CS145 – PROGRAMMING ASSIGNMENT

CARD ARRAY LIST

OVERVIEW

This program primarily focuses on the implementation of a ArrayList type interface and the necessary methods to implement the ArrayList. It also includes polymorphism and class comparison.

You are expected to use Chapter 15 of the textbook to assist you.

INSTRUCTIONS

Your deliverable will be to turn in three files. The files will be named `Card.java`, `PremiumCard.java` and the last file will be called `CardArrayList.java`.

For this assignment, any use of a data control structure other than a simple Arrary or String will result in no credit. I am aware that this assignment could be done quite simply by the Collections implementation of an ArrayList<> but the point of the assignment is to do it without the standard implementation to get a feel for how they work "under the hood".

COLLECTABLE CARD

While the primary goal of this assignment will be the implementation and use of a custom ArrayList. The particular implementation we will use will focus on the idea of a simple collectable card game.

For our card game, we will assume that every card has three values expressed as integers. Each card will have three intrinsic values. These values will be labeled its **R, S, P** values. Each of these will need to be monitored to be a minimum of 1 and a maximum of 1000.

In addition, Cards then have a calculated **cost** attribute that comes from how far apart the R, S, P values are from one another.

Let X be the sum of R,P and S : $X = R + P + S$

Then the cost can be calculated by the formula:

$$cost = \left\lceil \frac{12}{10 \cdot \left(\left(\frac{R}{X}\right)^5 + \left(\frac{P}{X}\right)^5 + \left(\frac{S}{X}\right)^5 \right)} \right\rceil$$

Rounded **up** to the nearest integer.

Examples:

A card with R= 5, S = 5, and P = 5 has a cost of 98. *Which is the highest possible cost.*
A card with R= 1, S = 1, and P = 1000 has a cost of 2.
A card with R= 1, S = 2, and P = 3 has a cost of 34.

Remember to round UP.

When comparing two cards, the one with the higher cost comes after the other unless they have the same cost. If the two cards have the same cost, then the one with the SMALLER sum of R,P,, S values comes first. If the cost is the same, and R+P+S is the same, then they are equal.

{1,1,1000::2} < {1,2,3::34} = {2,1,3::34} = {3,2,1::34} < {60,70,70::93} < {600,700,700::93}

In addition, there are some cards that are considered Premium cards. If we were using a GUI interface, these would be drawn graphically different on the screen with some sort of special treatment, but for our purposes there will only be a slight difference in the output of the card information. But this will give you some practice in polymorphism.

PROGRAM DESCRIPTION

In this assignment you will construct an ArrayList that can manage a set of Cards which includes both normal and premium cards. You will implement the various methods that are required in order to make the ArrayList function.

A sample program will be provided for you to test with, but you may want to alter it to check different functionality of your ArrayList.

YOUR INSTRUCTIONS

You will write the following classes, implementing the necessary methods as appropriate.

NECESSARY METHODS – CARD CLASS

The Card class will be your primary class to contain one particular card. It should have three interior fields for R,P and Sand should calculate the cost as needed. **Do not store the cost as a field!'**

Cards should correctly implement the comparable interface to other Cards as discussed in class using the ordering described above.

You might need additional methods not listed below…

PUBLIC CARD()

In the `Card()` constructor you create a random card with a random R, P, and S values within the acceptable bounds given above.

PUBLIC CARD(INT X)

In the `Card(int x)` constructor you create a card that has R, P, and S values all equal to the parameter x assuming that it is within the proper bounds.

If the input value is outside the bounds, then throw an appropriate exception.

PUBLIC CARD(INT R, INT P, INT S)

In the `Card(int R, int P, int S)` constructor you create a card that has R, S, and P values equal to the input values as long as they are within the proper bounds. If any input value is outside the bounds, then throw an appropriate exception.

PUBLIC INT GETR()

The `getR()` should return the current R value..

PUBLIC INT GETP()

The `getP()` should return the current P value.

PUBLIC INT GETS()

The `getS()` should return the current S value.

