INTRODUCTION TO COMPUTER SCIENCE I
LAB 1
Arithmetic and Basic Input/Output

For our first project, you will get familiar with the tools used to *write*, *compile*, and *execute* programs.[1] You will do so by writing small programs that read numbers that the user types with the keyboard, performs arithmetic on those numbers, and prints a result to the screen. Along the way, you may have to do a bit of scurrying about campus...

1 Your first Java program: Printing text messages

1.1 Getting started in the lab

The following steps will lead you through your first Java program. Here, the goal is to get used to the tools involved in writing, compiling, executing, and debugging these programs. After you get this pre-written and simple program working, you will then need to write a program of your own.

1. **Login to your workstation:** Our lab is full of basic Windows desktop computers. These are run by our Information Technology department, and you must begin by logging into them using your college username and password.

2. **Login to a server:** The computer systems that we will use for our projects are `romulus.amherst.edu` or `remus.amherst.edu`, (heretofore, `remus/romulus`), which are UNIX (Linux) systems. To use these systems, you must login to them from your workstation using *Remote Desktop*, software that allows you to connect graphically to these servers. To do so, follow Remote Desktop Connection instructions that describe not only how to use this software on the Windows machines in Seeley Mudd 014, but also how to install (if needed) and use this software on your own computer, whether Windows, Mac, or Linux.

   Once you have logged into `remus/romulus`, you will see the graphical interface for that server, which looks very much like the desktop that your own computer presents. Right-click anywhere in the empty part of the desktop (not on an icon), and a menu will pop up. In that menu, select *Open in Terminal*. You will then see a *terminal window* within which there is a *shell*—a prompt at which you can type commands to the system. The shell is the place from which you will direct the system to run the program that allows you to edit your source code, to perform the compilation of your source code, and to execute your programs.

3. **Make a directory:** When you first login, you will be working in your *home directory*—the UNIX analog of your *My Documents* folder. Within this directory, you should make a *subdirectory* (a folder) for your work for this lab by using the `mkdir` (*make directory*) command (shown below). Once you do that, you can `cd` (*change directory*) into that new folder. Using those commands looks like the following, where the dollar sign (`$`) is the *prompt*—what the shell prints before stopping to wait for you to type a command:

[1] If you don't understand these terms—particularly *compile* and *execute*—fear not! Such terms will be defined and used in examples soon enough. In the meantime, the point is that you will write programs and then attempt to make those programs "go".

```
$ mkdir lab-1
$ cd lab-1
```

4. **Get some sample source code:** Use the following command to obtain a sample Java source code file, being careful to include the tilde (˜) before my username and the trailing space followed by a period (.):

   ```
   $ cp ˜sfkaplan/public/COSC-111/lab-1/Howdy.java .
   ```

   To ensure that you have copied the file into your `lab-1` subdirectory, use the `ls` (*list directory*) command to list the files in the current directory, noting that the character following the dash (`-`) is a lowercase letter `L`, and **not** the numeral `1`, making the `-l` part of the command mean, *list directory using the long format*:

   ```
   $ ls -l
   ```

   You should see an output that looks something like this:

   ```
   total 4
   -rw-r--r-- 1 sfkaplan sfkaplan 235 Jan 28 22:18 Howdy.java
   ```

5. **Examine and modify the source code:** Run *Emacs*, a programming text editor, to examine the `Howdy.java` file. In the following command, be sure to include the trailing ampersand (`&`), causing the text editor to run in the *background*—that is, to run while allowing you to enter more commands:

   ```
   $ emacs Howdy.java &
   ```

   You will see, in the *Emacs* window that appears, the source code for `Howdy.java`, which is a small program much like the one we wrote on the blackboard in class. In fact, this program is simpler: it declares no variables and performs no arithmetic. Instead, it merely prints a message to the screen.

   You will find that, within the *Emacs* window, you can move around the source code with the arrow keys, and change the file simply by typing in a normal fashion. The pull-down menus allow you to save your file periodically and to exit the program. However, *Emacs* is a complex program that is capable of a great deal more. To really start learning how to use it, you should read this documentation/tutorial on using Emacs.

