Java Programming in the UNIX[1] Environment

Thanks to all the past 107 staff who contributed to this handout.

Introduction

In writing Java programs to run under Unix, there are several concepts and tools that turn out to be quite useful. The most obvious difference, if you are coming from a PC or Macintosh programming background, is that the tools are separate entities, not components in a tightly coupled environment like Thetis or Borland C. The appendix at the end of the handout gives a summary of some basic UNIX and EMACS commands.

The most important tools in this domain are the editor, the compiler, the interpreter, and the debugger. There are a variety of choices as far as "which compiler" or "which editor", but the choice is usually one of personal preference. The choice of editor, however, is almost a religious issue. `EMACS` integrates well with the other tools, has a decent interface, and is almost an operating system unto itself, so we will recommend its use.

Caveat

This handout is not meant to give a comprehensive discussion of any of the tools discussed, rather it should be used as an introduction to getting started with the tools. If you find that you need more information, all of the programs have extensive `man` pages, `xinfo` entries, and web pages fully describing their features. These man pages list numerous bits of information, some of which will appear to be quite attractive, but if you do not know or understand what an option does, especially for the compiler, please do not use it just because it sounds nice.

Also, O'Reilly & Associates publishes a pretty good set of references for these basic tools, the titles of which resemble "UNIX in a Nutshell". These are not required or endorsed (for the record), but may be useful if you prefer having a book reference. Also, the Sweet Hall consulting services hold classes covering using the leland workstations, see their web page http://consult.stanford.edu for more information.

The Editing Process

The first frontier a person is confronted with when programming is the editing environment. As mentioned, the choice of editor in UNIX was and still is, for various valid and not so valid reasons, pretty much set in stone as the 11th commandment: Whosoever writes programs with EMACS[2] shall live without bugs, or else... EMACS has pretty much all the features you will possible need in an editor, and then some; the only problem is finding out how to invoke them as needed. This section is only meant as a quick reference (note that extremely good reference cards and handouts on EMACS can be obtained from Sweet Hall) to get you started. To start editing a new or existing file using EMACS, simply type this to the UNIX prompt:

`emacs` *filename*

where *filename* is the file to be edited. Once EMACS has started up, text entry works much the same way as in any other text editors, i.e. the default mode is INSERT mode where text being typed is inserted at the current cursor position. All the fancy editing commands, such as find-and-replace, are invoked through typing special key sequences (however, if you have an X-capable

[1] UNIX is a trademark of some large faceless corporation.

[2] In the 4th Century AD, Rome was plagued by the worshipers of the Church of Emacs. They never bathed. They would gather in large, foul-smelling groups and harass small children.... - Cornell Quote

terminal or are willing to make the trek to Sweet Hall, you can enjoy the advantages of the Xemacs version which includes the standard mouse-driven operation). Two important key sequences to remember are: ^x (holding down the "ctrl" key while typing 'x') and [esc]-x (simply pressing the "esc" key followed by typing 'x'), both of which are used to *start* command sequences. Note that in most user manuals for EMACS, the "esc" key is actually referred to as the "Meta" key (let's not bother explaining that one). Therefore, you will often see commands prefaced with the key sequence M-x, as opposed to the [esc]-x that we will use in this handout. Since the two are pretty much synonymous, we'll stick with explicitly using the "esc" key.

Now let's see some examples of these mysterious "command sequences". For instance, to save the file being edited the sequence is ^x^s. To exit (and be prompted to save) EMACS, the sequence is ^x^c. To open another file within EMACS, the sequence is ^x^f ('f' for "find file"). This sequence can be used to open an existing file as well as a new file. If you have multiple files open, EMACS stores them in different "buffers." To switch from one buffer to another (very handy when you are editing several `.java` source files), you use the key sequence ^x followed by typing 'b' (without holding down "ctrl" when typing 'b'). You can then enter the name of the file to switch to the corresponding buffer (a default name is provided for fast switching).

The arrow keys usually work as the cursor movement keys, but in the event that they don't, ^f is for moving forward (right), ^b is for moving backward (left), ^p is for moving to the previous line (up), and ^n is for moving to the next line (down). [esc]-< ([esc] followed by '<') moves to the beginning of the file, and [esc]-> moves to the end of the file. Finally, ^v does a "page down" and [esc]-v does a "page up." **Note that the [esc] key is simply pressed and not held, whereas the [ctrl] key is held in all cases.** For a listing of some other functions, please consult the appendix at the end of this handout.