PUBLIC INT GETCOST()

The `getCost()` should return the current cost, calculated by the current values.

PUBLIC BOOLEAN EQUALS(CARD)

The `equals(Card x)` should return if the current Card and the X card are exactly the same in terms if R,P, and S.

Note, this will not help you with sorting… that will require something… "else".

PUBLIC STRING TOSTRING()

In the `toString()` should return a string that shows the current power, toughness, and cost of the card in the form "`[R,P,S::C]`" where the first number is the R, the second number is the P, the third number is the S, and the number after the "::" is the calculated cost.

PUBLIC VOID WEAKEN()

The `weaken()` method should find the smallest R,P,S and move it closer to 0 by 5, if the are low, pick any one of them. *(Note it still must stay inside the range after subtracting)*

For example a card with values [50, 60, 80] 50, is the smallest element, so make 50 become 45.

PUBLIC VOID BOOST()

The `boost()` method should find the smallest R,P,S and move it closer to 1000 by 5, if multiple are low, pick any one of them. *(Note it still must stay inside the range after adding)*

For example a card with values [50, 60, 80] 50, is the smallest element, so make 50 become 55.

Also, if they are all the same, do nothing.

NECESSARY METHODS – PREMIUM CARD CLASS

The `PremiumCard` class will be a secondary type of card. In a more advanced program there would be some extra work to implement this type of card graphically, but for this assignment, there is only one overridden function that you need to implement. All other methods from card should work the same, including comparison. However, you might have to include constructors that call the constructors from `Card`.

PUBLIC STRING TOSTRING()

In the `toString()` for PremiumCards should return a string that shows the current power and toughness of the card in the form "`{R/P/S::147}`" where the first number is the R, the second number is the P, the third number is the S, and the number after the "::" is the calculated cost. Note the different look using the "`/`" and the "`{}`" values.

NECESSARY METHODS – CARDARRAYLIST CLASS

The `CardArrayList` is the primary focus of this assignment. In it you will maintain a list of cards, allowing for all the methods listed below to manage a list of cards.

The basic idea of the `CardArrayList` will be to keep track of an internal array of cards and a list of how many of the array slots are currently being used. As values are added/removed to the internal array, you will update the "size" field to keep track of how many are being used.

Should the user try to add a value to the array that would cause it to overflow, then you will need to implement routines to create an array that is double the size of the current array, then copy the current array into the new array, then replace the old array with the new array. When that is done, then you may complete the original operation that caused the resize.

However, we want to be able to change the type of list in the future, so your `CardArrayList` will need to implement the `CardList` interface in order to work. So while the `CardList` interface is provided for you, you will need to make sure that your program properly implements it.

PUBLIC CARDARRAYLIST ()

In the `CardArrayList()` constructor should create an initial array of size 10, and set the internal size counter to zero to represent that none of the spaces are currently being used.

PUBLIC CARDARRAYLIST (INT X)

In the `CardArrayList()` constructor should create an initial array of size x, and set the internal size counter to zero to represent that none of the spaces are currently being used. If x is less than one, throw an appropriate exception.

PRIVATE VOID EXPAND()

A good idea for your class is to create a private `expand()` method that will double the size of your array and then copy the old array into the new array for storage. This should not change the value of the size counter, just make it big enough for added storage.

PUBLIC STRING TOSTRING ()

The `toString()` method should print the current values of the arraylist being stored. It should surround the entire cardArraylist inside [0: :*size*] with commas between the values. A sample output might look like.

```
[<--: [2,3,4:55],[8,9,10:90],{{{1|1|2:37}},[4,5,6:82] -->4]
```

Note the 4 to show the current size. If the array is empty it should print out:

```
[0: :0]
```

PUBLIC INT SIZE ()

In the `size()` method, you should return the current number of elements being stored in the array. Not the size of the array, but how many elements it is currently holding.

PUBLIC VOID ADD (CARD X)

In the `add()` method, you should add the card provided to the array list in the last empty location and increment the size counter. *Note that this may require a resize before running*.

