CISC 181 Spring 2014
Practice Set 1
Assigned: February 17
Due: February 23 at 11:55PM on Sakai

Practice sets are to be completed individually. You are free to consult other students to help complete the practice sets (see syllabus for collaboration policy). However, keep in mind that each practice set is designed to cover basic material on which you will be quizzed and tested.

This practice set is intended to cover the following major topics:

- Writing methods to meet tests
- Boolean logic
- Mutation of local variables
- Basic iteration

Please consult chapters 1-5 of your textbook, and your class notes for additional examples that will help with concepts found in these problems. Note: The review problems at the end of each chapter generally contain enough code to guide you to an answer, which is great practice. Also, the solutions for even numbered problems are available at http://www.cs.armstrong.edu/liang/intro9e/exercisesolution.html.

Due to the cancelled class period on February 14, you should watch the two videos on boolean logic posted at:

http://codingbat.com/doc/java-if-boolean-logic.html

I would also recommend that you read through the brief help on that page and on:

http://codingbat.com/doc/java-for-while-loops.html

The author of these resources, Nick Parlante, is a CS teacher at Stanford who I have met on several occasions and espouses a similar practical approach to coding in the small with unit tests. There are actually many extra practice problems on the codingbat.com site that you might want to use as an additional resource.

Setup - you must do this first

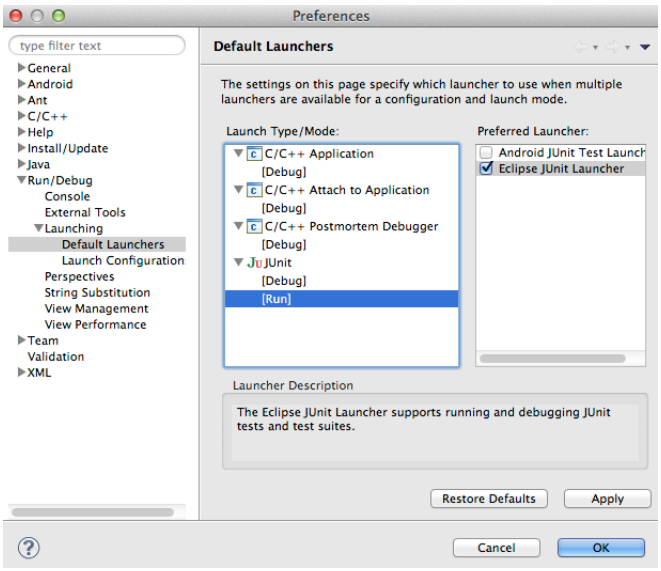
- Create a new Java project in Eclipse called PS1.

- You will also need JUnit for this project: Right click your project and click Properties (or go to menu Project->Properties). Choose Java Build Path on the left, and then the Libraries tab on the right. Choose Add Library... then JUnit, Next, choose version 3 and then Finish.

- To avoid an annoying dialog popup later, set the default JUnit launcher to Eclipse.

  Windows: Go to Windows->Preferences
  Mac: Go to ADT->Preferences
  (or Eclipse->Preferences)

  Both: Now go to
  Run/Debug->Launching->Default Launchers
  and under JUnit->Run choose
  Eclipse JUnit Launcher and click OK:



- Download the JUnit test file provided (PS1Tests.java) and put it in your Eclipse PS1 project under the src folder (you may need to tell Eclipse to refresh your project to see the file -- right-click the project and choose refresh). Use the tests as a guide for the problems below. **Hint: Start by looking at the tests!** Do not modify the actual tests, but you will want to uncomment the test for each problem as you progress.

- Create a class named PS1. This class will consist solely of *public static* methods that are used by the tests in PS1Tests.java.

- Please place a comment before each method describing its purpose and general algorithm. You do not need to comment individual lines of code in this assignment.

---

1. **[10 points]** How cold is it outside? The temperature alone is not enough to provide the answer. Other factors including wind speed, relative humidity, and sunshine play important roles in determining coldness outside. In 2001, the National Weather Service (NWS) implemented the new wind-chill temperature to measure the coldness using temperature and wind speed. The formula is:

   $$t_{wc} = 35.74 + 0.6215t_a - 35.75v^{0.16} + 0.4275t_a v^{0.16}$$

   where t_a is the air temperature in Fahrenheit and v is the wind speed in mph.

   Write an implementation for a static method, **`windChillTemperature`**, that computes this formula. Hint: you will need to use the static method, Math.pow from the built-in Math class (see http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html). You also must compute this efficiently (do not call Math.pow more than once!).