Pico is another editor similar to emacs that may be familiar to users of the pine mail program. Pico does not have the power of emacs, but it is a little more user-friendly. A menu of commands is displayed at the bottom of the screen to help remind you of your options. The main menu looks like this:

```
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Pg   ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where is  ^V Next Pg   ^U UnCut Text^T To Spell
```

Pressing ^g will display a help page explaining the commands.

Because pico does not support many capabilities useful to developers, we usually recommend that you use emacs, but the most important thing is that you can navigate and edit in an environment that works for you.

The Compilation Process

Before going into detail about the actual tools themselves, it is useful to review what happens during the construction of an executable java program. Your source files (.java files) contain the code that will become your running program. This code must be compiled down into a .class file. This file contains "java bytecode", which is a lower-level representation of the program in something similar to assembly language. The key difference between bytecode and assembly languages is that once a program is compiled into bytecode, that bytecode can be run on ANY machine through the java interpreter with no need to recompile for each individual machine. Notice this is a huge difference from other traditional compiled languages like C, where you must recompile for each machine you intend to run your program on. For example, you could compile your program on a UNIX machine, copy the .class files over to a Mac, and run the program on your Mac with no recompilation necessary.

Makefiles will be provided for the programs in your assignments. All you need to do is add your .java files to the SOURCES line as you add classes. Now you can compile your program the same way as traditional C programs, by just issuing the "make" command. However, it is good to know how to compile and run java programs outside the veil of a makefile. It is really just one simple step:

```
javac myfile.java
```

Javac is Sun's java compiler. You simply give it the name of a .java file and it will compile it into a .class file, or let you know about errors you need to fix. One thing to note is what happens when you have several .java files that make up your entire program. When you compile a particular .java file, the compiler looks to see what other files (classes) it uses. If these classes haven't been previously compiled (no .class file exists for them), then it will go ahead and automatically try to compile these .java files as well. It's a good thing to know about, since it can be confusing trying to compile one file and getting errors for a totally different file. If you want to compile your entire program in one swoop, just compile the file that contains your main() method, since this file necessarily contains all the dependencies in your full program.

NOTE: Javac only allows one class per file and it the filename MUST be the same as the class name. Ex: if I had a class "Shape" then it must be in a file called "Shape.java" (capitalization matters!) and it must be the ONLY public class defined in that file. Metrowerks CodeWarrior is more lax and lets you define several classes in the same file, but javac does not since it obeys the language rules more strictly.

NOTE: If a .class file already exists for a .java file, then don't assume it will be automatically recompiled when changes in the .java file are made. Ex: often times it is convenient to rebuild the entire project by recompiling the class with the main() method. However, if you changed something in another .java file, it sometimes won't get recompiled automatically. If you would like to force a full-rebuild of your project , you can delete all the .class files (or "make clean" if using a makefile) and then recompile from the beginning.

Java library source

The source for all the classes in the standard Java packages is compressed into a zip file at `/usr/pubsw/apps/java/src.zip.` We have an uncompressed version available in the "other" section of our class directory that you can view. It is exactly the same library source used by Metrowerks, which is available inside the Metrowerks Java Support folder. Although mostly you will read the class specifications (available on Sun's documentation site) or refer to Eckel to learn about standard classes, sometimes its helpful to also go directly to the source. You have the full source and the classes are somewhat well documented, so you find it interesting and helpful at times to check it out.

Running

To run your code, type java <name>, where <name> is the name of the class containing the main method. For example:

```
java Yahtzee
```

You don't append .class to the name, it knows to look for a file called Yahtzee.class to load and then finds and executes the main() method of that class.

The CLASSPATH environment variable

When you execute a Java program, you give the name of the class to load, but you don't specify exactly where the interpreter will find the appropriate .class file. The interpreter has a standard list of directories that it searches to find .class files as needed. That list of directories is known as your CLASSPATH (not unlike the UNIX shell's PATH variable). CLASSPATH is an environment variable and it lists the directories in order that they will be searched. The first directory that contains the desired class file (potentially in a subdirectory if inside a package) is used. The default CLASSPATH usually includes the current working directory and has entries for the installation directories for the standard Java classes. On leland, this default should work just fine and you shouldn't need to change it, but you will need to run the interpreter and debugger from the directory where your class files are.

