CS20B: Homework 4 (55 points)

prof. dr. Irma Ravkic

Questions/Reading exercises (26 points)

1. (10 points) Read section 4.9 (Concurrency, Interference, and Synchronization) and do the following activities. Demo code for this exercise is provided for you in Canvas (ch04.zip).

   (a) (5 points) Edit the Demo06.java code to run the application 100 times and report how many times you receive the expected/correct. Also copy-paste the modified code into your answer document!

   i) Create/draw a table that will summarize the result of running the application 100 times in the following way:

Expected Result	Unexpected Result
#times	#times

   denoting how many times out of 100 you received the result you expected, and how many times not? Ignore the exceptions, just looks at the number of correct results. ii) Explain this result?

   (b) (5 points) Do the process specified in a) above for Demo07.java.

   Note: Demo07.java should use SyncArrayBoundedQueue class instead of ArrayBoundedQueue

   Hint: this relies just on a for loop which would run 100 times and compare with each run whether the final count is equal to the expected count. You can modify the Demo_.java file itself for this. You just need to rearrange some stuff.

   (Hint: First read the section in order to know what result to expect for each of the Demo codes; you don't need to generate the table using Java code, but System.print the number of time expected result occured and create/draw the table in some other tool (Word or similar))

   (Disclaimer: I know you're not supposed to know Multithreading in Java in this course, but this Demo will provide you a very important skill of looking at a code you don't fully understand, and being able to run it and analyze it or its performance)

2. (16 points) Read Chapter 5, view all the code examples there: even those we didn't see in the class (the complete code for classes from this Chapter is provided for you, you can also find partial code in the book). Answer the following questions:

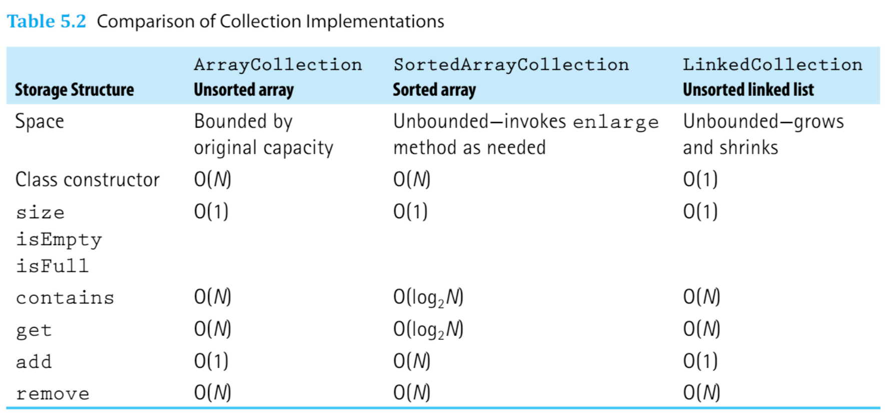

Table 5.2 Comparison of Collection Implementations

Storage Structure	`ArrayCollection` **Unsorted array**	`SortedArrayCollection` **Sorted array**	`LinkedCollection` **Unsorted linked list**
Space	Bounded by original capacity	Unbounded—invokes `enlarge` method as needed	Unbounded—grows and shrinks
Class constructor	$O(N)$	$O(N)$	$O(1)$
`size` `isEmpty` `isFull`	$O(1)$	$O(1)$	$O(1)$
`contains`	$O(N)$	$O(\log_2 N)$	$O(N)$
`get`	$O(N)$	$O(\log_2 N)$	$O(N)$
`add`	$O(1)$	$O(N)$	$O(1)$
`remove`	$O(N)$	$O(N)$	$O(N)$

Figure 1: Comparison of Collection Implementations.

(a) (6 points) Look at the table provided in Chapter 5 (see Figure 1 in this file). Explain the following:

  i. (2 points) Why does *SortedArrayCollection* exhibit logarithmic time for *contains* and *get* methods?

  ii. (2 points) Why do *ArrayCollection* and *LinkedCollection* exhibit $O(1)$ time for *add* (adding an element method), while *SortedArrayCollection* exhibits a worse $O(N)$ time for the same method?

  iii. (2 points) How is $O(1)$ accomplished for *size* method for all of these collections in the table?

(b) (2 points) a) Which method do we need to implement to check if the contents of our objects are the same? b) Write its signature. c) Is it inherited from some class? d) If yes, which one?

