CS61B Lab #1

Since you've had only three mostly introductory lectures so far, today's lab is intended mostly to start getting acquainted with the tools that you'll be using, and to take care of various administrivia. Please do the activities below during lab.

1. Register

If you have not already done so, please use the "Account Administration" link on the class home page to register your particulars with our grading software.

2. Shells and Unix

Our official platform this year is Unix, with Emacs as the official editor and IDE (Interactive Development Environment). There is documentation for it available in our documentation reader, which is also available on-line from the "Emacs Quick Guide" and "Tools Documentation" links on the class home page.

In a terminal window, you can communicate with Unix through a "shell," basically a program intended to interpret text commands and print the resulting output. It is also possible to manipulate files and launch programs using menus and mouse motions. While these latter methods are more intuitive, they have their limitations, especially when it comes to scripting. The default shell is 'bash', which we assume when we specify shell commands. You can change your default shell (also your password) by logging in from a shell window to update.eecs.berkeley.edu.

In years past, we have used the Eclipse IDE. I have become rather disappointed with it: it has useful features buried in a poorly performing, unreliable, and sometimes quirky implementation, and the extra features it does provide are generally of little importance in this class. It is totally useless for remote use, requiring that you maintain your own version at home, which can give results incompatible with what we see (and grade on) on the lab computers. Nevertheless, feel free to use it (or other IDEs) if you prefer, as long as you make sure that whatever you hand in runs on OUR machines and environment.

2a. Review of some Unix Commands: The Hard Way

First, use the 'mkdir' command to create a directory for your assignments in your home directory:

```
... ~ $ mkdir work
```

The 'mkdir' command takes a single argument, the name of the directory to create. Make this new directory your current directory (the cd command), and create a directory named 'lab1':

```
... ~ $ cd work              # Now 'work' is the current directory.
... ~/work $ mkdir lab1
```

[We've only been showing the last part of the prompt here. Normally, it also tells you what machine you are on. In what follows, we'll abbreviate the prompts to just "$"]

Now use the 'cp' command to copy a file from the CS 61B code directory (~cs61b/code) to this new directory:

```
$ cp ~cs61b/code/lab1/YearCheck.stuff lab1
```

The 'ls' command should list the new file directory:

```
$ ls
... should list the current directory, including lab1 ...
$ ls lab1
... should list YearCheck.stuff ...
```

The 'rm' command can remove files permanently:

```
$ rm lab1/YearCheck.stuff
```

which gives us a chance to look at other ways of copying:

```
$ cp ~cs61b/code/lab1/YearCheck.stuff lab1/Yearcheck.stuff
# Creates the file lab1/Yearcheck.stuff (we've renamed the file
# this time, changing the capitalization).
```

Assuming this is a typo, we can correct it by renaming the new file:

```
$ mv lab1/Yearcheck.stuff lab1/YearCheck.stuff
```

We can remove lab1 and all of its files:

```
$ rm -r lab1
# The system prompts us about each file.
```

or

```
$ \rm -r lab1
# The system just deletes lab1 and YearCheck.stuff without
# prompting (be careful with this!)
```

After this, we have no lab1 directory, which gives us a chance to show one other way to copy a file or directory:

```
$ rsync -a ~cs61b/code/lab1 .
# Copy the directory lab1 and its contents to the current
# directory (aka, '.').
```

You can also persuade 'cp' to copy a directory, but rsync is rather more flexible (and allows copying between machines as well).

There are also visual (GUI) equivalents of these commands for Unix and other operating systems.

2b. Using the 'hw' Command

Section 2a talked about the "hard way" to set up a homework directory. Although the commands introduced there are useful, you'll probably want to use a rather different means for obtaining and managing your homework, project, and lab skeletons. Let's first remove the lab1 directory from 2a and start over:

```
$ cd ~/work
$ \rm -r lab1
```

In real life, one usually manages source code with the help of a "version-control system" (or "source-code control system.") These programs allow you to permanently archive periodic snapshots of the contents of all your working files and retrieve them as needed, to compare files with archived versions to see what changes you've made, to transfer files from one site to another and to update files at one site with archived changes from another, and to "tag"---attach labels to selected snapshots.

