /e -> /Volumes/e/
```

```
% ls -l ~/352/files/unix.pptx
```

```
lrwxr-xr-x 1 whm staff 18 Aug 23 17:45 /Users/whm/352/files/  
unix.pptx -> ../unix-clean.pptx
```

```
% ls -l ~/3bin
```

```
lrwxr-xr-x 1 whm staff 10 Sep 16 2013 /Users/whm/3bin ->  
src/337bin
```

Cygwin:

```
% ls -l /c
```

```
lrwxrwxrwx 1 TheBoss None 11 Jun 3 2010 /c -> /cygdrive/c
```

lectura:

```
% ls -l ~/www
```

```
lrwxrwxrwx 1 whm dept 18 Aug 20 2009 /home/whm/www ->  
/cs/www/people/whm
```

The mv command

The `mv` ("move") command can be used to rename files or move files from one directory to another.

Let's rename `users` to `usercount`:

```
% mv users usercount
```

If `mv`'s destination file already exists, it is silently overwritten unless `-i` is specified.

Directories can be renamed, too:

```
% mv 352 csc352
```

If `mv`'s last argument is a directory, the preceding entries (arguments) are moved into that directory.

```
% mv x y p1/notes.txt java/test.java ~
```

```
% mv ~/test.java ~/notes.txt .
```

(Note that `.` is used for destination--"move them here".)

To rename (not move!) many files at once, see `rename (1)`.

Odds and ends

Here are some handy options for `ls`:

- t Sort by modification time instead of alphabetically.
- h Show sizes with human-readable units like K, M, and G.
- r Reverse the order of the sort.
- S Sort by file size
- d By default, when an argument is a directory, `ls` operates on the entries contained in that directory. `-d` says to operate on the directory itself. Try "`ls -l .`" and "`ls -ld .`"
- `--full-time`
Show times with full resolution.

Two handy options for `cp`:

- r Recursively copy an entire directory tree
- p Preserve file permissions, ownerships, and timestamps
`cp -rp ~/352 ~/352.bak.20150901`
- L "Follow" symbolic links.

Deleting files and directories

The `rm` (remove) command is used to permanently delete one or more files:

```
% rm tmp.out Hello.java.old a b c
```

To remove a directory, use `rmdir`.

```
% rmdir x
```

A directory must be empty before it can be removed with `rmdir`, but `rm`'s `-r` (recursive) option can be used to remove a directory and all its contents.

```
% mkdir -p y/z
```

```
% rmdir y
```

```
rmdir: y: File exists
```

```
% rm -rf y
```

```
%
```

The `-i` option of `rm` causes a prompt before a file is removed.

The `-f` option forces removal of read-only files. It also suppresses warnings about non-existent files.

See <http://cs.arizona.edu/computing/accounts/snapshots.html> for a description of a facility that allows recovery of files. (ZFS facility.)

REPLACEMENTS!

Discard the 144-154 set from Sep 4.

bash Customizations and Conveniences

Sidebar: Programs and processes

The executable code and literal data for a program is held in a file.

```
% ls -l /bin/bash /bin/date /bin/ls
-rwxr-xr-x 1 root wheel 959120 Oct 7 2014 /bin/bash
-rwxr-xr-x 1 root wheel 59984 Jan 13 2015 /bin/date
-rwxr-xr-x 1 root wheel 105840 Jan 13 2015 /bin/ls
```

When we run a program with bash, the program code and literal data are loaded into memory, and execution of the code begins.

A running copy of a program is called a process.

We can say a process is an instance of a program.

A program can be likened to a Java class: both contain code and constant data.

Starting a process is somewhat like calling the constructor of a Java class. In both cases we end up with an in-memory entity that contains code and data.

An object is an instance of a class.

A process is an instance of a program.

Good news and bad news

Good news:

The behavior of bash can be customized by putting commands in the initialization files that bash reads.

Bad news:

Several files and rules are involved.

The initialization files present and their contents vary from user to user based on when the CS account was created.

We'll first talk about the mechanics of bash's initialization files and then look at types of things we might add to initialization files.

Anything that's valid at the bash prompt can appear in an initialization file and vice-versa. In other words, initialization files simply contain a sequence of bash commands.

The rules (simplified), and my practice

If bash is specified as your shell in `/etc/passwd` and you login, the instance of bash that's started is said to be a *login shell*.

When bash is started as a login shell it first reads `/etc/profile`. It then looks for three files in turn: `~/ .bash_profile`, `~/ .bash_login`, and `~/ .profile`. Upon finding one, it executes the commands in that file and doesn't look any further.

Sometimes you'll want to start another instance of bash from the bash prompt:

