CS 445 – Data Structures – Assignment#1[1]

Due: Tuesday Sept. 26th @ 11:59pm

All source files plus a completed Assignment Information Sheet zipped into a single .zip file and submitted properly to CourseWeb.

Late submission: Thursday Sept. 28th @11:59pm with 10% penalty per late day

OVERVIEW

Purpose: To refresh your Java programming skills and to emphasize the object-oriented programming approach used in Java. Specifically, you will work with control structures, class-building, interfaces and generics to **create** and **utilize** a simple array-based data structure.

Goal 1: To design and implement a simple class MultiDS<T> that will act as a simple data structure for accessing Java Objects. Your MultiDS<T> class will primarily implement 2 interfaces – PrimQ<T> and Reorder. The details of these interfaces are explained in the files PrimQ.java and Reorder.java. **Read these files over very carefully before implementing your MultiDS<T> class.**

Goal 2: To utilize your MultiDS<T> class by implementing a simple version of the card game "Snap". In this case your program will be a client using MultiDS<T> and the details of the MultiDS<T> implementation will be abstracted out.

DETAILS

Details 1: For the details on the functionality of your MultiDS<T> class, carefully read over the files PrimQ.java, Reorder.java and Assig1A.java provided on the CourseWeb folder where you find this assignment description. You must use these files as specified and **cannot remove/alter any of the code that is already written in them**. There are different ways of implementing the PrimQ<T> and Reorder interface methods, some of which are more efficient than others. Try to think of the best way of implementing these methods in this assignment, but the most important thing at this point is getting them to work. A lot of pencil and paper work is recommended before actually starting to write your code. Later we will discuss the relative merits of different implementations. Your MultiDS<T> class header should be:

```
public class MultiDS<T> implements PrimQ<T>, Reorder
```

[1] Assignment adapted from Dr. John Ramirez's class.

Important Note: The primary data within your MultiDS<T> class must be an array. You may not use any predefined Java collection class (e.g., ArrayList) for your MultiDS<T> data.

After you have finished your coding of MultiDS<T>, the **Assig1A.java** file provided for you should compile and run correctly, and should give output identical to the output shown in the sample executions (except for the segments where the data is shuffled, since it will be pseudo-random in that case).

Details 2: Snap is a card game played often by children that has many variations. You will implement a simple version as described below:

- Initially shuffle a 52-card standard deck of cards
- Deal the cards out completely to two players, alternating cards to each player
- Each player puts his cards into his **face-down** card pile and has his **face-up** pile empty.
- A **snap-pool** pile (at the center of the table) is initially empty.
- We have three card piles with face-up cards: first player's face-up pile, second-player's face-up pile, and the snap-pool.
- Do the following until one player is out of cards or until a set number of rounds have completed:
    1. Each player moves the top card of his face-down pile into his face-up pile.
    2. If two cards at the top of any two of the three face-up piles have the same rank (suits don't matter), a **match** is declared and either of the following three outcomes happen with certain probabilities (probability is given between parentheses):
        i. The first player shouts "Snap!" first (with a probability of 0.4),
        ii. The second player shouts "Snap!" first (with a probability of 0.4), or
        iii. No player shouts "Snap!" (with a probability of 0.2).
    3. The player who *correctly* (see below) shouts "Snap!" first then moves the other matching pile(s) into the bottom of his own face-down pile then shuffle them. Note that there can be a match between all face-up piles. The winning player in the round gets the face-up pile of the other player and the snap-pool pile into his face-down pile.
    4. If no cards at the top of the face-up piles have the same rank, either of the following three outcomes happen:
        i. The first player *incorrectly* shouts "Snap!" (with a probability of 0.01),
        ii. The second player *incorrectly* shouts "Snap!" (with a probability of 0.01), or
        iii. No player shouts "Snap!" (with a probability of 0.98).
    5. The player who *incorrectly* (can be both players) shouts "Snap!" moves his face-up cards into the bottom of the snap-pool pile after shuffling.

