CMPU 102 Final Exam

The exam is a series of problems for you to complete in NetBeans on your computer. Each problem corresponds to a NetBeans project included in the files for the exam. The instructions for each problem are included in this document, below. In solving each problem you should make appropriate changes to the corresponding NetBeans project. The exam is open-book and open notes. You may access the internet only for the purpose of consulting the following sites:

Java Documentation: http://docs.oracle.com/javase/6/docs/api/
ACM Student Package Documentation: http://jtf.acm.org/javadoc/student/
Java Tutorial On Line: http://docs.oracle.com/javase/tutorial/
Intro Java Text, David J. Eck: http://math.hws.edu/javanotes/

During the in-class exam, you must complete any five of the six problems on this exam document. When you are done with your exam, you should make a zip file of all the NetBeans projects and give the zip file a name similar to Jones-Tom-In-Class-Final.zip and upload it to the In-Class Exam Drop Box on our Moodle Site. After our in-class exam ends today, you should work on your own time to revise/complete the problems below. Completing this work constitutes your take-home exam. During the take-home exam period, you must complete any five of the six problems on this exam document. You may consult with your fellow CMPU102 students, our coaches, or with me about how to implement the problems below. You should only discuss general ideas. You should not look at other students’ code. You have 48 hours from the end of the in-class exam to complete and submit your take-home exam. (Please submit it in a zip file similar to Jones-Tom-Take-Home-Final.zip) to the Take-Home Exam Drop Box on our Moodle Site.

1. **CounterApplication**: Define a class called “**Counter**”. The constructor **Counter(int r)** for this class takes a parameter called “r” indicating the range \([0, 1, 2, \ldots, r-1]\) of values over which the counter counts. The **void increment()** method increases the current count by one, wrapping around to zero when the count passes \(r-1\). The **int getCount()** and **void setCount(int c)** methods are used to get and set the current count.

2. **ClockApplication**:
   a. Define an abstract class called “**Clock**” that keeps time in terms of hours \([0...23]\), minutes \([0...59]\) and seconds \([0...59]\) using three instances of the **Counter** class. The constructor **Clock()** should initialize the time to 0 hours, 0 minutes and 0 seconds. The **void tick()** method should move the clock forward in time by one second. The abstract **String timeString()** method should return the current time as specified by classes that extend the **Clock** class.
   b. Define a class called “**MilitaryClock**” to extend the **Clock** class. It should return time in the form \(h:m:s\) where \(h\) is in the range \(0 \ldots 23\).
   c. Define a class called “**CivilianClock**” to extend the **Clock** class. It should return time in the form \(h:m:s\ AM\) or \(h:m:s\ PM\) where \(h\) is in the range \(1 \ldots 12\).
3. **ListApplication:** Write two generic methods for the ListApplication class that demonstrate reversal of a singly-linked list.
   
   a. A method with signature: `<D> ListSL<D> reverse(ListSL<D> list)` that takes a ListSL<D> instance as input and returns a new ListSL<D>, with the same contents, but in reversed order. The input list should not be modified. Make use of appropriate methods of the ListSL<D> class.
   
   b. A method with signature: `<D> void display(ListSL<D> list)` that takes a ListSL<D> instance as input and prints its contents in order on a single line in the console window.

4. **IteratorApplication:** Implement two classes that make strings work with Java’s for-each loop in order to iterate over the characters in a string.
   
   a. **IterableString:** This class must implement Java’s `Iterable<Character>` interface. It should have a constructor that takes a String as argument. It should also have an `iterator` method that takes no arguments and returns a StringIterator.
   
   b. **StringIterator:** This class must implement Java’s `Iterator<Character>` interface, including a `Character next()` method; a `boolean hasNext()` method and a `void remove()` method. (The last of these must be present, but the body should be left empty.)

5. **FormatterApplication:** Implement a Formatter class that can be used to print a string in a specified format. The Formatter class constructor should take two parameters: a String template that describes the format in which the string should be displayed; and a char formalParameter that indicates where the string should be inserted into the template. The class should also have a format method that takes a String actualParameter as argument. It returns a new string in which all occurrences of formalParameter in template are replaced with actualParameter. Thus, if template is “I long for $, my home” and formalParameter is ‘$’, then `new Formatter(“I long for $, my home.”, ‘$`).format(“Jamaica”)` should return the String: “I long for Jamaica, my home.”. You should use the IterableString and StringIterator classes to implement the Formatter class, by iterating over the characters in template and replacing formalParameter with actualParameter in the resulting string. (You should not use any substitution methods of the String class.)

6. **TreeApplication:** A n-ary tree is a tree in which each node may have any number of children. Consider the n-ary tree class Tree defined in the TreeApplication project.
   
   a. The TreeApplication class has a Tree<String> variable called “testTree”. Modify the TreeApplication class to initialize testTree according to the diagram drawn below.
   
   b. Write an `int depth()` method for the Tree class. This method should use recursion to compute and return the depth of the deepest node in the tree, according to the following rule: A tree has depth zero, if it has no children. Otherwise its depth is one plus the depth of its deepest child tree.