```
% bash
```

```
%
```

Such an instance of bash is an "interactive non-login shell". It reads `/etc/bash.bashrc` and `~/ .bashrc`.

My practice:

- (1) `~/ .profile` has a single line, which loads `~/ .bashrc`.
- (2) All my customizations are in `~/ .bashrc`.

Creating a "starter" bash configuration

Note: This procedure is for students who have done no customization of their bash startup files on lectura.

Use **cd** with no arguments to go to your home directory.

Confirm that you've got **.profile** but not **.bash_profile** or **.bash_login**

Make a directory **bashoriginals** and move (**mv**) **.profile** and **.bashrc** into it.

Create **.profile** and put one line in it:

```
source ~/.bashrc
```

Create **.bashrc** and put these lines in it: (Details on all will follow.)

```
PS1="\w % "
```

```
PATH=$PATH:~/bin
```

```
alias ll="ls -l"
```

```
alias restart="source ~/.bashrc"
```

When making major changes to my bash initialization files, I usually log out and back in, to check for errors.

We'll learn about the commands in the following slides.

Shell variables

bash supports variables that hold values of various types, including integers and arrays but we'll focus on string-valued variables, which are the default.

bash variables have four common uses:

- Specification of various bash behaviors
- Access to information maintained by bash
- Command line convenience variables
- Use as a conventional variable for programming (later, maybe)

The variable `PS1` falls in the first category. Its value specifies the primary bash prompt. We set it in our simple `.bashrc`:

```
PS1="\w % "
```

If we assign a value to `PS1`, the next prompt reflects it:

```
~ % PS1="What now, master? "
```

```
What now, master? PS1="C:>"
```

```
C:>
```

Search for `PROMPTING` on the bash man page to see the various escape codes that are recognized, like `\w`. There's `PS2`, `PS3`, and `PS4`, too!

Variables that control bash behavior

Here's a sampling of other variables that control bash's behavior:

```
FIGIGNORE=.class:.o
```

When doing TAB completion of file names, ignore file names with the suffixes `.class` and `.o`

```
HISTSIZE=5000
```

```
HISTTIMEFORMAT='%Y%m%d-%H:%M:%S: '
```

The number of commands shown by `history`, and how their times are formatted. (See `man strftime` for the recognized time formatting specifiers.)

```
PROMPT_COMMAND
```

The name of a command to execute before the `PS1` prompt is printed.

Variables that make information available

bash makes a variety of current shell-centric information available as variables. Two simple ones are PWD and OLDPWD.

Variables are accessed by prefixing their name with a dollar sign:

```
% echo $PWD  
/home/whm/352
```

Practical example with OLDPWD:

```
% pwd  
~/src/ruby/examples
```

```
% cd ~/work
```

```
% cp $OLDPWD/fbscores.rb .
```

Another is RANDOM:

```
% echo $RANDOM $RANDOM $RANDOM  
7947 27666 7603
```


Command line convenience variables

Let's add three more lines to our simple `.bashrc`: (no spaces around `'=!`)

```
f352=/cs/www/classes/cs352/fall15  
a2=$f352/a2          # concatenation!  
mya2=~ /352/a2
```

We'll reload our `.bashrc` with our `restart` alias and then check variables.

