

## Linux, X, Emacs, and Java

### 1 Lions and Tigers and Bears – Oh My!

This semester, we will be using the CS department’s Linux workstations in the mini-mini-focus for all programming in CS230. There are currently 14 such machines, most of which are named after big cats and bears: druantia, grizzly, jaguar, kodiak (in SCI 173), leopard, lion, lynx, ocelot, panda, polar, rhodes, sloth, teddy, and tiger.

You are not expected to have any experience with Linux coming into this course. As part of doing your assignments, you will become familiar with Linux and related tools (e.g., shells, X Windows, Emacs, the Sun Java system) as the semester progresses. However, in order to complete your first assignment, you will need at least a rudimentary familiarity with Linux, Emacs, and the Sun Java system. It is expected that you will devote a good chunk of time during the first week the semester “getting up to speed” with these systems.

We will be providing some information on using these systems during lab and a special tutorial session, but you are expected to learn many details on your own by studying available documentation and taking on-line tutorials. The purpose of this handout is to provide some simple information and tell you where to find more information.

### 2 Puma Accounts

In order to use a Linux workstation, you must have an account on **puma**, the new CS department fileserver, which replaces the old fileserver, **nike**. If you’ve taken CS111 here, you should already have a **puma** account with the same username and password that you used in CS111. If you do not have a **puma** account, or you have forgotten your password, please contact Lyn ASAP.

No matter which Linux workstation you use, all files in your account are physically stored on the **puma** fileserver and *not* on the Linux workstation itself. For instance, if you are user **gdome**, and you create a file named `~gdome/test.txt` while working on the workstation named **teddy**, this file is actually stored on **puma** and not on **teddy**. You can later view, edit, or delete this file from any other CS department Linux machine.

### 3 Logging In To A Linux Console

The easiest way to do your work in CS230 is to directly log in to one of the Linux consoles in the mini-mini focus or in SCI 173. A console that is not in use displays a Linux login screen, which looks something like this:

```
Red Hat Linux release 7.3 (Valhalla)
Kernel 2.4.18-3 on an i686
```

```
login:
```

To log in, type your username (the same as your “short” FirstClass username) followed by the ENTER key at the **login:** prompt. You will then be prompted for your password, which you should type, followed by the ENTER key.

If the log in is successful, you should be presented with a so-called *shell prompt* that looks something like

```
[username@hostname ~]$
```

where *username* is your username and *hostname* is the name of the machine into which you logged in.

A few things can go wrong when you are logging in:

- If you misspelled or used the wrong username or password, you will be prompted for your username and password again. If you cannot log into the console of a Linux workstation after repeated attempts, use FirstClass on a Windows or Mac machine to send an email message to Lyn *and* Stella.
- If the screen is initially blank, type any key and the login screen should appear.
- If the screen displays a screensaver of some form, it has been “locked” by another student (see thenotes on locking etiquette in Sec. 10) and you cannot use it. Try to log in to another machine.
- If the screen is displaying text that is not the login screen, or if it is displaying windows, then another student is logged into the machine and may still be using it. Try to log in to another machine.

## 4 The Linux Shell

The prompt that you see after logging in is part of a command-line interface to a Linux known as a *shell*. At the shell prompt, you type a Linux command to execute, followed by the ENTER key. Linux will then execute this command, and, upon finishing the execution, will present you with another shell prompt.

This mode of interaction may be unfamiliar if you’ve only had experience with a point-and-click, drag-and-drop window system. Although many tasks are not as convenient in a text-based shell as in a graphical interface, you may be surprised to learn that certain tasks (such as finding all files that satisfy a non-trivial specification, or automating a sequence of tasks) are actually *more* convenient in the text-based interface.