PUBLIC CARD REMOVE ()

In the `remove()` method, you should return the last element from the array list. You should then decrement the size counter by one to show that we don't care about that element anymore. You don't actually have to delete it, it effectively gets removed by being ignored.

Make sure to return the Card as you exit however so it can be used by the internal program.

PUBLIC CARD GET (INT X)

In the `get(int x)` method, you should return the card located in the array at the x location. This method does not delete the element; it just returns the current value. If x is outside the bounds of the array, throw an appropriate exception.

PUBLIC INT INDEXOF (CARD X)

In the `indexOf()` method, you should return the location of the first card that is "equal" to the card that is provided. If it isn't found, return -1. *Note that the card found may not match the card given precisely due to the unusual comparison of cards.*

PUBLIC VOID ADD (INT L, CARD X)

In the `add(location,x)` method, you should add the card(x) provided to the array list in location x. However because the location is inside the array, you will need to move everything after the location over one spot. So you will need to move everything to make room, then add x into location. *Note that this may require a resize before running*. Make sure that x is inside the current array OR at the end, do not extend the array more than one value. Make sure to alter the size counter as appropriate. If x is outside the bounds of the array plus one, throw an appropriate exception.

PUBLIC CARD REMOVE (INT J)

In the `remove(int j)` method, you should remove the element from the array list in location j and then return it. However in this method, the item may be in the middle of the array, so you will need to store the item, then move everything after the item to the left one spot, then adjust the size counter. If j is outside the bounds of the array in use, throw an appropriate exception.

Make sure to return the Card as you exit however so it can be used by the internal program. *Hint: you will need a temp holder to hold the return card while you are readjusting the array.*

PUBLIC VOID SORT ()

The `sort()` method should simply sort the array from smallest to largest. However I want you to implement your own version of the **merge** sort. **Do not use Arrays. Sort()**.

Also, keep in mind that you will not be sorting the entire array. Your array might be size 100 but you are only using 60 elements currently. Your merge sort should take into account that you are only sorting a section of an array. *This is a hard method, and only worth a couple of points, so spend your time wisely here if you are having problems.*

PUBLIC VOID SHUFFLE()

The `shuffle()` method should shuffle the array into a non-ordered arrangement. One way of doing this is picking two random numbers within the size of the array, and then swapping those two values. Then repeat this process a bunch of times. *(For example five times the number of elements in the array list).* Also make sure that you are only shuffling the part of the array that you are using, and not the "empty" array elements.

Note that this isn't mathematically a good shuffle, but it will work for our purposes.

PRIVATE BOOLEAN ISROOM ()

A good idea for your class is to create a private `isRoom()` method that will check to see if adding one more element will require the array to grow. This way you can use this method before accidently adding a value that will cause and overflow.

PRIVATE VOID SWAP(INT A, INT B)

A good idea for your class is to create a private `swap(a,b)` method that will swap two cards around as necessary. Helpful for methods above.

PUBLIC MYSTERY METHOD

There is one more method that is required according to the CardList interface. You need to find the method and implement it correctly.

NOTES AND HINTS

Chapter 15 contains a number of ideas on how to implement an ArrayList of numbers, so do not be afraid to consult the chapter for hints/ideas.

The testing program is divided up into stages. Comment out the stages that you haven’t finished and work on things step by step.

`.equals()` will not help you with the sort(), what other method might you use?

Look at the Java Documentation on the CardList interface to know what exceptions to throw.

STYLE GUIDELINES AND GRADING:

Part of your grade will come from appropriately utilizing object methods.

Your class may have other methods besides those specified, but any other methods you add should be private.

You should follow good general style guidelines such as: making fields private and avoiding unnecessary fields; declaring collection variables using interface types; appropriately using control structures like loops and if/else; properly using indentation, good variable names and types; and not having any lines of code longer than 100 characters.

Comment your code descriptively in your own words at the top of your class, each method, and on complex sections of your code. Comments should explain each method's behavior, parameters, return, and exceptions.