```
~ % restart  
~ % echo $f352  
/cs/www/classes/cs352/fall15  
~ % echo $a2  
/cs/www/classes/cs352/fall15/a2  
~ % echo $mya2  
/home/stdntwm/352/a2
```

Let's run the `a2 Tester`:

```
~ % cd $mya2  
~/352/a2 % $a2/tester eval  
...
```

The scope of a (non-exported) variable is the bash instance in which it's created.

The search path for commands (PATH)

Your `PATH` variable specifies the directories that are to be searched when you run a command.

```
% echo $PATH
```

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/  
bin:/sbin:/bin:/usr/games:/usr/local/rvm/bin:/  
home/stdntwm/bin
```

```
% echo $PATH | tr : " " | wc -w
```

```
9
```

If I type "`xyz`" at the prompt, `bash` will search each of those nine directories in turn for an executable file named `xyz`. If it finds an `xyz`, it will run it, and look no further.

What's a key benefit of being able to specify a search path for commands?

I can specify additional sets of commands to run!

What does the last path entry specify?

PATH, continued

At hand: (the command search path for my student account)

```
% echo $PATH
```

```
...various system "bins"... : /home/stdntwm/bin
```

A common convention is to put your own commands in `~/bin`.

Some programmers have more than one "bin".

```
3bin  ccbn  ebin  mbin  qqbin  winbin
```

```
bin   cwbin  jbin  qbin  sbin
```

(732 commands in those)

Speculate: Does running "`a2/tester`" involve the search path?

No. If there's a / in the command, it simply runs that file.

Another example: `./x`

Setting the path in ~/ .bashrc .

Here's a line from the "starter" ~/ .bashrc. What's it doing?

```
PATH=$PATH:~/bin
```

Append ":~/bin" to whatever PATH already is.

Java analog:

```
PATH = PATH + " :~/bin";
```

When bash starts up, PATH gets set to a system-specific value.

The net effect of `PATH=$PATH:~/bin` is "If you don't find a command in the standard directories on this system, look in my bin."

The truth: With our starter .bashrc you'll see this:

```
% echo $PATH
```

```
/usr/local/rvm/gems/ruby-1.9.3-p484/bin:/usr/local/rvm/gems/ruby-1.9.3-p484@global/bin:/usr/local/rvm/rubies/ruby-1.9.3-p484/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/rvm/bin:/home/stdntwm/bin
```

Dot danger!

Some programmers have "dot" in their search path (`PATH=...:.`) so that a script `x` in the current directory can be run with "`x`" instead of "`./x`".

"Dot in the path" can be convenient but imagine...

- Instead of typing `ls` you accidentally typed `sl`.
- You happened to be in a "world-writable" directory like `/tmp`.
- A malicious student has put this in `/tmp/sl`:

```
chmod o+rwX ~
```

...or maybe this:

```
rm -rf ~/ & # "&" runs a command "in the background"
```

In the first case, they've gained access to your home directory!

In the second case, deletion of all your CS account files is in progress!

But the above is a combination of a bad guy, a bad neighborhood and a boo-boo!

Is it stupid to have dot in your path?

Dot danger, continued

Like with any risk, you need to weigh risk vs. convenience.

If you're developing a lot of scripts, not having to type `./x` is convenient.

But if you're developing in one place, like `~/352/a2`, you could just put `~/352/a2` in your path temporarily.

I have a number of directory-specific scripts, like "rst", short for "rsync this directory" but I alias `rst="./rst"`.

Having dot in your path on your personal machine might seem perfectly safe but imagine you're simply navigating a tree of data files from a trusted colleague who's not security-saavy.

My recommendation:

Don't put dot in your path on lectura! Be cautious even on your own machine.