   Once you have gotten somewhat comfortable with your new text editor, use it to **add one more line of text to what is printed on the screen.** It doesn't matter what text you add—just have the program print something new and unique. Once you are done adding this additional line of code, be sure to **use the *save* command**.

6. **Compile:** Now that you have changed the source code, you must translate it into a form that the computer can execute. Leaving your *Emacs* window open, click over to your terminal window again. In it, use the following command to compile your source code:

   ```
   $ javac Howdy.java
   ```

   In this case, **no news is good news**. That is, if the computer simply presents the shell prompt to you after you issue this command, then **the compilation succeeded.** The compiler—the `javac` program—will print messages into your terminal window only if it was unable to translate your program.

   If you see such an error message, then you must have made some type of mistake in adding your line of code to print one more line of text. Go back to your *Emacs* window and see if you can spot your error. If you can, correct it, save the source code, go back to your shell window, and issue the compilation command (as above) again. If the error persists, or if you could not see what your error was in the first place, then **ask for help**.

7. **Execute:** Once you have successfully compiled your program, it is time to run it and see what happens. Go to your shell window and issue this command:

   ```
   $ java Howdy
   ```

   Your program should (very quickly) print into your shell window the lines of text that your source code indicated it should. If you don't see the text that you expected, then go back to your source code in your *Emacs* window, and see if you can spot your error. If you cannot find the error, then **ask for help.**

Congratulations! You've (partially) written, compiled, and run a Java program! Although the programs will get more complex, you will continue to use the *write, compile, execute* sequence throughout. You can now close your *Emacs* window since you are done with this program.

1.2 Continuing your work after lab

When you finish your lab section, at which time you close your windows and log out, all of your work is saved on the servers (`remus` and `romulus`). Therefore, no matter how and from whence you connect to those servers, your files will be waiting for you, as you last left them.

Connecting from outside the lab: In order to continue your work, you must connect again to one of the servers (`remus` or `romulus`). Install and use *Remote Desktop*, just as you did in the lab, to connect from wherever you are. Note that if you are not on the Amherst College campus network, you will need to use a VPN (*virtual private network*) connection for *Remote Desktop* to work.

Finding and opening your work: Once you are connected to one of the servers, you need only to change into the appropriate directory and open your source code within that directory. It may look something like this:

```
[sfkaplan@remus ~]$ cd lab-1
[sfkaplan@remus ~/lab-1]$ ls -l

-rw-r----- 1 sfkaplan sfkaplan  500 Feb  7 16:55 Howdy.class
-rw-r----- 1 sfkaplan sfkaplan  423 Feb  7 16:55 Howdy.java
-rw-r----- 1 sfkaplan sfkaplan  415 Feb  7 16:55 Howdy.java~

[sfkaplan@remus ~/lab-1]$ emacs Howdy.java &
```

You will see the source code files (which end with `.java`) are just where you left them, ready to be opened with *Emacs*. Moreover, the files that have a tilde (~) appended are *auto-saved* copies that are the result of *Emacs* saving a backup of your work every couple of minutes. Finally, the compiled files with the suffix `.class` are the result of your use of the `javac` command; that is, the translation from your source code into a format that the computer can run is stored in the `.class` files.

At this point, you may resume your work from the point at which you left off. The `emacs` command opens the source code for editing; you may use the `javac` command to compile (i.e., translate) your source code; and you may use the `java` command to run your compiled programs.

Mistakes to avoid: There are a number of ways to make mistakes in using these servers, but there are two particular steps that you should try not to commit.

1. **Don't recreate the directory:** When you began the lab, you used the `mkdir` command to create the `lab-1` directory. Having created that directory, **you should not do so again**. The `mkdir` command should be used to **mak**e the **dir**ectory exactly once per project.