The Debugger (jdb)

During the course of the quarter you may run into a bug or two in your programs[3]. There are a variety of different techniques for finding these "anomalies," but a good debugger can make the job a lot easier and faster. While `jdb` does not have a flashy graphical interface, it is a reasonable command-line tool that gives the knowledgeable programmer the ability to observe the working internals of a program and track down execution problems. It's not nearly as sophisticated as the gdb or dbx debuggers you may have used for C programs, but it provides the basics.

This section does describe all of the features of `jdb`, but will hit on the high points. There is some on-line help for `jdb` which can be seen by using the `help` command from within `jdb`.

A debugger is invaluable to a programmer because it eases the process of discovering and repairing bugs at run-time. In most programs of any significant size, it is not possible to determine all of the bugs in a program at compile-time because of oversights and misconceptions about the problem that the application is designed to solve.

The way debuggers allow you to find bugs is by allowing you to run your program on a line-by-line basis, pausing the program at times or conditions that you specify and allowing you to examine variables, registers, the run time stack and other facets of program state while paused.

Sometimes these bugs result in your program exiting prematurely with a message like "Runtime Exception" or the like. If your program has a severe bug that causes a program termination, the debugger will "catch" the signal that indicates the error it found, and allow you to further examine the program. This information can be quite valuable when trying to reason about what caused your program to halt.

Compiling with debug information

To ensure that the compiler includes the proper information for the debugger, you should invoke javac with the -g flag. (The make debug option of our makefile will do this for you if desired). You can debug a program that wasn't compiled for debugging, but it will have less information about the variables and classes which makes it harder to sort out what is going on.

Starting the debugger

To start the debugger from the shell just type `jdb <TargetName>` where `<TargetName>` is the name of the class with the main() method that you want to debug, like this :

```
jdb Yahtzee
```

[3] We recommend that you have fewer.

(note you do not include the ".class" part of the name at the end). If you do not specify a target then jdb will start without a target and you will need to specify one later before you can do anything useful.

Running the debugger
Once started, the debugger will load your application and its symbol table (which contains useful information about variable names, source code files, etc.). This symbol table is the map that the debugger reads as it is running your program.

The debugger is an interactive program. Once started, it will prompt you for commands. The most common commands in the debugger are: setting breakpoints, single stepping, continuing after a breakpoint, and examining the values of variables.

When a target application is first selected (usually on startup) the current source file is set to the file with the `main` function in it, and the current source line is the first executable line of the this function.

As you run your program, it will always be executing some line of code in some source file. When you pause the program (using a "breakpoint"), the "current target file" is the source code file in which the program was executing when you paused it. Likewise, the "current source line" is the line of code in which the program was executing when you paused it.

Breakpoints
You can use breakpoints to pause your program at a certain point. A breakpoint is set by using the command `break` specifying the location of the code where you want the program to be stopped. This location can be specified in two different ways, by line number or by method name. Breakpoint-related commands are shown here:

```
classes                      -- list currently known classes
methods <class id>           -- list a class's methods
stop in <class id>.<method> -- set a breakpoint in a method
stop at <class id>:<line>   -- set a breakpoint at a line
clear <class id>:<line>     -- clear a breakpoint
```

When you stop at breakpoint, you can examine the stack and variables (see below) and when you want to go on with further execution you use the step and continue commands:

```
step                         -- execute current line
cont                         -- continue execution from breakpoint
```

Examining the stack
To answer the question of "where are we in the program?", we use the `where`, `locals, up,` and `down` commands to examine the run-time stack. The run-time stack is like a "trail of breadcrumbs" in a program; each time a method call is made, a "crumb is dropped" (an RT stack frame is pushed). When a return from a method occurs, the corresponding RT stack frame is popped and discarded. These stack frames contain valuable information about where the method was called in the source code (line # and file name), what the parameters for the call were, etc.

`jdb` assigns numbers to stack frames counting from one for the innermost (currently executing) frame. At any time jdb identifies one frame as the "selected" frame. Variable lookups are done with respect to the selected frame. When the program being debugged stops (at a breakpoint), jdb selects the innermost frame. The `where` command prints a complete list of all stack frames,

starting from the inner and listing outward to the main. The commands below can be used to select other frames by number or address.

```
up [n frames]              -- move up a thread's stack
down [n frames]            -- move down a thread's stack
locals                     -- print all local variables in current stack frame
```

Examining source files

Another way to find our current location in the program and other useful information is to examine the relevant source files. `jdb` provides the following commands:

```
list [line number|method] -- print source code
use [source file path]    -- display or change the source path
```

Examining data

It is also useful to answer the question, "what are the values of the variables around us?" In order to do so, we use the following commands to examine variables:

```
print <id> [id(s)]         -- print object or field
dump <id> [id(s)]          -- print all object information
```

Dump is particularly handy since it will show all of the instance variables for an object in one listing, for example, inside a method you might want to "dump this" to see the entire state of the receiving object.

