CS107 Handout 45
Autumn 2000 November 15, 2000

Sun's Java Development Kit

Written by Lisa Laane and revised by Jerry.

There are quite a few Java development environments to choose from, but most reliable one out there is Sun's JDK2 for Solaris. The JDK (short for Java Development Kit) package several command line tools—tools java, javadoc, and javac. Dealing with javac, the Java compiler, is much like dealing with gcc. Because javac and gcc are so similar, one is rarely surprised during Java compilation. Whatever surprises there might have been are made invisible by the make process, which works just as well on top of Java development as it does for C development. What follows is a brief but complete coverage of how to deal with the JDK while working on your Assignment 4 submission.

Well aware that there are several other Java platforms to choose from, I have no problem with your wanting to work on some other platform using some other suite of Java tools. After all, Java trumpets its compile once and run anywhere message all over Silicon Valley. True platform independence is not quite a reality though, so if you choose to work elsewhere but run into problems, you'll be expected to wise up and bring your work back home to the Suns and the JDK. Regardless of your success or failures on Macs and PCs, you'll ultimately need to port everything back to the Sun machines, test one final time, and then electronically submit as if you've worked at Sweet Hall all along. Please give yourself ample time for the port; you don't want to ftp everything over from your PC at 11:45 p.m. and bank on a smooth transition. My experience is that you should give yourself a good three hours to overcome porting issues. Most will need only a fraction of that much time, but if you do running into platform compatibility issues, chances are they alone with keep you busy for a good chunk of your evening.

In all honesty, it's generally just best to work on a Sun, either directly or remotely, to get most if not all of your work done. `xemacs` and `make` are your pals now, so there's very little reason to abandon them. The Solaris workstations we will use for Java are the `epics`, `elaines`, `sagas`, and `myths`, and the `fables`. (the same machines we used for concurrency).

Editing and Compiling

We recommend using `emacs` for editing again, since it provides automatic formatting for Java code. Editing your code is really nothing new. You will have many small class files, each named appropriately, all in the same directory.

This time, instead of using `gcc` as your compiler, you use `javac`. `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 in case of fatal compiler errors, let you know the errors you need to fix. In this respect, it is much like `gcc`.

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, you usually can just compile the file that contains your `main` method, since this file contains all the dependencies in your full program.

Actually, to make it even easier on you, we will give you a makefile with the assignment. All you need to do is add to the SOURCES line as you add classes. Thus you can compile your program the same way as before, by just issuing the `make` command.

Sometimes when you make changes to a parent class, it doesn't correctly detect the dependencies and fails to trigger a re-compile of the subclasses that also need to be updated. The easiest thing to do when things seem out of sync is to make clean to remove all compiled files and do a fresh build.

Important Note: Like all strict compilers**,** `javac` only allows one class per file and it the filename must be the same as the class name. For example, a class named `Shape` must be in a file called `Shape.java` (capitalization matters!) and it must be the only `public` class defined in that file. Some compilers are more lax and let you define several classes in the same file, but you won't get such sloppiness past Sun's compiler.

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 our handouts to learn about standard classes, sometimes it is 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 jdb debugger

If you thought `gdb` was a pain, `jdb` won't do much for impressing you. It's the same basic idea, but with much more minimal functionality. However, the good news is that Java is such a wonderful language that there is a lot less need for a debugger! There is a little on-line help for `jdb` available from the `help` command from within `jdb`.

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.

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
```

(note just as when invoking the interpreter you do not include the `.class` on the end of the name). 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 main function. Executing the `run` command at the `jdb` prompt will start your program running. 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 `stop` 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 runtime state

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, quite similarly to `gdb`. `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
```

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
```

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 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.