BIG PITFALL: An empty entry is considered to be "dot"! How many are below?

```
PATH=:/usr/local/bin::/usr/bin:~/bin:
```

Aliases

Here's the first *alias* in our simple `.bashrc`:

```
alias ll="ls -l"
```

Aliases are another type of command that bash supports. With it in place I can type `ll` instead of `ls -l`.

Because `ls` has good option handling, `ll -t` works, too. (If `ls` required all options to be specified in one argument, `ll -t` wouldn't work.)

Instead of an alias, how about a script `~/bin/ll` with `"ls -l"` in it?

We'd need to add a little code for argument handling. (`ls -l "$@"`)

Here's a sampling of my 283 aliases: (No spaces around the '='s!)

```
alias jc="javac"  
alias gcc="gcc -g -std=c99"  
alias tfl="tail -f -n 50 /flashlog.txt"  
alias +x="chmod +x"  
alias jarwars="open /w/372/jarwars/jarwars.swf"
```

With no arguments, `alias` displays your aliases. Use `\gcc x.c` to ignore the `gcc` alias, for example.

Aliases, continued

Here's the second alias shown in our simple `.bashrc`:

```
alias restart="source ~/.bashrc"
```

With it, we need to type only "restart" to make additions take effect after changing `.bashrc`.

The `source` command, a builtin, executes commands in the current instance of `bash`, as if they'd been typed in.

Contrast: `bash .bashrc` starts a new instance of `bash`, executes the commands in `.bashrc`, and then exits. The aliases and variables created in `.bashrc` would exist only for a brief moment!

We can match the functionality of the `ll` alias with an `ll` script but we can't match `restart` with a script!

Wildcards

Wildcards allow the user to specify files and directories using text patterns in bash commands.

args.java is great for exploring wildcards! Let's use an alias:

```
alias args="java -classpath $f352/java args"
```

The simplest wildcard metacharacter is `?`, a question mark. A question mark matches any one character.

Observe:

```
% ls  
a  out  x  xy  z
```

```
% args ?  
|a|  
|x|  
|z|
```

```
% args ??? ??  
|out|  
|xy|
```

The command line "args ?" causes bash to *expand* the question mark into the names of all matching files.

"args ?" is expanded into "args a x z".

What does "args ??? ??" mean?

Replace ??? with all three-letter names.

Replace ?? with all two letter names.

Run the resulting command, whatever it is.

Wildcards, continued

echo is also good for exploring wildcards, and uses less vertical space on slides:

```
% ls
```

```
a out x xy z
```

```
% echo ?
```

```
a x z
```

```
% echo ??? ??
```

```
out xy
```

Predict the output:

```
% echo ? ? ?
```

```
a x z a x z a x z
```

```
% echo x? ?y
```

```
xy xy
```

```
% echo ? ?? x ??? ?????
```

```
a x z xy x out ?????
```

A general rule:

If a command line argument contains one or more wildcards, the shell replaces that argument with the file names in the current directory that match the specified pattern.

Important:

Wildcards are not regular expressions.

The two facilities share a bit of common behavior but they are far more different than they are alike!

If there's no match, like with `????`, the argument is passed through as-is.

Wildcards, continued

Situation:

```
% ls  
out  p1.c  test  x.java  x2.c  y.java
```

Problem:

Write commands to delete the Java files and move the C files to `csrc` in your home directory.

Solution:

```
% rm *.java  
  
% mv ???.c ~/csrc
```

Problem: Describe the file names that would be matched by each of the following.

???

Three character names.

a??b

Four character names that start with an a and end with a b.

200?-1?-3?.log (assuming all files are YYYY-MM-DD.log)

The last day or two of October, November and December in 2000-2009.

The * wildcard

A more powerful wildcard is * (asterisk). It matches any sequence of characters, including an empty sequence.

* Matches every name

*.java Matches every name that ends with .java

x Matches every name that contains an x

Examples: x, ax, axe, xxe, xoxox

What would be matched by the following?