There are a plethora of shell commands for such tasks as navigating through and modifying the file system, searching for files that match certain criteria, finding documentation, and invoking programs like text editors and compilers. For a quick introduction to some very basic commands, see Section 4 (Shell Commands) of Scott Anderson’s article *Introduction to Unix and the X Window System*, which can be found on-line at:

<http://puma.wellesley.edu/user-info/handouts/unix-intro.pdf>

and is linked from the *Resource Links* section of the CS230 home page. For a more detailed introduction, read Chapter 4 (The Unix Shell) of Larry Greenfield’s *The LINUX Users’ Guide*. There are red-bound copies of this guide next to most of our Linux workstations. The guide is also available on-line:

- *HTML version*: <http://espc22.murdoch.edu.au/stewart/guide/guide.html>

- *PDF version:* <http://puma.wellesley.edu/user-info/handouts/linuxUsersGuide.pdf><sup>1</sup>

Once you've mastered the simple shell commands (and are somewhat comfortable with X Windows, Emacs, and Java, as described below), you are encouraged to learn more powerful shell commands. A good starting point is *The LINUX User's Guide*, particularly the following chapters: Chapter 6 (Working with Unix); Chapter 7 (Powerful Little Programs); Chapter 9 (I Gotta be Me!); and Chapter 11 (Funny Commands). Also, the Linux `man` command can be used to find detailed documentation on any command. For example, executing `man ls` gives documentation on the file-listing command `ls`.

## 5 X Windows

Almost all of your work in CS230 (except for the segment on graphical user interfaces at the end of the course) can be done entirely within the text-based interface of the Linux shell and Emacs (see Sec. 6). However, a graphical user interface with windows is more convenient for many tasks.

We will be using the X Windows System ("X" for short) in this course. You can launch X via the shell command `startx`. This will change the display from a text-based interface to a graphical windows interface similar to that on Macs and Windows.

The particular window manager we are using this semester is called *Gnome*.<sup>2</sup> Using Gnome is fairly intuitive. At the bottom left corner of the screen is a footprint icon that serves a purpose similar to the START button in Windows. Click on this icon for a menu of options. Some particularly important options are `Programs>Applications>Emacs` (to launch Emacs) and `Programs>Internet>Netscape` (to launch Netscape). Next to the footprint are several other icons:

- The lock icon "locks" the screen;
- The computer terminal icon creates a new Linux shell window. You can execute any Linux commands in such a window<sup>3</sup>;
- The dinosaur head icon launches Mozilla, a web browser.

In the top left corner of a typical window are three buttons; from left to right:

1. The "dash" button iconizes the window. An iconized window appears in a *TaskList* in the bar at the bottom of the screen;
2. The "window" button enlarges/shrinks the window;
3. The "X" button closes the window.

Unfortunately, the current version of Gnome does not appear to come equipped with a decent help system. Luckily, most interactions with Gnome are fairly straightforward.

---

<sup>1</sup>The PDF version of the guide is difficult to read on screen because of poor font resolution.

<sup>2</sup>In other documentation, you may find references to KDE, which is a similar window manager.

<sup>3</sup>Alternatively, you can create a new shell window by executing the command `xterm &` in an existing shell window.

## 6 Emacs

Emacs is an extensible, customizable, self-documenting text editor created by Richard Stallman. Many (including this author) consider it to be one of the greatest programs of all time. It also happens to be one of the flagship programs of Stallman's Free Software Foundation and GNU project.

You will be doing most of your work this semester – writing, executing, and debugging programs in Java – using Emacs. In fact, it is possible to do almost all your work in the course entirely within Emacs. It is very important for you to become a proficient Emacs user because this will save you a lot of time during the semester.

There are two standard ways to launch Emacs:

- Execute the command `emacs` from within a shell. This works in both the text-based and graphical interfaces.<sup>4</sup>
- Select the Gnome menu sequence `Programs>Applications>Emacs`.

All Emacs documentation, including a tutorial and reference information, is on-line. If you are unfamiliar with Emacs (or have used it before but are rusty), you should take the on-line Emacs tutorial. You can do this by typing the `Control` and `h` keys at the same time, followed by the `t` key.<sup>5</sup> This will load a interactive tutorial, whose directions you should follow. When you complete the tutorial, you will know how to do basic editing in Emacs.

