Mid-Semester Examination (R) – Object Oriented Programming (CS F213)
First Semester: 2022-23

Dr. Amit Dua & Dr. Tanmaya Mahapatra
Department of Computer Science and Information Systems

Name and ID: _____

Important Information:

- Write your name and ID on the question paper.
- Please check the completeness of the exam booklet **(8 double-sided pages)**
- Use only **blue** or **black** pen to answer the questions; Pencils, green and red pens are not allowed!
- You are only allowed to answer the questions using the provided exam sheets.
- Working time is **90 minutes** for **3 questions** of **60 points!**
- On any attempt to **cheat**: the exam will be assessed with a **0** and you would be barred from the makeup exam too!
- Permitted writing aids: none — **closed book exam!**

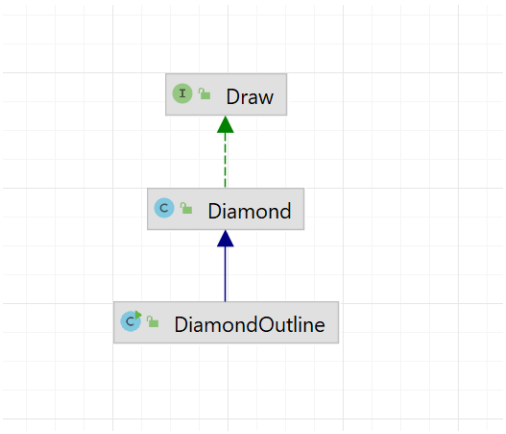
---

1. Java fundamentals with concepts of inheritance and usage of interfaces. **Question 1: 25 pts**

   (a) We have to write a program which would take an integer, a symbol as input from terminal and print a specific pattern on the console. Now, we have a hierarchy as depicted in Figure 1a. [20]

   1. There is an interface **Draw** which provides a method. This interface should provide the logic for the method to print a basic square pattern (Figure 1b).
   2. Class **Diamond** should implement this interface and provide the logic to print the diamond pattern (Figure 1b).
   3. Finally, Class **DiamondOutline** should provide the logic to print the outline of a diamond.

   Study the code snippet given below and complete it to achieve the required functionality. Study the **main()** and refer to **Figure** 1 to understand the flow. *The code snippet contains the distribution of grades for this task in the code comments.*



(a) Hierarchy of classes

```
Enter the number of rows: 3
Enter the symbol you want to print: $
$$$
$$$
$$$
  $
 $$$
$$$$$
 $$$
  $
  $
 $ $
$   $
 $ $
  $
```

(b) Execution of the code

Figure 1: Figure for Question 1a

```
1  public interface Draw {
2
3      // Modify as required [2 Points]
4      public void drawPattern(______________________);
5
6  }
7  public class Diamond implements Draw {
8
9      // Method Overriding
10     public void drawPattern(______________________) {
11         // Complete the switch case [2 Points]
12         switch (_______) {
13              // Fill the cases appropriately
14             default:
15                 System.out.println("Wrong Choices");
16         }
17     }
18
19     //Specify the correct access specifier [1 Point]
20     _______ void drawPattern(int n, char symbol) {
21
22         // Write the Logic to print the required pattern
               [5 Points]
23     }
24 }
25
```

```
26 import java.util.Scanner;
27 public class DiamondOutline extends Diamond {
28
29     //complete to support Method overriding [1 Point]
30     public void drawPattern(______________________) {
31
32         // Complete the switch case [2 Points]
33         switch (_________) {
34             // Fill the cases appropriately
35             default:
36                 System.out.println("Wrong Choices");
37         }
38     }
39
40     //Specify the access specifier & support Method
          Overloading [0.5 Point]
41     _____ void drawPattern(___________) {
42
43         // Write the Logic to print the required pattern
             [6 Points]
44     }
45
46     public static void main(String args[]) {
47         Scanner scan = new Scanner(System.in);
48         System.out.print("Enter the number of rows: ");
49         int rows = scan.nextInt();
50         System.out.print("Enter the symbol you want to
             print: ");
51         char symbol = scan.next().charAt(0);
52         //Complete the below statement [0.5 Point]
53         Diamond obj = ____________
54         obj.drawPattern(rows,"default", symbol);
55         obj.drawPattern(rows,"pattern", symbol);
56         obj.drawPattern(rows,"outline", symbol);
57     }
58 }
```

```
FALSE
TRUE
FALSE
TRUE
TRUE
FALSE
FALSE
TRUE
TRUE
TRUE
It took 10 alterations to generate 3 consecutive Trues.

Process finished with exit code 0
```