*x*y

Names that contain an x and end with y.

.

Names that contain at least one dot.

..*

Names that contain at least two dots.

a*e*i*o*u

Names that start with an a, end with a u, and have e, i, o, in sequence in the middle.

Experiment!

/usr/bin has a lot of files. Try some wildcards there!

```
/usr/bin % ls *x*y
```

```
dbiproxy  expiry  iproxy  smproxy  syslinux-legacy  ...
```

```
/usr/bin % ls *.*.* | wc
```

```
39      39      526
```

```
/usr/bin % ls a*e*i*o*u*
```

```
akonadi_mixedmaildir_resource
```

```
/usr/bin % ls *-*-*-*-*
```

```
rarian-sk-get-extended-content-list
```

```
/usr/bin % ls *1*2*3
```

```
ls: cannot access *1*2*3: No such file or directory
```

Make some files with touch and try some matches.

```
% touch a.b ab axe b oxo
```

```
% echo a*b
```

```
a.b ab
```

```
% echo *b
```

```
a.b ab b
```

Combining wildcards

Wildcards can be 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 ? *

Names that are at least three characters long and have an x as their second character.

* - ? - *

Names that contain two dashes separated by a single character.

The character set wildcard

The character set wildcard specifies that a single character must match one of a set.

```
% ls  
a b e n out x xy z
```

```
% echo [m-z] # one-character names in the range n-z  
n x z
```

Another:

```
% ls  
Makefile fmt.c utils.c utils.h
```

```
% echo *. [chy]  
fmt.c utils.c utils.h
```

More:

```
[A-Z]*.[0-9]  
Matches names that start with a capital letter and end with a dot and a digit.
```

```
*.[!0-9] (Leading ! complements the set.)  
Matches names that end with a dot and a non-digit character.  
Equivalent: *.[^0-9]
```

```
[Tt]ext  
Matches Text and text.
```

Wildcards and paths

Slashes can be included in a pattern to match files anywhere.

What do these commands do?

```
wc ~/*.java
```

Runs wc on every Java source file in my home directory.

```
ls -l */Readme.txt
```

Runs ls -l on every Readme.txt in a subdirectory of this directory.

```
more a2/[s-z]*/questions.txt
```

I'd used more a2//questions.txt to browse solutions but I got interrupted. I used the above to start roughly where I'd left off, with NetIDs that start with "s".*

```
ls -ld /usr/lib/*/.
```

List directories in /usr/lib.

Programmer to programmer communication:

"It's pretty much a mess as I've included everything, but generally the files phase[123].[ch] are the students' solutions."

Lots more with wildcards

The bash man page uses the term "pathname expansion" for what I've called wildcard expansion.

Another term that's used is "globbing". Try searching for "glob" on the bash man page.

Wildcards don't match hidden files unless the pattern starts with a dot, like `.*rc`.

There are other wildcard specifiers but `?`, `*`, and `[...]` are the most commonly used.

As of version 4 of bash you can do `shopt -s globstar` in `~/.bashrc` to enable recursive matching with wildcards. Example, using the tree shown on slide 119:

```
~/352 % echo **/*
d1 d1/cal.out d1/java d1/notes d1/notes/langs d1/
notes/langs/java d1/notes/platforms one two work
work/cal.out work/one work/two
```

Prior to version 4 you'd do something like `echo * */* */*/* ...` or use `find` with *command substitution*.

Command substitution

One way to view `echo` is that it turns arguments into output.

```
% echo just testing  
just testing
```

Command substitution provides a way to turn output into arguments.

```
% cat srcfiles  
lc.java  
mkall.icn  
getpid.c
```

```
% echo aaa $(cat srcfiles) bbb  
aaa lc.java mkall.icn getpid.c bbb
```

On a command line, the form `$(command-line)` indicates to run the enclosed *command-line*, and substitute the whitespace-separated words it produces for the `$(...)` construct. The resulting command line is then executed.

