COMS W1007x: Honors Intro to Computer Science
Fall, 2012
Assignment 3: Due Thursday, November 1, 2012, 1:10pm, in class

Part 1: Theory (44 points)

The following problems are worth the points indicated. They are generally based on the book's exercises at the ends of Chapters 5, 6, and y, under the section heading "Exercises" and "Programming Projects". They cover concepts involving ifs, loops, object-oriented design, and fancy GUIs.

"Paper" programs are those which are written on the same sheet that the rest of the theory problems are written on, and are not to be text-edited, compiled, executed, or debugged. For fairness, there will be no extra credit for doing any work beyond the design and coding necessary to produce on paper the objects or object fragments that are asked for, so computer output will have no effect on your grade. The same goes for the Theory Part in general: clear handwriting is sufficient, and computer-generated documents will earn no extra credit.

1) (3 points) Based on EX 5.10: Write a while loop that verifies that the user enters an integer value equal to or between two constant values.

2) (5 points) Based on EX 5.12: Write a method called mid that accepts three integer parameters and returns the value of the middle one.

3) (5 points) Based on EX 5.18: Can you put a bunch of Checkbox instances together in a ButtonGroup? If not, can you do something similar using the API? If so, what happens when a Checkbox is turned "on" or "off": how does the listener respond?

4) (3 points) Based on EX 6.14: Write a method called triangle that returns the sum of the integers from 1 to n, where n is a positive integer input. Note that this value should be equal to n*(n-1)/2.

5) (5 points) Based on EX 6.14: Write a method called skipTriangle that is like triangle, but also takes s, a second positive integer input, so that numbers increase by S and not by 1. For example, skipTriangle(100, 5) gives the sum of 1+6+11+16+...+96. What should the output value be equal to, in terms of n and s?

6) (5 points) Based on PP 6.9: Write a method that takes as parameter a string S, and counts and prints out the number of times a "Latin liquid" lower case character occurs. Latin liquids are simply the characters 'l', 'm', 'n', and 'r'. Since the world is handing you this enum collection, you can use the switch statement.

7) (5 points) Based on EX 7.4 and EX 7.5: Write a method that takes a String s and an integer n, and returns a String that gives every nth character. So, if s=="abcdefghi" and n==3, return "cfi". Then, write a second method that overloads the first so that if there is only a String s input as a parameter, n is assumed to be 2: that is, it would return "bdfh".

8) (3 points) Based on EX 7.11: Explain why a static method can refer to static variables, but not instance variables. Then, explain why there can or cannot be static parameters.

9) (5 points) Based on EX 7.15: Create an interface that Paintable that has a method called paintIt that takes as a parameter a Color, and a method called getPaint that returns a Color.

10) (5 points) Based on Ex 7.17: Assume that you have a system that has a class called Block, which represents in the real world a wooden block that can be painted. Blocks have width, depth, and height, and you can stack a Block on another Block, returning a new Block (just like you can concatenate Strings). Show the UML that shows the relationship of class Block with interface Paintable.

Part 2: Programming (56 points)

This assignment is intended to exercise some the more advanced concepts in ifs, loops, and object-oriented design. It is based on PP5.7, "rock paper scissors", except we extend it to "rock paper scissors lizard spock". For reference, please see the following resource:

`http://en.wikipedia.org/wiki/Rock-paper-scissors-lizard-Spock`

Just to bring this all into the real world, see the following article that shows how this "game" has life and death consequences, at least for real lizards. We won't be using this resource, but it is fascinating nonetheless:

`http://en.wikipedia.org/wiki/Common_side-blotched_lizard#Mating_strategy`

The goal of this assignment is not only to exercise some of the more complex GUIs of Java, but also to show how badly people play these games. And, in contrast to Assignment 2, where the computer could win by cheating, in this assignment you are asked to help the computer win by having some artificial intelligence.

For guidance for the look and feel of this project, you can look at Figure 7.12 on page 370 and Listing 7.24 on page 368, as well as the pop-ups in Listing 6.9 on page 293. Some of the book's code in the Online Companion can be useful, but in this assignment much of this design will be on your own.