2. **Don't re-copy the starting source code files:** Again, when you began the lab, you used the `cp` command to **cop**y source code (i.e., `.java`) files from a directory of mine into your project directory. **You should copy those files only once.** If you re-copy them a second time, those copies will overwrite the work that you've done, eliminating your work from the source code files.[2]

2 Your second program: User input and arithmetic

You are now going to write a program that reads a few values that the user of the program types in, performs a few arithmetic operations on those values, and then prints the results to the screen. This program will seem almost absurdly arbitrary—and in some sense, it is—but the purpose of this program will become clear later.

[2]If you make this mistake and lose significant work, contact me. The IT department performs tape-based back-ups of the files on the servers frequently. Although it may take a day, I can request that copies of your files from before the mistaken use of `cp` be restored, thus recovering your lost work.

2.1 Getting started

Because this program will do a few new things that we have not yet discussed in class, I am providing some portions of the program. Much like your `Howdy` program, you will add the critical, arithmetic instructions to the program, making it whole. You should begin by obtaining the initial, partially written program by issuing the following command at your shell prompt, again being sure to put the *tilde* (`~`) and the trailing *space* ( ) and *period* (`.`) in proper places as shown here:

```
$ cp ~sfkaplan/public/COSC-111/lab-1/StrangeMath.java .
```

Once again, use *Emacs* to examine and modify the source code of this program:

```
$ emacs StrangeMath.java &
```

2.2 Understanding the user input code

You will quickly notice that there are unfamiliar lines within the `StrangeMath` source code. Specifically, at and near the top are the lines:

```
import java.util.Scanner;
[...]
  public static Scanner keyboard = new Scanner(System.in);
```

Once again, I will make like the Wizard of Oz and ask that you "not look behind the curtain"—that is, don't ask what these magic lines, this "goop", mean. These are, for the moment, lines that are simply necessary for allowing the user of your program (usually, *you*) to type in numbers while the program runs that the program can then use. To that end, notice the pairs of lines that look something like:

```
System.out.print("Enter a value for a: ");
int a = keyboard.nextInt();
```

The first of these two lines does something familiar: it prints a line of text to the window. In this case, that text is a *prompt*, asking the user to enter a datum. The second line, however, is less familiar. Declaring an integer variable named `a` is something we know how to do, and so is assigning a value into that space. What is new, however, is the expression, `keyboard.nextInt()`. Simply put, this command causes the program to wait for the user to **type an integer value and press the *return* key**. When the user does so, the integer value is assigned into the space named `a`.[3]

[3] *What happens if the user doesn't enter an integer?* Short answer: Try it and find out! Long answer: The program will *crash*—that is, it will abruptly stop running, but not before printing a strange collection of currently indecipherable (to us) error messages. We will learn how to read such crash messages later. And even later than that, we will learn how to keep the program from crashing when the user types the wrong kind of input. For now, don't worry about those things.

2.3 Your task: Adding the arithmetic

The user must enter three integer values, namely: `a`; `b`; and `c`. It is your task to then compute three new values, each of which depends on some subset of `a`, `b`, and `c`. Specifically, you must compute `x`, `y`, and `z`, noting that **all of these values are integers, and all computations should be done with integer arithmetic:**

$$x = (b \bmod a) + 12$$

$$y = \frac{b}{a}$$

$$z = \frac{ab}{10c} - 1$$

Why these wacky arithmetic operations? These will serve as your "magic decoder ring" for the wild goose chase, below ...[4]

Notice that the final part of the program prints the values of variables `x`, `y`, and `z` to the terminal window. Therefore, the code that you add to the program must **declare** and **assign** these variables their correct values, as described above.

2.4 Testing your program

Once you have added the lines of code that perform the strange arithmetic, you should test that your program works! Specifically, these are three arithmetic operations that you could perform with pen and paper or, for those so inclined, with a calculator.[5] Therefore, you should dream up a handful of values for `a`, `b`, and `c`. Before running your program, calculate for yourself what `x`, `y`, and `z` **should** be if your program is written correctly.

Armed with a few *test cases*, now run your program:

```
$ java StrangeMath
```

When prompted by your program to enter values for `a`, `b`, and `c`, choose any one of your predetermined trio of values for those variables. Then examine your program's output. Did it produce the values for `x`, `y`, and `z` that you expected? If not, then either your program or your test case contains an error, and you must determine which is at fault and fix it. If the output **does** match your expectation, then you have one (more) test case to support your belief that your program is correct.[6] Once your program has passed enough test cases to convince you that it is likely to be working correctly, then you should move on to ...

