INTRODUCTION TO COMPUTER SCIENCE I

PROJECT 1
Arithmetic and Basic Input/Output

For our first project, you will get familiar with the tools used to write, compile, and execute programs. 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

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:** 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 *Xming*, software that allows you to connect graphically to these servers. To do so, follow the Windows *Xming* instructions that describe how to use this software on the Windows machines in Seeley Mudd 014. Notice that this page also describes how to install and use *Xming* on your own computer if it is a Windows machine as well. If you have a Mac, follow the Mac *X11* and *ssh* instructions.

   Once you have logged into `remus/romulus`, you will be presented with a *shell*—a text window with 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. Specifically, enter the following command to create and then change into that subdirectory:

   ```
   $ mkdir project-1
   $ cd project-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/cs11/project-1/Howdy.java .
```

To ensure that you have copied the file into your `project-1` subdirectory, use the following 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`:

```
$ 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 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 easily 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 shell 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 shell 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 don't see 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.

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 its purpose of this program will become clear later.

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 this 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/cs11/project-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—at least, not yet. 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`.[1]

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:**

$$x = b \bmod a$$
$$y = \frac{ac}{b} - 6$$
$$z = \frac{b}{a}$$

Why these wacky arithmetic operations? These will serve as your "magic decoder ring" for the wild goose chase, below . . .[2]

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

[1] *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.

[2] Be afraid, be very afraid.

2.4 Testing your program

Once you have added the lines of code that perform the strangely needed 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.[3] 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 programs 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 a 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.[4] 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 ...

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,[5] is to find **one particular person in one particular such photograph.**[6] Moreover, **it's a race**, where those who submit a correct answer sooner get more credit than those who do so later.[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, frighteningly little 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 and avoid eye contact. *Bonus point:* Why might I have involved the poor Folger in this fiasco?

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

[4] 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.

[5] I highly recommend that you **do** accept it.

[6] No, I am not kidding.

[7] I'm still not kidding. And don't freak out about the credit thing. It's not like you're going to get terrible grades for slow but correct solutions to these projects. The race is a small component of the grading. Now get to work.

`b`: In the *Olds Mathematics Library/Reading Room*,[8] there is posted, upon the wall, a collection of prime numbers. The value you seek is *the number of primes listed in this collection.*

`c`: On level 3 of the Merrill Science Center, there are posters, each presenting scientific work carried out using the slave labor of an undergraduate student who toiled in a lab, utterly wasting one of the beautiful summers of that student's youth. Among them you will find one that addresses the important but not terribly uplifting topic of female college students, dieting, self-esteem, and body image.[9] The value you seek is N*, as shown in Table 1 of the poster*—where N is the number of subjects measured for the study.

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.[10] Go to the *hallway on the north side of the basketball court*.[11] 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 top 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.
5. From the left side of that row of people, find the z^{th} person. **Note the exact name of this person in the photograph.**

Recording your big find: Run, don't walk, to a computer from which you can login to `remus/romulus`. Within your `project-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.[12] Save the file and then close your *Emacs* window.

[8] Don't know where that is? Use The Google, Luke.

[9] Why on earth did I pick this particular poster? Because I'm on a diet, and it caught my attention. Now bring me a slice of Antonio's.

[10] Duh.

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

[12] **Super-mega-bonus points:** Find the two other photographs in that gym in which this same person appears.

4 How to submit your work

You will use the `cs11-submit` command to turn in your work. Specifically, you should submit your completed `Howdy.java`, `StrangeMath.java`, and `final-answer.txt` files, like so:

```
cs11-submit project-1 Howdy.java StrangeMath.java final-answer.txt
```

This assignment is due on Thursday, February 4, at 11:59 pm