Debugging Strategy

If your program has been crashing spectacularly, you can just run the program by using the `run` command[4] right after you start the debugger. The debugger will catch the exception and allow you to examine the program (and hopefully find the cause and remedy).

More often the bug will be something more subtle. In these cases the “best” strategy is often to try to isolate the source of the bug, using breakpoints and checking the values of the program’s variables before setting the program in motion using `run`, `step`, or `cont`. A common technique for isolating bugs is to set a breakpoint at some point before the offending code and slowly continuing toward the crash site examining the state of the program along the way.

Printing Your Source Files

There's a really neat way to print out hardcopies of your source files. Use a command called "enscript". Commonly, it's used at the UNIX command line as follows:

```
enscript -2GrPsweet5 *.java
```

The above command would send all java source files to printer sweet5. You can change these parameters to fit your needs.

[4] If your application takes command line arguments include these on the same line as the ‘run’ command.

Appendix A: UNIX[5]/Emacs[6] Survival Guide

This handout summarizes many of the commands helpful for getting around on the Unix operating system and the Emacs editor. AIR provides nice thick readers on both of these topics at the LAIR and on the 2nd floor of Sweet Hall if you need more information.

Directory Commands

`cd` *directory*	Change directory. If *directory* is not specified, goes to home directory.
`pwd`	Show current directory (*p*rint *w*orking *d*irectory)
`ls`	Show the contents of a directory. **ls -a** will also show files whose name begins with a dot. **ls -l** shows lots of miscellaneous info about each file
`rm` *file*	Delete a file
`mv` *old new*	Rename a file from old to new (also works for moving things between directories). If there was already a file named new, its previous contents are lost.
`cp` *old new*	Creates a file named new containing the same thing as old. If there was already a file named new, its previous contents are lost.
`mkdir` *name*	Create a directory
`rmdir` *name*	Delete a directory. The directory must be empty.

Shorthand Notations & Wildcards

`.`	Current directory
`..`	Parent directory
`~`	Your home directory
`~<user>`	Home directory of *user*
`*`	Any number of characters (not '.') Ex: *.c is all files ending in '.c'
`?`	Any single character (not '.')

Miscellaneous Commands

`cat` *file*	Print the contents of *file* to standard output
`more` *file*	Same as `cat`, but only a page at a time (useful for displaying)
`less` *file*	Same as `more`, but with navigability (`less` is `more`)
`w`	Find out who is on the system and what they are doing
`ps`	List all your currently active processes
`jobs`	Show jobs that have been suspended
process`&`	Runs a process in the background
`%`	Continue last job suspended (suspend a process with ^Z)
`%` *number*	Continue a particular job
`kill` *process-id*	Kill a process
`kill -9` *process*	Kill a process with extreme prejudice

[5] UNIX* is a registered trademark of AT&T
*UNIX** is a registered trademark of AT&T
**UNIX† is a registered trademark of AT&T
†…

[6] GNU EMACS is provided for free by the FREE SOFTWARE FOUNDATION, which write software and gives it away because they think that's how software should be.

`grep` *exp files*	Search for an expression in a set of files
`wc` *file*	Count words, lines, and characters in a file
`script`	Start saving everything that happens in a file. type **exit** when done
`lpr` *file*	Print *file* to the default printer
`lpr -Pinky` *file*	Print *file* to the printer named *inky*
`diff` *file1 file2*	Show the differences between two files
`telnet` *hostname*	Log on to another machine
`webster` *word*	Looks up the given word in the dictionary. Works from most AIR machines, but is not standard UNIX (It actually talks to a NeXT...)

Getting Help

`man` *subject*	Read the manual entry on a particular subject
`man -k` *keyword*	Show all the manual listings for a particular keyword

History

`history`	Show the most recent commands executed
`!!`	Re-execute the last command
`!`*number*	Re-execute a particular command by number
`!`*string*	Re-execute the last command beginning with string
`^`*wrong*`^`*right*`^`	Re-execute the last command, substituting right for wrong
`Ctrl-P`	Scroll backwards through previous commands

Pipes

a `>` *b*	Redirect a's standard output to overwrite file b
a `>>` *b*	Redirect a's standard output to append to the file b
a `>&` *b*	Redirect a's error output to overwrite file b
a `<` *b*	Redirect a's standard input to read from the file b
a `|` *b*	Redirect a's standard output to b's standard input

GNU EMACS

For the following "^z" means hit the "z" key while holding down the "ctrl" key. "M-z" means hit the "z" key while hitting the "META" or after hitting the "ESC" key.