[4]Put differently, these arithmetic operations were chosen so that, if you find the right values for a, b, and c, you will then calculate the useful-but-only-superficially-meaningful x, y, and z to find something. Yes, it's contrived. Get over it.

[5]Don't forget that you're using **integer arithmetic!**

[6]Do not confuse this belief as being **proof** that your program is correct. Proving that program produces correct output in all cases is exceedingly difficult, and way outside of the scope of this course.

3 The wild goose chase, take I

Have you even been to the gym? Have you noticed, on the walls surrounding the main, old basketball court (**not** LeFrak), the pictures of so many alumni who have competed on various teams, going back over 100 years? Your mission, should you choose to accept it, is to find **one particular person in one particular such photograph.**[7]

3.1 Finding the inputs

To find this photograph and the person in it, you must find three very important numbers. Finding them will require a bit of patience, a stunning lack of ingenuity, and, one hopes, a sunny disposition. Here are the clues—none too subtle—for finding those three numbers:

`a`: This value is *the numeric portion of the street address of the Folger Shakespeare Library*. Truly low cunning is required to discover this value. Should you require more than two minutes for this task, hang your head in shame and avoid eye contact. *Point of curiosity:* Why might I have involved the poor Folger in this fiasco?

`b`: In *Frost Library*, there is an old dictionary sitting upon a reading pedestal. You need to find the number of the page on which the word "fantod" is defined in this *Webster's Third New International Dictionary*.[8]

`c`: *Something there is that doesn't love a wall.*
Which class year of Amherst alumns donated a statue of this quote's author to the college?

3.2 Using the outputs

This next step should not surprise you. Go run your `StrangeMath` program, and enter the values of `a`, `b`, and `c` that you worked so hard to obtain. From it, you will, of course, obtain values for `x`, `y`, and `z`. These are the values you need to find the person among the pictures of alumni athletes in the gym. To whit, follow these steps to find the person in question:

1. Go to the gym.[9] Go to the *hallway on the north side of the basketball court*.[10] Then, look at the *north wall of that hallway*—that is, with your back to the basketball court itself.
2. Starting from the far left side of this wall, find the x^{th} column of photographs.
3. From the bottom of that column, find the photograph in the y^{th} row. **Note the team and year of this photograph.**
4. Within the photograph, find the middle row of people. On its far left is the (rather aged, at that time) head coach.

[7] No, I am not kidding.

[8] Yup, another *interesting question*: What author, also an alumnus, would get the howling fantods if he were alive to see my mimicry of his heavy use of footnotes?

[9] Duh.

[10] Don't know which way is north? Seriously, you can't figure that out?

5. From the left side of that row of people, find the z^{th} person. **Note the exact name given for this person in the photograph.**

Recording your big find: Login to `remus/romulus`. Within your `lab-1` subdirectory, use *Emacs* to open a plain text file, like so:

```
  $ emacs final-answer.txt &
```

Into this file, type the two pieces of information that you noted from alumni photograph: the team/year of the photograph, and name of the person. When you have entered these data, save the file and then close your *Emacs* window.

4 How to submit your work

Go to the Amherst CS Homework Submission System (henceforth, just the *Submission System*). Login, choose the course (`COSC 111, 1:00 lectures`), and then choose `Lab-1, First programs` as the assignment for which you want to submit work.

You will then see three separate `Choose File` buttons. There is one each for the three files you've worked with on this assignment: `Howdy.java`, `StrangeMath.java`, and `final-answers.txt`, all clearly labeled. Click each and select your copy of the given file. Finally, click `Submit files` to have all of your work submitted.

Notice that you may, later, submit updated or corrected versions of your work. Each submission is kept separately, and each is marked with the time submitted. So feel free to submit updated work if needed.

This assignment is due on Tuesday, Feb-02, 11:59 pm, before it becomes Wednesday, Feb-03.