The tutorial teaches you keystroke commands for basic Emacs functionality. If you prefer, most of this functionality can instead be accessed by using a combination of the mouse, menu items, and arrow keys. However, I strongly recommend that you learn the keystroke commands, as they will save you lots of time and make it easier for you to work remotely via telnet and ssh (see Sec. 12).

In addition to taking the tutorial, you should read Scott Anderson's article *Introduction to The Emacs Editor*, which can be found on-line at:

<http://puma.wellesley.edu/user-info/handouts/emacs-intro.pdf>.

Another useful introduction to Emacs is Chapter 8 (*Editing Files with Emacs*) of Larry Greenfield's *The Linux Users' Guide*. You will find links to these and several sites containing more detailed Emacs documentation in the *Resource Links* section of the CS230 home page. A particularly useful link is the Emacs reference card you can find at

<http://www.refcards.com/download/emacs-refcard-letter.pdf>.

You may want to print out a copy of this card and carry it with you for handy reference.

It turns out that Emacs even has its own hypertext information system. This system contains detailed documentation on Emacs itself, and is worth exploring to find out more about Emacs. In order to access this information system, type the `ALT` key and the `x` key at the same time, followed by the character sequence `info`.<sup>6</sup> This will load up an editor buffer that contains a top-level menu of the system documentation. You can browse this system via mouse clicks, much as you browse web pages in a web browser.

---

<sup>4</sup>In the graphical interface, executing `emacs` will launch Emacs in a separate window. If you wish emacs to appear within the same window as the shell, instead execute `emacs -nw` (the `-nw` means “no window”).

<sup>5</sup>In Emacs notation, this keystroke combination is usually written `C-h t` and pronounced “control-h t”.

<sup>6</sup>This keystroke combination, pronounced “meta-x info”, is usually notated as `M-x info`.

The Emacs command `M-x shell` creates a shell that runs inside of an Emacs buffer. It is very convenient to have a shell within Emacs, because then any shell command can be easily executed without leaving Emacs. This can be especially important if you are accessing a Linux machine remotely via telnet or ssh (see Sec. 12).

One minor drawback of running a shell under Emacs is that Emacs sometimes interprets or prints character sequences in a different way than a separate shell window would. For instance, an Emacs shell will echo passwords that a normal shell would not. Also, the `ls` command in an Emacs shell may print a lot of annoying formatting characters; these can be removed by first executing `unalias ls` in the Emacs shell.

## 7 Java

We will use Java for all programming in this course. Our development environment will be Sun's Java 2 Software Development Kit (SDK), Version 1.4.0.

Using the Java SDK in Linux is rather different than using Java in CodeWarrior. The following subsections present the main things you need to know about using the Java SDK.

### 7.1 Editing Java Source Files

All editing of Java source files can be done with Emacs. As a simple example, you can use Emacs to create a file named `Test.java` with the following contents:

```
public class Test {  
  
    public static void main (String [] args) {  
        System.out.println("This is a test.");  
    }  
  
}
```

Since the filename ends with `.java`, Emacs “knows” that it is a Java source file and will enter a Java mode that facilitates the writing of Java code. For instance, Java mode uses different colors to displays different kinds of syntactic entities (e.g., keywords, types, method names, strings, etc.) and will match close braces with open braces. Java mode also “knows” the correct indentation for statements; typing `TAB` on any line will indent that line according to Java pretty-printing conventions.

### 7.2 Compiling Java Source Files

A Java compiler translates a Java source file (i.e., `.java` file) into one or more `.class` files that are suitable for execution on a Java Virtual Machine (JVM). In the Sun SDK, the java compiler is invoked via the `javac` command. For example, executing

```
javac Test.java
```

generates the class file `Test.class` for the sample program presented above.