In this class, we use the Subversion version-control system for these purposes. You can use it to fetch homework, lab, and project

skeletons, save your work as often as desired (thus protecting yourself from disasters such as accidentally erasing your files), and to turn in assignments. Bare Subversion is complex, and has proved rather error-prone for CS61B students, so I have written a script for dealing with it, called 'hw'. This week's entry on the Homework web page contains a pointer to more documentation.

The 'hw' command allows you to start work on a new assignment (such as this lab) with a command sequence like

```
$ cd ~/work    # If not already there
$ hw init lab1
$ cd lab1
$ ... edit, compile, etc.
```

After working for awhile in directory work/lab1, you can then back up your work at any time (we suggest frequently) with the command

```
$ hw commit
```

and to submit your assignment, the command

```
$ hw submit
```

(which will also do a 'hw commit' if you haven't done so recently).

2c. Starting Emacs

It is likely that you used a terminal window to do part 2a-b. I don't usually recommend that. A decent IDE will provide a better base of operations. Personally, I generally start up Emacs and leave it only when I need to use the browser.

Start up Emacs using a pop-up menu or a terminal window (the command line would be

```
$ emacs &
```

). Create a shell window and buffer in Emacs with the command 'M-x shell' (that's "meta-x <space> shell <enter>", pressing 'x' while holding down the 'Meta' key. Depending on how you are using Emacs, you may not have a meta key, in which case use '<ESC> x' in place of 'M-x'.)

In the shell window, you can execute commands just as before. All the text you type, and the system's responses, go into the shell buffer, where you can edit them as you can any other text. There's no need to keep a transcript: the buffer IS automatically the transcript.

3. Creating and Running a Program

In your lab1 directory, you should have a file YearCheck.java that was included when you copied over the code/lab1 directory above. You'll see that this file is essentially a skeleton containing a main program and a single method, whose body you are to supply using lines from YearCheck.stuff.

First, let's do things "by hand". Create a shell buffer in Emacs (if you haven't already) and go to that window. Use the 'cd' command to change to your lab1 directory, if needed. Then issue the Unix command

```
javac -g YearCheck.java
```

in that shell window. This should compile your program without errors. Test it with the command

```
java YearCheck 1900 1901
```

You should get two correct statements.

Now let's introduce a little automation into the process. You may have noticed that we included a file called 'Makefile' in the lab1 directory that you copied. Take a look inside and see if you can relate what happens next to the contents of that directory.

First, issue the command

```
gmake clean
```

Your YearCheck.class file should disappear. Restore it with the command

```
gmake
```

which takes the first action in your Makefile (called 'default' by convention). As is customary, this default action is to rebuild the program in your day1 directory ('gmake compile' and 'gmake YearCheck.class' do the same thing). Now issue the 'gmake' command again, and you'll see the program tell you that there's nothing to do (YearCheck.class is up to date). Finally, issue the command

```
gmake check
```

which, by convention, runs various tests on the program in the directory.

Try this same sequence ('gmake clean', 'gmake', 'gmake check') using Emacs's compilation commands. In the YearCheck.java buffer, issue the command 'M-x compile' or 'C-x C-e' (Control-x Control-e). It will prompt you with a command to execute, which you can edit before hitting "enter". You should be able to give the same sequence of commands to 'gmake' as before; each will execute in a buffer called "*compilation*". An advantage of compiling this way is that Emacs will prompt you to save any files that you have modified but not yet saved. Be sure to do so, since otherwise the compiler will process out-of-date information.

The current tests in Makefile (under "check") are rather wimpy. Add tests to the Makefile until you see erroneous results.

4. Editing and Debugging a Program

The skeleton YearCheck.java file that we provide is wrong, as your revisions to the 'gmake check' target should now demonstrate. The lines of YearCheck.stuff, when arranged in the correct order (possibly with duplicates of some lines), will provide a correct body for the isLeapYear method, which determines if the value stored in the variable <tt>year</tt> is a legal leap year (divisible by 400, or divisible by 4 and not by 100). Your task is to figure out the necessary selection of lines to make the program work. It doesn't matter that you don't know Java; your native intelligence should allow you to make educated guesses and eventually come up with a solution.