Running Emacs

`emacs <filename>`	Run `emacs` (on a particular file). Make sure you don't already have an `emacs` job running which you can just revive. Adding a '**&**' after the above command will run emacs in the background, freeing up your shell)
`^z`	Suspend `emacs`— revive with % command above
`^x^c`	Quit `emacs`
`^x^f`	Load a new file into emacs
`^x^v`	Load a new file into emacs and unload previous file
`^x^s`	Save the file
`^x-k`	Kill a buffer

Moving About

`^f`	Move forward one character
`^b`	Move backward one character
`^n`	Move to next line
`^p`	Move to previous line

(Note: the standard arrow keys also usually work.)

`^a`	Move to beginning of line
`^e`	Move to end of line
`^v`	Scroll down a page
`M-v`	Scroll up a page
`M-<`	Move to beginning of document
`^x-[`	Move to beginning of page
`M->`	Move to end of document
`^x-]`	Move to end of page
`^l`	Redraw screen centered at line under the cursor
`^x-o`	Move to other screen
`^x-b`	Switch to another buffer

Searching

`^s`	Search for a string
`^r`	Search for a string backwards from the cursor (quit both of these with `^f`)
`M-%`	Search-and-replace

Deletion

`^d`	Deletes letter under the cursor
`^k`	Kill from the cursor all the way to the right
`^y`	Yanks back all the last kills. Using the `^k ^y` combination you can get a cut-paste effect to move text around

Regions

Emacs defines a region as the space between the **mark** and the **point.** A mark is set with ^<spc> (control-spacebar). The point is at the cursor position.

M-w	Copy the region
^w	Kill the region. Using ^y will also yank back the last region killed or copied. This is what we used for "paste" back in the bad old mainframe days before there was "paste".

Screen Splitting

^x-2	Split screen horizontally
^x-3	Split screen vertically
^x-1	Make active window the only screen
^x-0	Make other window the only screen

Miscellaneous

M-$	Check spelling of word at the cursor
^g	In most contexts, cancel, stop, go back to normal command
M-x goto-line <#>	Goes to the given line number
^x-u	Undo
M-x shell	Start a shell within emacs

Compiling

M-x compile	Compile code in active window. Easiest if you have a makefile set up.
^c ^c	Do this with the cursor in the compile window, scrolls to the next compiler error. YEAH!

Getting Help

^h	Emacs help
^h t	Run the emacs tutorial

Emacs does command completion for you. Typing M-x <spc> will give you a list of emacs commands. There is also a man page on emacs. Type man emacs in a shell.

Appendix B: JDB command reference

```
threads [threadgroup]     -- list threads
thread <thread id>        -- set default thread
suspend [thread id(s)]    -- suspend threads (default: all)
resume [thread id(s)]     -- resume threads (default: all)
where [thread id] | all   -- dump a thread's stack
threadgroups              -- list threadgroups
threadgroup <name>        -- set current threadgroup

print <id> [id(s)]        -- print object or field
dump <id> [id(s)]         -- print all object information

locals                    -- print all local variables in current stack frame

classes                   -- list currently known classes
methods <class id>        -- list a class's methods
stop in <class id>.<method> -- set a breakpoint in a method
stop at <class id>:<line> -- set a breakpoint at a line
up [n frames]             -- move up a thread's stack
down [n frames]           -- move down a thread's stack
clear <class id>:<line>   -- clear a breakpoint
step                      -- execute current line
cont                      -- continue execution from breakpoint

catch <class id>          -- break for the specified exception
ignore <class id>         -- ignore when the specified exception

list [line number|method] -- print source code
use [source file path]    -- display or change the source path

memory                    -- report memory usage
gc                        -- free unused objects

load classname            -- load Java class to be debugged
run <class> [args]        -- start execution of a loaded Java class
!!                        -- repeat last command
help (or ?)               -- list commands
exit (or quit)            -- exit debugger
```

The following are examples of using some of the commands above:

```
> methods Player
void <init>(String, int)
boolean DoTurn()
ScoreBoard getScores()

> stop in Player.DoTurn
Breakpoint set in Player.DoTurn

> run
```