It can sometimes take `javac` a long time to compile a file. Unless there are errors, `javac` will not give any feedback about where it is in the compilation process. If you want more feedback, use the `-verbose` option. Here is an example of the information provide by this option:

```

[cs230@koala linux] javac -verbose Test.java
[parsing started Test.java]
[parsing completed 200ms]
[loading /usr/java/j2sdk1.4.0/jre/lib/rt.jar(java/lang/Object.class)]
[loading /usr/java/j2sdk1.4.0/jre/lib/rt.jar(java/lang/String.class)]
[checking Test]
[loading /usr/java/j2sdk1.4.0/jre/lib/rt.jar(java/lang/System.class)]
[loading /usr/java/j2sdk1.4.0/jre/lib/rt.jar(java/io/PrintStream.class)]
[loading /usr/java/j2sdk1.4.0/jre/lib/rt.jar(java/io/FilterOutputStream.class)]
[loading /usr/java/j2sdk1.4.0/jre/lib/rt.jar(java/io/OutputStream.class)]
[wrote Test.class]
[total 751ms]

```

Recall that in CodeWarrior a “project file” (.mcp file) is used to declare which source (.java) files are collected together to form a program. Unlike CodeWarrior, the Java SDK has no notion of a project file. As a default, it is assumed that all the user source files for a program will be in the same directory<sup>7</sup> and that the user will compile the “main” source file with `javac`. The compiler determines all the files that the “main” source file ultimately depends on, and recursively compiles these as well.

For example, suppose that the the contents of the three files `Pair.java`, `Swap.java`, and `SwapTest.java` are as shown in Fig. 1. Executing `javac SwapTest.java` will not only compile `SwapTest.java` (generating `SwapTest.class`), but will also compile `Swap.java` (generating `Swap.class`) and `Pair.java` (generating `Pair.class`), because the `SwapTest` class “depends on” both the `Swap` and `Pair` classes.

### 7.3 Executing Java Applications

To execute a Java *application*, invoke the `java` command on the name of the class containing the desired `main()` method. For instance, here is a transcript of invoking the two sample programs considered above:

```

[cs230@koala code] java Test
This is a test.

[cs230@koala code] java SwapTest
Before swap: <foo, bar>
After swap: <bar, foo>

```

Note that `java` is invoked on the name of the *class* and *not* on the name of the *class file*. For instance, the invocation `java Test.class` yields an error:

```

java Test.class
Exception in thread "main" java.lang.NoClassDefFoundError: Test/class

```

---

<sup>7</sup>It is possible to have source files spread out over several directories, but we will not consider that here.

```

//The contents of Pair.java
public class Pair {
    public Object left, right;
    public Pair (Object left, Object right) {
        this.left = left;
        this.right = right;
    }
    public String toString () {
        return "<" + left.toString() + ", " + right.toString() + ">";
    }
}

//The contents of Swap.java
public class Swap {
    public static void swap (Pair p) {
        Object temp = p.left;
        p.left = p.right;
        p.right = temp;
    }
}

//The contents of SwapTest.java
public class SwapTest {
    public static void main (String [] args) {
        Pair p = new Pair ("foo","bar");
        System.out.println("Before swap: " + p.toString());
        Swap.swap(p);
        System.out.println("After swap: " + p.toString());
    }
}

```

Figure 1: The contents of three Java source files.

## 7.4 Executing Java Applets

To execute a Java *applet*, invoke the `appletviewer` command on an `.html` file containing an applet tag referring to class file for the desired applet. For instance, suppose that `KnitWorld.class` is the class file for the CS111 applet that displays M.C. Escher's knitting patterns. Then the applet can be invoked via `appletviewer KnitWorld.html`, where `KnitWorld.html` is a file containing the following applet tag:

```
<applet code="KnitWorld.class" width=375 height=400>
</applet>
```

The above example assumes that `KnitWorld.html` and `KnitWorld.class` are in the same directory. If `KnitWorld.class` is in a different directory, the applet tag must contain an appropriate `codebase` attribute. For instance, if `KnitWorld.class` is in a subdirectory named `Java Classes` (as is often the case in CodeWarrior), the appropriate applet tag in `KnitWorld.html` is as follows<sup>8</sup>:

```
<applet code="KnitWorld.class" codebase="Java%20Classes" width=375 height=400>
</applet>
```

## 7.5 Debugging

Finding and fixing bugs (i.e., errors) in your programs is an essential part of the programming process, so it is important to hone your debugging skills in this course. Bugs can be classified into three categories:

- *Compile-time Bugs*: These are errors in the syntax (i.e., form) of the program that are noticed and reported by the compiler. Classic compile-time errors in Java include forgetting a semicolon or brace, omitting the type of a parameter or the return type of a method, misspelling the name of a class or variable, and confusing an instance method or variable with a class method or variable.