The following rules also apply to the game:

- If at any point in the game a player's face-down pile is empty, the player should move the cards in his face-up pile into his face-down pile then shuffle them.

- If at any point in the game a player has no cards remaining in both his piles, he/she has lost the game.

– If both players have cards remaining after a set number of rounds (can vary), the player with the most cards (face-up + face-down) is the winner. In this case, it is possible for a tie to occur, with each player having 26 cards.

IMPLEMENTATION REQUIREMENTS

– Your initial card deck, the players' face-up and face-down piles, and the snap-pool pile must all be stored in MultiDS<Card> objects.
– The Card class must be used as provided and cannot be changed.
– The maximum number of rounds for the game must be read in from the command line
– To allow better testing of your program, you must output when correct and incorrect snaps occur and when piles are reshuffled. See the various sample output files for required output information.

SUBMISSION REQUIREMENTS

You must submit, in a single .zip file, at least the following six complete, working source files for full credit:

1. PrimQ.java
2. Reorder.java
3. Assig1A.java
4. Card.java

The **above four files** are given to you and must not be altered in any way.

5. MultiDS.java
6. Snap.java

The **above two files** must be created so that they work as described. If you create any additional files, be sure to include those as well.

The idea from your submission is that your TA can unzip your .zip file, then compile and run both of the main programs (Assig1A.java and Snap.java) **from the command line** WITHOUT ANY additional files or changes, so be sure to test it thoroughly before submitting it.

If you cannot get the programs working as given, clearly indicate any changes you made and clearly indicate why (ex: "I could not get the reverse() method to work, so I eliminated code that used it") on your Assignment Information Sheet. You will lose some credit for not getting it to work properly, but getting the main programs to work with modifications is better than not getting them to work at all. A template for the Assignment Information Sheet can be found in the assignment's CourseWeb folder. You do not have to use this template but your sheet should contain the same information.

Note: If you use an IDE such as NetBeans, Eclipse, or IntelliJ, to develop your programs, make sure they will compile and run on the command line before submitting – this may require some modifications to your program (such as removing some package information).

RUBRICS

Please check the grading rubric on CourseWeb.

HINTS / NOTES

1. See program A1Help.java for some help with the Card class and using the MultiDS<T> class with Card objects.
2. See file A1Out.txt to see how your output for Assig1A should look. As noted, your output when running Assig1A.java should be identical to this with the exception of the order of the values after being shuffled.
3. See files SnapOut1.txt, SnapOut2.txt, and SnapOut3.txt for example runs of my Snap.java program. Your Snap program output does not have to look exactly like this but the functionality should be the same.
4. Your MultiDS<T> class will need to allow for a variable number of objects, up to some maximum number (set by the initial size). You can implement this by having an integer instance variable (ex: "count") that indicates how many slots in the MultiDS<T> are filled. For example, you could have a MultiDS<Card> with a capacity of 52 cards for a pile with anywhere from 0 up to 52 Cards in it. In this case, the array length (i.e., physical size) would be 52 but the "count" (i.e., logical size) would be some value between 0 and 52. When implementing MultiDS<T> be careful to use the "count" value rather than the array length when doing operations such as the toString() method.
5. In order for the Assig1A.java output to show correctly, you must override the toString() method in the MultiDS class. You should be familiar with the toString() method from CS 0401.
6. You can easily generate the "deck" of cards for your Snap game with nested for loops. You should NOT have 52 individual statements to do this! See A1Help.java for some hints about how to generate the deck of cards.
7. Note that for the most part, all of your methods (in both interfaces) in your MultiDS class should act on the logical data structure -- accessing only the locations in the array that actually are storing data. (See note 4 above)
8. If you have never used command line arguments in Java you can find information about them in your CS 0401 text or by Googling "Java command line argument". Note that it is required that you obtain the number of Snap game rounds from the command line -- you should not prompt the user to enter it once the program has started.
9. For Javadoc comments and code style, please refer to Appendix A of the textbook.