Figure 2: Figure for Question 1b

(b) Write a code-snippet that simulates repeated alteration between true and false automatically. This continues until three consecutive true values are generated. At that point, your program should display the total number of alterations that were made. Figure 2 illustrates one possible sample run of the program. [5]

2. Java Collections and String manipulations in Java. **Question 2: 20 pts**

(a) The **Collatz Conjecture** states that if you pick a number and if it is even then divide it by two and if it is odd then multiply it by three and add one. You repeat this procedure till you are mentally exhausted and decide to take a tea break. Now, observe the given figure below. [10]

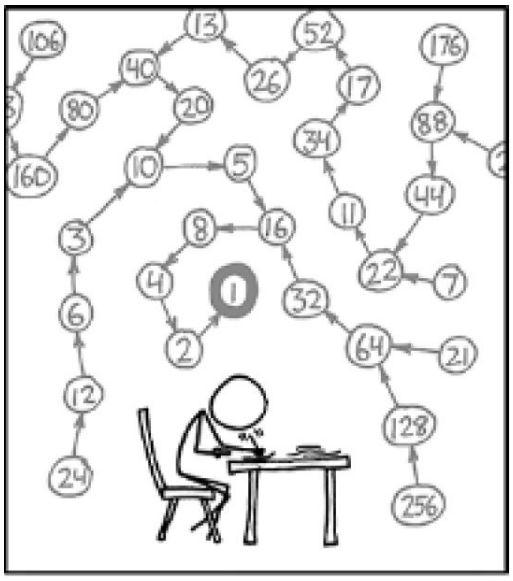


Figure 3: Figure for Question 2a

In the Figure 3, we make reference to the ***hailstone sequence*** (the Collatz Con-

jecture – which is the as-yet unresolved claim that this process always terminates). As the figure makes clear, many of the sequences include common patterns. For example, all hailstone sequences (except for 1, 2, 4, and 8) go through the value 16 and therefore share the last four elements of the path. If you've followed through this path once, you don't need to follow it again. Keeping track of these common patterns can make calculating the length of hailstone sequences vastly more efficient. Suppose, for example, that you are calculating the number of steps in all the hailstone sequences between 1 and 100. What happens when you get to 24, which appears at the lower left corner of the figure? By the time you get to this point, you've already figured out that it takes nine steps to get from 12 down to 1, so that 24 must take ten steps, given that 12 is just one step away from 24.

It's even more interesting to follow what happens with 7, which is in the middle of the right hand side. Here, figuring out the number of steps for 7 entails running through the number 22, 11, 34, 17, 52, 26, 13, 40, and 20 before you come to 10, which you've already encountered in counting the number of steps from 3 and 6. At this point, you have learned an enormous amount. Given that it takes six steps to reach 1 starting at 10, you know that 20 takes seven steps, 40 takes eight, 13 takes nine, and so forth, backwards along the list of numbers in the chain, until you determine that there are sixteen steps in the hailstone chain between 7 and 1. If you kept track of these values in, for example, a **HashMap<Integer,Integer>**, you'd already know the answers for the other values in the chain. Keeping track of previously computed values so that you can use them again without having to recompute them is called ***caching***.