When `javac` encounters such an error, it prints an error message with the name of the file and a line number indicating the location of the error. Treat this line number as a hint rather than an absolute fact – often the *real* error is a few lines before or after the reported location. To locate the error, view the file in Emacs and use the M-x `goto-line` command to jump to the offending line.

The compiler often reports several errors. Since a single mistake can often cascade into what the compiler thinks are many different errors, you should focus only on the first error it reports.

- *Run-time Bugs*: These are errors that are not caught by the compiler but are found and reported by the Java Virtual Machine (JVM) when running the program. Classic run-time errors include attempting to invoke a method on the null pointer, accessing an array at an out-of-bounds index, or casting an object to an inappropriate type.

Tracking down a run-time bug involves thinking carefully about the execution of the program – how does the program get into a state where the reported error occurs? To figure this out, it often helps to insert `System.out.println` statements at judiciously chosen spots in the

---

<sup>8</sup>The `%20` appearing in the `codebase` tag denotes a space character

program. Based on the information printed by these statements before the run-time error occurs, it is often possible to narrow down where the error occurs.

Graphical models like the Java Execution Model and Java Object Diagrams are also helpful aids in debugging. Expect to draw lots of diagrams on scrap paper as part of the debugging process.

- *Logical Bugs*: These are errors in the behavior of the program that are *not* reported by the Java compiler or the JVM but instead show up in testing the program. Classic examples of logical bugs are a sorting method that doesn't correctly sort elements or an array-summing method that always returns 0. As with run-time bugs, good techniques for finding logical bugs are `System.out.println` statements and drawing lots of diagrams on scraps of paper. Another important technique is running the program on a wide variety of test cases to get a better sense of the kinds of conditions under which it misbehaves.

As a good overview to debugging, you should read the *Debugging* appendix from Allen Downey's *How to Think Like a Computer Scientist*, which can be found on-line at:

<http://puma.wellesley.edu/~cs230/debug.pdf>.

This contains some very nice advice and concrete examples relevant to debugging.

## 7.6 Developing Programs

In CS111, most of your programming activity revolved around modifying existing programs. In contrast, in CS230 you will be writing many programs from scratch. This is a very different activity from modifying an existing program. In particular, just as essay writers experience the problem of the blank piece of paper, program writers experience the angst of the blank screen. Where do you start?

One important strategy in this regard is *incremental program development*. Start with a very simple program that does something related to the problem you are solving, and then incrementally modify it to solve more and more of the actual problem. The key is having a working program at each intermediate step.

For example, if asked to write an array sorting program, you might first write a trivial `sort` method that leaves the array unchanged and embed it in a program that simply prints out the contents of some test arrays before and after calls to the `sort` method. Next you might modify `sort` to find the smallest element of the array and swap it with the first element. Because you have already wrapped the `sort` method in testing code, it is easy to test if this modification to `sort` works as expected. Finally, you can modify `sort` to embed the swap-the-minimum-element code in a loop so that the whole array becomes sorted.

For some nice advice on how to develop programs from scratch, please read the *Program Development Plan* appendix from Allen Downey's *How to Think Like a Computer Scientist*. This can be found on-line at:

<http://puma.wellesley.edu/~cs230/develop.pdf>.

## 7.7 Java Documentation

You will spend much time in this course studying Application Programming Interfaces (APIs) for Java classes and interfaces – both those that are officially part of the Java SDK, and those that are specific to CS230. Links to relevant APIs will be posted in the *Resource Links* section of the CS230 home page. You can browse all APIs for version 1.4 of the Java 2 SDK at:

<http://java.sun.com/j2se/1.4/docs/api/>.