Command substitution was originally done with ``...``, and that still works.

Command substitution, continued

In the a2 write-up I show you the sizes of my solutions like this:

```
% wc $(cat a2/delivs)
 12    31   332 lengths.java
 13    38   372 rev.java
 17    47   415 sum.java
...
```

```
% cat a2/delivs
lengths.java
rev.java
...
mgrep.java
amj
revnum
...
```

Problem: Not counting `mgrep.java`, how many lines of Java are there?

```
wc -l $(fgrep java a2/delivs | fgrep -v mgrep)
```

How would we have done that without command substitution?

```
% cp delivs x
% vim x
% wc -l $(cat x)
```

But we didn't have to!

Command substitution, continued

Here's a script that prints a YYYYMMDD.HHMM timestamp:

```
% cat tstamp  
date +%Y%m%d.%H%M
```

```
% tstamp  
20150907.0046
```

Assuming the directory with `tstamp` is in our search path, let's use it to make a timestamped backup of a file:

```
% cp lc.java bak/lc.java.$(tstamp)  
% ls bak  
lc.java.20150907.0046
```

The `cp` above is repetitious! How can we type less?

```
% cat cpstamp  
cp $1 bak/$1.$(tstamp)
```

```
% cpstamp lc.java
```

```
% ls bak  
lc.java.20150907.0046  lc.java.20150907.0048
```


The for loop

bash has a number of *control structures* including `if`, `while`, and `case`.

bash's `for` loop is a control structure that's particularly handy for interactive use.

Here is the *general form* of the `for` loop:

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

Example: (bash prompts with `>` (PS2) while the `for` is incomplete)

```
% for i in a simple test
> do
> echo $i has $(echo -n $i | wc -c) characters
> done
a has 1 characters
simple has 6 characters
test has 4 characters
```

for, continued

Handy: If we hit up-arrow, bash shows us the `for` as a one-liner:

```
% for i in a simple test; do echo $i has $(echo -n $i  
| wc -c) characters; done
```

Note that bash has inserted semicolons where needed to make it a valid one-liner.

Problem: How many files are in each of the directories in my PATH?

```
% for dir in $(echo $PATH | tr : " ")  
> do  
> echo $dir: $(ls $dir | wc -l)  
> done
```

```
/usr/local/sbin: 21  
/usr/local/bin: 79  
/usr/sbin: 353  
/usr/bin: 3275  
/sbin: 183  
/bin: 168  
/usr/games: 10  
/home/stdntwm/bin: 1
```

Problem: What's the total number of files?

Solution:

Hit up-arrow and append...

```
| cut -d " " -f2 | java sum
```

for, continued

Here is "args" as a bash script:

```
% cat args
for a in "$@"
do
    echo "$a"
done
```

Usage:

```
% args one ' 2 ' III a\ b\ c
|one|
| 2 |
|III|
|a b c|
```

Notes:

- "\$@" expands into the arguments of the script, with quoting preserved. \$* is often wrongly used instead of "\$@". (Try it!)
- The echo uses double quotes ("soft quotes") so that \$a is expanded. Try it with apostrophes instead.

Handy stuff in my 2005 slides

This round with UNIX ends here.

Some UNIX topics will be blended into the C material. We might have time for more UNIX-centric material later, like non-trivial scripts, but we might not.

Here are some command-line topics covered in my 2005 slides, on Piazza, that you might find particularly handy for day-to-day use in this class or others.

- The history mechanism: 121-125
- The directory stack: 126
- Brace expansion: 72-74
- Process substitution: 119-120

The "Assorted Utilities" section starting on slide 127 talks about `diff`, `find`, `tar`, `sed`, and regular expressions.

"Files and File Management—Part 2", starting on 167 talks about file permissions and more.

Unless it is also covered in this class, you won't be expected to know any of the material in those 2005 slides.