4.1 Compilation Errors

First, however, let's see what to expect when things go wrong. Modify YearCheck.java to introduce a compile-time error. For example, remove the 'return false' line at the end of isLeapYear and recompile (be sure to save the modified file, as 'M-x compile' will prompt you to do.) The '*compilation*' buffer will show the error, and the Emacs command "C-x '" (control-x backquote) will sequence you through the erroneous lines in your file, positioning Emacs at each one in turn (there's only one in this case.) Correct the error, and try 'gmake check'.

The line on which the error occurs is not always the one that the Java compiler fingers. Try removing the keyword ‘static’ from the declaration of isLeapYear and recompile. The error message will be on the CALL to isLeapYear, not the definition. After seeing this, restore the missing keyword.

4.2 Runtime errors

Now let’s introduce a run-time error (one that is not observed until the program begins execution). Change the second ‘printf’ call to read

```
         System.out.printf("%d is not a leap year.%n");
```

and run ‘gmake check’ again. This time, you should see a message that begins "Exception in thread ...", which is the sort of thing the Java interpreter emits when it encounters a run-time error. If you examine the message in detail, you’ll see that it names a sequence of methods and line numbers within them. The first line:

```
        at java.util.Formatter.format(Formatter.java:2432)
```

Tells us that the error actually got detected at line #2432 of a file Formatter.java (part of the standard Java library) in a method named ’format’. The next line:

```
        at java.io.PrintStream.format(PrintStream.java:920)
```

tells us that ’java.util.Formatter.format’ was called from another method named ’format’, this one in PrintStream.java. The last line:

```
        at YearCheck.main(YearCheck.java:16)
```

tells us that this whole debacle started with a call at line 16 of YearCheck.java, which happens to be the one we changed. This series of "at ..." lines is called a "stack trace."

Of course, this particular error would be easy to figure out just by looking at the offending line in your code. However, sometimes it is useful to take a closer look, for which purpose one typically uses a "debugger". Our official debugger this year is GJDB (a Java debugger resembling GDB adapted by yours truly from Sun’s original jdb debugger). Let’s try it out.

In the YearCheck.java buffer, issue the command "M-x gjdb" and modify the prompt in the minibuffer (the line at the bottom of the Emacs window) to read "gjdb YearCheck". Alternatively, use the menu entry Debug->Start Debugger. This gives you buffer named "*gud-YearCheck*", which should contain a prompt "[-]". Type the command "run 1900 1901" at the prompt. You should see the message

```
   "Exception occurred: java.util.MissingFormatArgumentException ...."
```

The command ‘where’ will give you a stack trace at this point, starting with java.io.PrintStream.format (this is just because gjdb doesn’t treat the method that detects and ’raises’ an exception as the error, but rather the method that calls it without being prepared to do something about it). You’ll find that the commands ‘up’ and ‘down’ (or the keys F3 and F4, respectively) move you up and down the stack trace, positioning Emacs at the designated lines (actually, you’ll never see lines in the standard library, but lines in your own code do get displayed). By this means, you can always find where an error occurs and how you got there.

4.3 Examining Program State

Using the ‘up’ and ‘down’ commands (or F3 and F4), position GJDB to the printf command that caused the runtime error from 4.2. In the "*gud..." buffer, enter the following command at the "main[2]" prompt"

```
    main[2] p year
```

GJDB will print the value of variable ‘year’ in response. Now try

```
    main[2] p args
```

You should get a response something like this:

```
    $2 = instance of java.lang.String[2] (id=69)
```

telling you that ‘args’ has a 2-element array of Strings as its value. To see what these are, type

```
    main[2] p/1 args
```

which should print the contents of the array as well (‘args’ is a pointer to an array; the ‘/1’ suffix means "follow any pointers for one step before printing the result". In fact, you can print the result of evaluating many Java expressions. Try

```
    main[2] p args[0]
    main[2] p args.length
    main[2] p args[0]+args[1]
    main[2] p year * 3 - 1
```

Finally, the command

```
    main[2] info locals
```

will print out all the "local variables" (those defined in the method that GJDB is currently looking at), including a number of "hidden" variables that the system adds for its own purposes.