## 8 Printing

There are two standard ways to print your files from a Linux cluster machine (the second gives nicer looking output):

- Within a shell, execute `lpr filename`.
- Within Emacs, select the `File:Print Buffer` or `File:Postscript Print Buffer` menu options.

Either of these options will print your document on printer `psci11`, which is the smaller printer near the mini-focus consultant's desk. If you use `lpr`, you can print to a different printer using the `-P` option. E.g., `lpr -Ppsci1r Test.java` (note that there is no space between `-P` and `psci1r`).

In the past, the connections from some of the cluster machines to `psci11` have been flaky (e.g., nothing prints out when you try the above). It remains to be seen if the flakiness persists this year. If you experience printing problems, please report them to our Linux system administrators, Scott Anderson and Susan Kohler.

## 9 Saving Work

In addition to saving work in your `puma` home directory, you should make backup copies of your work on a Zip disk or on your own personal computer. For instructions on how to use a Zip disk with the Linux cluster machines, please read Section 8 (Using Removable Disks) of Scott Anderson's *Introduction to Unix and the X Window System*. To copy your work from `puma` to another computer, use a file copying program like `Fetch` (on a Mac) or `WS-FTP` (on a PC).

## 10 Locking A Linux Machine

If you want to leave the mini-mini for a short break, you can “lock” your console by clicking on the lock icon on the menu bar. This will lock the screen in such a way that your password is required to unlock it. You should only lock machines for *short* breaks (as a rule of thumb, no more than 15 minutes). Otherwise, you will be tying up an important resource that someone else may need to use.

## 11 Logging Out Of A Linux Machine

After you are done using a Linux workstation, you need to log out.

- If you are using the Gnome window manager, logging out is a two-step process:
  1. Exit the Gnome window manager by selecting the `Log Out` option from the Gnome “footprint” menu. This should exit the window manager and bring you back to the Linux shell.
  2. Log out of the Linux shell by executing `logout` or `exit` at the Linux shell prompt.

**Do *not* forget the second of the above two steps, otherwise you will be leaving your account open to accidental or malicious abuse.**

- If you are just using a text-based interface (no windows), just type `logout` or `exit` at the Linux shell prompt.

You know that you have succeeded in logging out when you see the Linux login prompt appear.

It is important not to accidentally leave yourself logged in to a Linux machine when you are done. If you do so, someone may accidentally or purposely read, modify, or delete your files. Also, you will be tying up an important resource.

## 12 Using Linux Machines Remotely

You do not have to be physically seated in front of one of the Linux workstations in order to use it. You can access any of the department's Linux machines remotely via programs called `telnet` or `ssh` (a more secure version of `telnet`). Examples of client programs supporting `telnet` and/or `ssh` on various platforms are:

- *Macintosh*: BetterTelnet (telnet only) and NiftyTelnet (telnet and ssh);
- *Windows*: QVTTerminal (telnet only);
- *Linux*s: You can connect from one Linux machine to another via the `telnet` or `ssh` commands. E.g., to access `teddy` from `kodiak`, execute one of the following in a shell on `kodiak`:

1. `telnet teddy`
2. `ssh -l username teddy`

You can use one of clients list above to connect to a remote Linux machine using your `puma` username and password. A list of all CS department Linux machine names appears at the top of this handout.

There are two key advantages of accessing a Linux machine remotely. First, you can access the Linux machines from any other machine on the Internet, including Macs and PCs – a fact which is important when the Science Center is closed or you don't wish to walk there. Second, via `telnet/ssh` you can still use the machines even when all consoles are actively being used (several people can be logged into the same Linux machine at once). This is important to know when the mini-mini is crowded with people near a problem set deadline.

A disadvantage of using `telnet` is that a `telnet` clients provide only a text-based interface, so you will not be able to use the graphical user interface familiar from the console. However, since very little in the course depends on graphics, you can work on most assignments in the course via `telnet/ssh`. (It helps to be familiar with Emacs control- and meta- key commands!)

Note: there are programs (such as Hummingbird's Exceed software) for both Macs and PCs that can display X windows from a remote Linux machine, but these are not standard on the public cluster machines at Wellesley.