Please recall that all programs must compile, so keep a working version of each Step before your proceed to the next.

Step 1 (18 points): Getting started: understanding the game logic.

First, write a console application that uses text output and text input. This is often the way applications are developed: you first get the "story" right, then you draw the "pictures".

The application should do the following: It first displays some welcome text that give the rules. Then the computer selects its own random move, or "throw". Then it asks the user for a single character from 'r', 'p', 's', 'l', 'k', or 'z'. When a valid character is thrown by the user (make a loop to do sanity checking!), the game stops if the throw is 'z', the sound of sleeping. Before it ends, though, it shows how many rounds of the game the computer won, how many the user won, how many were tied, and the percentage of user wins compared to total wins, and the percentage of ties compared to total rounds. If the character thrown is one of the five that are legal for the game, the computer then references the rules (which probably are best encapsulated in a private helper method) to see who won this round. Then it prints what it had chosen, what the user had chosen, who won, and then repeats.

Do what is necessary in a Tester class to show that your RPSLK class works as required and as documented. You probably should consider that the strings in "rock paper scissors lizard spock" have been determined by the real world and are not negotiable, so it is very likely that using enum will help simplify your design and code.

It is likely that you will discover that the computer, simply by playing randomly, tends to win more than the user does, and that ties are approximately 1/5 of the total.number of rounds.

If you stop at this Step, turn in a hardcopy listing of your code, and some trial runs. Remember to use the Javadoc conventions to properly document your classes, constructors, and methods. Also, submit a softcopy listing of your code, your trial runs, and the *softcopy* of your Javadoc outputs for this Step to the *Assignments* page of Courseworks. Note that if you continue on to the Steps that make this a frame-based application, you don’t have to show that your console application worked, since you should design these earlier Steps so that much of it carries forward anyway.

Step 2 (10 points): Making it smart.

Have the computer learn your strategies (or lack of them) in the following way. Have it keep track of the user’s preferences for the five throws. You can do this by using an ArrayList or even just simple variables, for example, totalUserSpocks == 3. Then, without cheating, have the computer select for its own throw something that the accumulating history of user preferences indicates is more likely to lead to a win. This of course is best encapsulated in a class, which you can call Smarts. It should have method(s) that accept new information about the most recent round, and have method(s) that presents its determination of what is the best throw for it to choose.

Note that Smarts is a straightforward improvement on random guessing. And from a design perspective, even random play can be encapsulated in a Smarts class that is not too bright, since it can accept information about the most recent round (but then throw it away!) and then can present a determination of the next throw (by simply guessing).

Please also note that this artificial intelligence in Smarts can be made incredibly more complex, but we will not require it here. For example, it may be the case that a user employs what is known as the "Gambler’s Fallacy". This is the erroneous belief that somehow a coin, or a die, or deck of cards, or a random number generator "knows" that it has used "too much" of certain values, so it is more likely that the next move will be something different. That is, after 10 coin flips that each come up "heads", a user will expect "tails". It is possible to capture this, too, by recording more information in Smarts. For example, you can record what the previous round was (that is, record both the computer and the user throws), and record what the user throws this round, and see if there are some patterns in how a previous round affects the user in this round. In the extreme, Smarts can record the *entire game*, and look for patterns going all the way back. You can do some extra AI if you want, but you will not get extra credit for it.

Now, play some more rounds. What happens to the win percentage? What happens to the tie percentage? Once you see the pattern, capture your observation in the javadoc comment for your main class, so that future designers can understand its effects. Your grade for this Step will depend in part on this comment.

If you stop at this Step, turn in a hardcopy listing of your code, and some trial runs. Remember to use the Javadoc conventions to properly document your classes, constructors, and methods. Also, submit a softcopy listing of your code, your trial runs, and the *softcopy* of your Javadoc outputs for this Step to the *Assignments* page of Courseworks. Note that if you continue on to the Steps that make this a frame-based application, you don't have to show that your console application worked, since you should design these earlier Steps so that much of it carries forward anyway. And, since this Step implies the prior Step, so you don't have to submit anything for the prior Step.