4.3 Controlling Execution

Correct the problem you introduced in section 4.2 and recompile the result. Next, move the Emacs cursor to the line that reads

```
            if (isLeapYear(year))
```

and type C-x <SPACE> (a Control-x followed by a blank), or the ’Debug->Set Breakpoint’ menu command. Go to the GJDB buffer ("*gud-YearCheck*"), and at the prompt, enter the plain command ’run’. GJDB will ask if you want to terminate the current program (which hasn’t finished yet); answer ’y’. The program will run and stop on the line you marked. At this point, you can step through the program while in the GJDB buffer by using the ’step’ (or just ’s’) command or the F5 key.

Issue the ’run’ command again. This time, step through the program using the ’next’ (’n’) command or the F6 key. You’ll see that this time, you don’t step into the isLeapYear method, but instead "step over" each call to it.

Again issue the ’run’ command, followed by ’next’. Now enter the ’continue’ (or ’cont’) or F8 key. The program will start running again until it hits the breakpoint again.

There is one final command we haven’t talked about: the ’finish’ command or F7 key. This causes the program to continue until the current method exits. When you’ve added some lines to isLeapYear, you might try it to see how it works.

4.4 Finish YearCheck

Using the lines from YearCheck.stuff, finish the isLeapYear program. The raw lines in YearCheck.stuff are unindented. As a stylistic matter, you should indent to indicate the structure of your program. Your Emacs setup can apply a number of indentation rules that we'll be using throughout this course. To correct the indentation of one or more lines, you can either position the Emacs cursor at the beginning of a line and press the tab key or you can select a region of text by dragging the mouse with button-1 pressed and then use the menu item 'Java->Indent Line or Region', the command 'M-x indent-region' command, or C-M-\ (Control-Meta-backslash) key.

5. Improving Testing

Testing is an important component of any programming task. At the moment, your tests do nothing but run the program and print the results. Presumably, you are to examine them each time you do 'gmake check' to make sure they're right. This is better than entering test sets by hand from a terminal, but with any sufficient set of tests for a substantial program, it is a tedious and error-prone procedure. It's much better to have 'gmake check' confirm the results as well. There are many approaches to this problem. Let's start with a simple manual procedure, based on Unix utilities.

In your Makefile, you'll find an "Alternative test procedure" that has been commented out with "#" characters. Remove those "#" marks and place them instead in front of the lines of the current "check" lines. Now run 'gmake check'. The first command,

```
sh -ve tests.cmd > tests.out
```

means "Read the contents of the file tests.cmd and execute as commands to the shell, printing each command and stopping at the first error. Put the output in tests.out." The second command,

```
diff -b tests.std tests.out
```

compares the output file just produced with a file 'tests.std', which contains the proper output for the commands in tests.cmd. If either of these commands fails, 'gmake check' will give an error message; otherwise, it will terminate normally. The -b switch just means not to be fussy about matching spaces: ignore spaces at the end of the line, and pretend all other sequences of blanks and tabs are just one blank. It's not always appropriate, but probably makes sense in this case.

Add more tests to tests.cmd and corresponding lines in tests.std so as to make a reasonable test suite. Throw in a command such as

```
'java YearCheck GLORP'
```

to see what happens when a command fails, and also see what happens when the output doesn't match. Be sure to set everything right when you finish fiddling around, however.

6. Style

Any real-life project that involves more than one person will establish various conventions for how programs are to look, how names are to be chosen, what language features are to be avoided, etc. The idea is to make our work mutually intelligible; we give up some freedom of expression so as to eliminate unnecessary barriers to modifying, checking, or using each others' work. Some stylistic guidelines are subjective, but others are subject to mechanical verification. To get you acquainted with the whole concept (as well as to keep us from getting severe headaches when reading your programs), our software will be enforcing a number of style guidelines in all your submitted Java programs. For a complete discussion, see the "style61b program" link on the "Labs and Homework" page.

In your lab1 directory, check your YearCheck.java file with the command:

```
$ style61b YearCheck.java
```

It will report a few problems with "magic numbers", which you should understand but ignore for this lab, and might report other things. We have also set up Makefile so that the command

```
$ gmake style
```

executes this command without reporting magic numbers. Fix any errors it reports (you won't be able to hand this file in until you do!).