2. **[15 pts]** We are having a party with amounts of tea and candy. We want a static method that takes two parameters, tea and candy, and returns the outcome of the party encoded as a 0, 1, or 2 (bad, good, or great respectively). A party is good (1) if both tea and candy are at least 5. However, if either tea or candy is at least double the amount of the other one, the party is great (2). However, in all cases, if either tea or candy is less than 5, the party is always bad (0).

   Write an implementation for a static method, **`teaParty`**, that implements this logic.

3. **[15 pts]** Given two ints, each in the range 10..99, return true if there is a digit that appears in both numbers, such as the 2 in 12 and 23. (Note: division, e.g. n/10, gives the left digit while the % "mod" n%10 gives the right digit.)

   Write an implementation for a static method, **`shareDigit`**, that implements this logic **without using a conditional statement (no if, no switch, no ternary operator)**.

4. **[20 pts]** We can find the closest (smaller) factor to the square root of a given number using an iterative algorithm:
   a. Start at the whole number closest (but smaller than) the square root of a number
   b. Check to see if it evenly divides the number
   c. If so, stop because it is our closest factor. If not, decrease the number by 1 and try again.

For example, the closest factor for the number 10 can be found by
   √10 which is ~3.1622
   3 does not go into 10 evenly
   2 goes into 10 evenly 5 times, so the closest factor to the square root for 10 is 2

Write an implementation for a static method, **`closestFactorToSqrt`**, that implements this logic using a while loop.

5. **[20 pts]** Many digital identification numbers such as ISBNs, credit card numbers, bank routing numbers, and UPCs are designed with an extra digit that helps detect (and sometimes correct) human entry errors. In this problem we will assume a simple check-digit algorithm:

   All correct identification numbers should have odd parity. Odd parity is determined by adding all of the digits of the number and checking to see if the result is odd.

   4532120 is odd parity because 4+5+3+2+1+2+0 = 17 which is odd, whereas
   4532121 is not odd parity because 4+5+3+2+1+2+1 = 18 which is even

Write an implementation for a static method, **`oddParity`**, that implements this logic using a while loop. You **may not** convert the long to a String or array. Hint: make sure you have completed problem #3 before attempting this problem.

6. **[20 pts]** Computers are great at simulating mathematical models because they can quickly check and summarize thousands (if not millions) of samples per second. In this problem we will be using the simple dice game method written in class:

```
/**
 * Scoring rule for a simple dice game.
 *  3 of a kind - score 20
 *  2 of a kind - score 10
 *  otherwise score is the largest die roll
 */
public static int score(int d1, int d2, int d3) {
    if (d1 == d2 && d2 == d3) {
        return 20;
    }
    else if (d1 == d2 || d2 == d3 || d1 == d3) {
        return 10;
    }
    else {
        return Math.max(Math.max(d1, d2), d3);
    }
}
```

- Add the above score method to your PS1.java code. Note that there will only be a JUnit test for the last part of this problem due to the random numbers involved.

- Create a new method named **`scoreTurn`** that will take an int parameter representing the number of sides of the game die, and produce a score from a random roll of 3 of those dice. For example:

  ```
  scoreTurn(6) --> 5 (on a 6-sided die the options are 1, 2, 3, 4, 5, 6
  and in this sample the three dice rolled 2, 3, and 5)
  ```

  To complete this method we will need to generate a random number within a range of values for each die roll. We can do this by calling Math.random(), which returns a double with a value between 0 and 0.999999... To convert this to an integer range:

  ```
  int d = (int)(Math.random()*6) + 1;
  ```

  In the above expression the range is multiplied by 6, giving us numbers between 0 and 5.999999..., casting this to an int gives us either 0, 1, 2, 3, 4, or 5. Adding 1 shifts the range to 1, 2, 3, 4, 5, 6. Complete your scoreTurn method using this expression for 3 dice for the given number of sides (you can assume that the number of sides given will be >= 1) and the score method.

- Create a new method named **`simulate`** that will take an int parameter representing the number of sides of the game die, and returns a double which is the average score over 1,000,000 turns using that number of sides (you should call your scoreTurn

method). Your method should now pass the test included in PS1Tests.java.

- Create a graph in Excel or similar software showing simulations for many different sided die. The x-axis should be the number of sides (make sure to run from 1 side to at least 16 sides) and the y-axis should be the average score. Save your graph as either a .png picture or .pdf and include it in your submission on Sakai.

Turn in to Sakai your entire archived Eclipse PS1 project as follows:

1. Select project
2. File-> Export -> General -> Archive File
3. Click Browse to find place on your computer to place archive and make sure it is named PS1.zip
4. Upload PS1.zip to Sakai. If PS1.zip does not include the graph from problem 6, please upload it as well.