Step 3 (18 points): GUI it.

Convert Step 2 into a application that uses frames. You should use checkboxes for the user input for a throw. Have the user indicate that they want to see the rule by doing something with the mouse, and then have the rules appear in a pop-up in the JOptionPane. At the end (how does the user signal this?), have the statistics for the game appear in a pop-up in the JOptionPane, too. You should probably use a BorderLayout to help organize the frame, but whatever you decide to do, you must document your choice.

What happens to the sanity check and the sentinel values once you move away from the console application? You must include a comment about this, in whatever class it would be most appropriate.

In general, a good design for the first two Steps should make this Step relatively clean and easy. In fact, a good design from Assignment 2 can help here, also.

If you do this Step, you do not have to submit separate output from Steps 1 or 2. You should submit the same kinds of outputs, including some screen shots, though, that make it clear to the TAs what your design was, what choices you made, how well it worked, how you tested it, etc.

Step 4 (10 points): Creativity step.

First, you should "pretty up" your frames by using borders. Choose some that are distinctive, helpful, and interesting, and document your choices on why these help a user find whatever is important at any time.

Second, suppose your user knows that the computer is keeping track of user preferences. Can a user now play a game against your computer that uses this knowledge to actually work *against* the computer? You might want to Google up: "mongoose cobra cybernetics", which records one of the very first observations on natural intelligence in game playing. For this step, you must record and submit your own best run (playing at least 25 rounds) against your own machine, and document in a comment in the appropriate class on how successful a user can be against this artificial intelligence. Here, "creativity" refers to your creativity as a player. Note that if you were too clever in Step 2, you may regret it in Step 4!

If you do this Step, you do not have to submit separate output from Steps 1, 2, and 3, as your output on Step 4 takes care of the predecessor steps. But this Step still requires the same *kinds* of output for the TAs as usual.

General Notes:

For each Step, design and document the system, text edit its components into files, compile them, debug them, and execute them on your own test data. When you are ready, electronically submit the text of its code, a softcopy of your Javadoc output, a copy of your testing (including any screenshots), and whatever additional documentation is required. Make sure that the source code is clear and any user interface is comprehensive and informative.

Make sure you have properly attributed any code that is legal to incorporate from other sources, including the book's online companion.

Write your classes clearly, with a large block comment at the beginning describing what the class does, how it does it, and justify how it will be tested. Put the comments about testing in a Tester class; if possible, show some test data in the comment also. Document each constructor and method.

Clear programming style will account for a substantial portion of your grade for both the Theory Part and the Programming Part of this assignment. For one reasonable set of suggested practices and stylistic guidelines, see the book's Appendix F. But it is far more important to be consistent: PoLS, everybody!

Checklist:

Here is a checklist for submission. Please note that this is designed to be a *general* guide, and you are responsible for *all* things asked for in the text of the assignment above, whether or not they appear below.

For theory: Hard copy, handed in to Prof or TA by the beginning of class. No extra points for using a text editor. Neatly stapled, separately from programming hard copy! Name and UNI attached.

For programming: Hard copy, handed in to Prof or TA by the beginning of class. All code, all testing runs (cut and pasted from console, and/or captured screenshots), all Javadoc if hardcopy Javadoc is required in hardcopy (for this assignment, it is not). Neatly stapled, and separately stapled from the theory! Name and UNI attached.

Also for programming: Soft copy, submitted to the Assignments page of Courseworks by the beginning of class, in a tarball created by CUIT Unix tar command, given below. Note that "myUNI" should be replaced with your UNI. *Please* use this convention as it makes the TAs' job much easier:

```
tar -cvzf myUNI_HW3.tar.gz whateverMyDirectoryIs
```

Also for programming: Include the softcopy of Javadoc html. Name and UNI included as comments in *every* class. The code must compile. Your last electronic submission before deadline is the official one that will be executed if needed.