(c) (4 points) a) Which interface in Java provides the functionality of comparing objects of a class? b) Write the signature of the abstract method(s) this interface contains? c) Change the signature of this class below such that the elements of the generic type K can be compared:

```java
public class SomeSortableList<K>{
   protected K[] elements;
   //doesn't matter what comes here
}
```

(d) (2 points) Based on the equals method for Circle objects defined in Section 5.4, "Comparing Objects Revisited," what is the output of the following code sequence and **why**! (you need to provide 8 answers for each printout, and a very short explanation of why it is so)?

```java
Circle c1 = new Circle(5);

Circle c2 = new Circle(5);

Circle c3 = new Circle(15);

Circle c4 = null;

System.out.println(c1 == c1);

System.out.println(c1 == c2);

System.out.println(c1 == c3);

System.out.println(c1 == c4);

System.out.println(c1.equals(c1));

System.out.println(c1.equals(c2));

System.out.println(c1.equals(c3));

System.out.println(c1.equals(c4));
```

(e) (2 points) Consider the following interfaces:

```java
public interface Event{
    public void RSVP();
}

public interface PublicEvent extends Event{
    public void doSecurityCheck();
}

public interface PrivateEvent extends Event{
    public void checkIdentification();
}

public interface PartneredEvent extends Event{
    public void findPartner();
}
```

Which method(s) should implement a class

i. (1 point) called *Hockey* that implements *PublicEvent*?

ii. (1 point) called *Prom* that implements *PrivateEvent* and *PartneredEvent*.

Programming exercises (29 points)

For the programming exercises, please follow the provided instructions in the instructions.pdf. Each of the exercises below should be a separate zip of a Java project.

1. (6 points) Add the following methods to the *LinkedQueue* class (from ch04.zip), and create a test driver for each to show that they work

correctly. In order to practice your linked list coding skills, code each of these methods by accessing the internal variables of the LinkedQueue, not by calling the previously defined public methods of the class.

   i. (2 points) *String toString()* creates and returns a string that correctly represents the current queue. Such a method could prove useful for testing and debugging the class and for testing and debugging applications that use the class. Assume each queued element already provides its own reasonable toString method.
   ii. (4 points) *void remove(int N)* removes the front *N* elements from the queue; throws *QueueUnderflowException* if less than *N* elements are in the queue.

2. (17 points) Add the following methods to the *ArrayCollection* class, and create a test driver for each to show that they work correctly. **Note** that you can find the implementation of the classes mentioned here in *ch05.zip* or check the book for the implementation details. Some classes require imports from other packages. All the packages you need and classes can be found in the code I shared with you. In order to practice your array coding skills, code each of these methods by accessing the internal variables of the ArrayCollection, not by calling the previously defined public methods of the class. Make sure you have a driver class that will put your methods to use (and also serve to check whether your methods are correct).
   i. (2 points) *String toString()* creates and returns a string that correctly represents the current collection. Such a method could prove useful for testing and debugging the class and for testing and debugging applications that use the class. Assume each stored element already provides its own reasonable *toString()* method.
   ii. (4 points) *int count(T target)* returns a count of the number of elements **e** in the collection such that **e.equals(target)** is true.
   iii. (5 points) *void removeAll(T target)* removes all elements **e** from the collection such that **e.equals(target)** is true.
   iv. (6 points) ArrayCollection<T> combine(ArrayCollection<T> other) creates and returns a new ArrayCollection object that is a combination of this object and the argument object. **(Note: add the signature of this method to *CollectionInterface*. It is not the best programming practice, but for the time being, do that).**
3. (6 points) Create a *FacebookUser* class that includes *numFriends* and *username* attributes of type int and String, respectively, both set by a constructor. The attribute *numFriends* represents the number of friends a user with *username* has. Implement a *compareTo* method from Comparable interface for this class so that *FacebookUser* objects can be ordered/compared based on the

   a. Number of Friends.

   b. Username.

Note: you should create this class, provide two compareTo methods, one for a) and one for b), and use a driver class to test your code with comparing two objects of the class. A cooler way to test it is to create a list of FacebookUser objects, and use Collections.sort(list); Check the book to see more details how this is done for other classes.

Next are two bonus exercises. These can make up for the lost points on some other exercises, or in general can make me decide in favor of the higher grade when borderlining. Submit a solution only for one of these! To receive credit you have to write a driver class I can run and see your solution work, and 90% of your code should work.

4. (Bonus Exercise) *This one is not as hard as the next Bonus Exercise. Remember, choose doing one of these depending of which one you like more.*

   An *Advanced Set* includes all the operations of a *Basic Set* plus operations for the union, intersection, and difference of sets.

   a. Define an Advanced Set interface.

   b. Implement the Advanced Set using an unsorted array; include a test driver that demonstrates your implementation works correctly.

   Hint: you can find the implementation of Basic Set in ch05.zip code, and also you can check the book for more details.

5. (Bonus exercise) *This is a rather challenging but very interesting exercise from Chapter 4.*

   Design and code the Queue ADT operations using a circular linked implementation. The book gives you some hints on how to do this in the end of Section 4.5.