CSCI 1933 Lab 2

Basic Java Programming

Rules

For this lab you will work in pairs. We suggest that you have a TA evaluate your progress on each step before you move on the next. The labs should be completed primarily in the lab time; however, we require that at least the starred steps (this week, steps 1, 2 and 3) be completed and checked off by a TA during lab time. Starred steps ★ cannot be checked off during office hours. You have until the end of office hours on monday to have non-starred steps checked off, but we suggest you have them checked by Friday to avoid crowded office hours on Monday. There is nothing to submit via Moodle for this lab.

Command Line Input

Some lab steps this week will make use of command line inputs. Command line inputs are the arguments listed after the command line to run your Java program. Command line arguments that are interpreted by your Java program as Strings can be supplied as follows:

```
java MyProg argument1 argument2 argument3 ...
```

For example, if you entered the following into your program:

```
java MyProg 1 2 3 a b c
```

The array `args` from your main method would look like `["1", "2", "3", "a", "b", "c"]`. The values can be obtained by referencing args[0], args[1], args[2]..., respectively. Since all are Strings, if a scalar integer or double equivalent is needed for one of the arguments, it can be derived by the following:

`Integer.valueOf(args[0]);` or `Double.valueOf(args[0]);`

In order to process all the arguments typed, the length of the array of String structure, `args`, can be found by `args.length`. This is shown in the simple loop to process all the command line values in the H3.java example from lecture during the first week.

1 More Testing ★

Write tests for the Complex3 class definition discussed in lecture using the expected value - actual value method of testing. For example if you created a method `int mult(int a, int b)`, to test if it worked you would create a variable `actual` and a variable `expected` and compare them like so:

```
int actual = mult(2, 3);
int expected = 6;
boolean result = expected == actual;
System.out.println("TestMult: " + result);
```

Doing this allows for quicker evaluation of the tests, since you don't need to inspect the results any further than looking for any test that returned `false`.

2 Reading from the Command Line ★

Using the examples from the previous lab (Lab 1) and lecture as resources, write a Java program to average a list of numbers supplied on the command line. It should function as follows:

```
java Av 1 2 3 4 5 6
Average of 6 value(s) entered is: 3.5

java Av
No arguments entered

java Av 5
Average of 1 value(s) entered is: 5.0
```

Test your code thoroughly. When does it work? When does it fail?

3 Thinking Ahead ★

While you do NOT need to fix the problem(s) in your code from Step 2 above now, can you identify the main problem present in the code as written? Try to think of *two* possible approaches to eliminating the error messages that result. Again, you do not need to implement them, but later on in the course you can re-visit these solutions and implement them.

Explain the main problem and your proposed solutions to a TA.

4 What's my GPA?

Quick introduction to `Scanner`

`java.util.Scanner` is a built-in class for basic (i.e. non-regex) parsing of `String` objects. We can use it to extract tokens from a `String` – a token is simply anything that is surrounded by whitespace. For example, in the string `"Hello 1933 Class!"`, three tokens exist: `"Hello"`, `"1933"`, and `"Class!"`.

`Scanner` has a variety of methods we will find useful when trying to calculate a GPA. Of particular importance to us are `hasNext()`, `next()`, `hasNextInt()`, and `nextInt()`. The method `hasNext()` returns the boolean value `true` if there is another token to match, and `next()` will return that token as a `String`. The methods `hasNextInt()` and `nextInt()` work similarly, but expect the tokens to be integers (which are stored in strings).

A snippet of code working with `Scanner` follows.

```
String inputString = "Hello 1933 Class!";
Scanner scanner = new Scanner(inputString);

String firstWord = scanner.next();
int number = scanner.nextInt();
String secondWord = scanner.next();

if (scanner.hasNext()) {
    System.out.println("I thought we were out of tokens!");
} else {
    System.out.println(secondWord + firstWord + number);
}
```

Running this prints the following:

```
Class!Hello1933
```

Is this what you expected? Notice that when we used `Scanner` to parse the tokens, it removed the whitespace from around them! Also notice that `scanner.hasNext()` returned false, because we had reached the end of the input.

Caution: you are going to want to read from `System.in` instead of a fixed string. `System.in` has the peculiar property that `hasNext()` and its variants will always return true, because the program can't know when the user is done entering data*. It might be better to read an entire line into a string using `nextLine()`, and then use another `Scanner` to parse that string.

Calculating GPA

Students and advisors commonly need to find the GPA for specific groups of courses taken. Examples include finding the "technical GPA" or the GPA for prerequisite courses for a specific major.

Write a Java program that takes in letter grades and credit pairs and computes the GPA. Instead of taking in arguments via the command line, we will use a built-in Java class - `Scanner`.

Use the following chart for grade point values:

Grade	Grade Points
A	4.0
A-	3.667
B+	3.333
B	3.0
B-	2.667
C+	2.333
C	2.0
C-	1.667
D+	1.333
D	1.0
F	0.0

*Unless the user presses Control-D, which is not intuitive!

Some examples follow:

```
java GPA
a 4 b 4 a 2 b- 4
The GPA is: 3.33

java GPA
a 4
The GPA is: 4

java GPA
c- 4 b+ 4 a 3
The GPA is: 2.91
```

To calculate GPA use the following method:

$$\text{gpa} = \frac{\sum_i (\text{grade}_i \times \text{credit}_i)}{\sum_i \text{credit}_i}$$

For example, if you enter `java GPA a 4 b- 3 c+ 2` your calculation would be:

$$\text{gpa} = \frac{(4.0 \times 4) + (2.667 \times 3) + (2.333 \times 2)}{9} = 3.185$$

Test your code thoroughly. When does it work? When does it fail?

5 Write a Rational Number Class

Create your own class called `Rational`. `Rational` should parallel Complex3 from lecture. It will include a constructor, but no default constructor. It will have get and set methods for the numerator and denominator attributes. It will also include the following methods:

- `addRational(Rational r)`
- `subRational(Rational r)`
- `mulRational(Rational r)`
- `divRational(Rational r)`

that will perform the appropriate rational arithmetic. You do NOT need to reduce the result to lowest terms, however. Write tests for all of the methods for the Rational class.

Reflection: What should the access modifier for your methods be? What shouldn't they be?

6 The Long and Short of It

There are a few places, such as in printed checks, where dollar amounts are written on long hand as well as numerically. For example, $1328.50, can be written out in longhand as

one thousand three hundred twenty eight and 50/100 dollars.

Write a program that will accept a dollar amount as a command line input and print out the amount in longhand notation. The program should accept amounts that are in the range 0.00 - 1000000.00. Suggestion, it may be best to convert the command line String value to a double before converting to longhand. This can be done with `Double.parseDouble(...)`. Use a combination of switch and if conditionals in your solution. Write tests for a variety of inputs to verify correctness.

Assume that your input does not have commas, but does have a $ symbol as in the example above. After you complete your lab and get it checked off, try and see if you can write your code so that it accepts inputs that have commas, e.g. $1,328.50.