Complete the function skeleton given below. This should determine the number of steps in the hailstone sequence starting at ***n***. The *second parameter is a* ***HashMap*** *containing previously computed values. Your implementation should look up each value it encounters during the process to check whether the answer from that point is already known. If so, it should use that previously computed value not only to compute the current result, but also to add the counts for all the intermediate steps to the HashMap.* Thus, if you call countSteps with 7 as the first parameter, your code should add the counts for 20, 40, 13, 26, 52, 17, 34, 11, 22, and 7 to the HashMap before returning the answer.

```
1 import java.util.*;
2 public class CachingHailStone {
3     public void run() {
4         HashMap<Integer,Integer> cache = new HashMap<
              Integer,Integer>();
5         // Complete the Logic [5 Points]
6         System.out.println("Hailstone(" + i + ") requires
              " + nSteps + " steps");
7     }
8     /**
9      * Returns the number of steps in the hailstone
          sequence beginning
```

```
10      * at n. The cache parameter keeps track of all
            previously calculated
11      * chain lengths to speed up the computation.
12      *
13      * @param n The starting value
14      * @param cache A map of previously calculated step
            counts
15      * @return The number of steps in the hailstone
            sequence
16      */
17     private int countSteps(int n, HashMap<Integer,Integer>
            cache) {
18
19         // Complete the Logic [5 Points]
20         return nSteps;
21     }
22
23     public static void main (String args[]) {
24         new CachingHailStone().run();
25     }
26 }
```

(b) **Analysing DNA:** Our sample DNA is made up of "bases" (the letters B, O, R, K). This can easily be represented as a String. [10]

For instance, the string ***"KRBOORBOKRB"*** is a valid string of our DNA. A ***k-mer*** is a **k-length** substring of bases in that DNA. Figure 4 illustrates all the 3-mers in this sequence:

```
"KRBOORBOKRB"
-------------
KRB
RBO
BOO
OOR
ORB
RBO
BOK
OKR
KRB
```

Figure 4: Figure for Question 2b

Conversely, in this example, **"BBB" is not a 3-mer because it is not a substring of this DNA sequence.** An example of a 4-mer in this string is "BOKR".

Write a method named **mostFrequentKmer** that returns the k-mer found most often in a given strand of DNA. The method accepts two parameters: the DNA string, and the value of k (an int). As an example, if we are given the DNA string

"BOBOK" and k=2, then the 2-mer found most often is the string "BO" because it appears twice, while other 2-mers ("OB", "OK") appear only once. If there is a tie for the k-mer found most often, you may return any one of the most frequent k-mers.

Constraint: **you can use only 1 data structure** (map, list, array).

3. Exception Handling, Generics and Lambdas. **Question 3: 15 pts**

(a) **Chained Exception** helps to identify a situation in which one exception causes another Exception in an application. [10]

For instance, consider a method which throws an **ArithmeticException** because of an attempt to divide by zero but the actual cause of exception was an I/O exception which caused the divisor to be zero.The method will throw the ArithmeticException to the caller. **The caller would not know about the actual cause of an Exception. Chained Exception is used in such situations.**

In our scenario of leave application, when we apply for leave the request goes to team lead then it goes to Manager. If the boss of manager is unhappy then he will raise an exception which will cause the manager to raise an exception. This will lead to the team lead raising an exception and our leave application will raise an exception to fail. Complete the code snippet given below to support the creation of 4 custom exceptions. Second, complete the methods given inside main() with appropriate logic to support chaining of exceptions. The points are distributed for various tasks as indicated in the code comments.

```
1  public class NoLeaveGrantedException {
2      // Modify to support custom exception class [1 Point]
3  }
4  public class TeamLeadUpsetException {
5      // Modify to support custom exception class [1 Point]
6  }
7  public class ManagerUpsetException {
8      // Modify to support custom exception class [1 Point]
9  }
10 public class BossOfManagerUpsetException {
11     // Modify to support custom exception class [1 Point]
12 }
13
14 public class LogWithChain {
15     //Modify and Complete to support chaining of
           Exceptions [1 Point]
16     public static void main(String[] args) __________ {
17         getLeave();
18     }
19
20     private static void getLeave() {
21         //Complete the Logic [1.5 Points]
```

```
22      }
23
24      private static void howIsTeamLead() {
25          // Complete the Logic [1.5 Points]
26      }
27
28      private static void howIsManager() {
29          //Complete the Logic [1 Point]
30      }
31
32      private static void howIsBossOfManager() { // Modify
            as needed [1 Point]
33          throw new BossofManagerUpsetException("Boss of
                manager is in bad mood");
34      }
35  }
```

(b) What are constructor references in Java? Discuss how constructor references to generic classes can be defined and used with the help of an example code-snippet. [5]

****

(Don't let fear get the better of you. All the best!)