**SUTD 50.001 Introduction to Information Systems and Programming**

**Problem Set 2A**

|  |
| --- |
| **Note:**   * **For all questions, please access the vocareum link found at eDimension for the starter code and to submit.** * **The Vocareum link is for submission only. Please work on the problems in Android studio, and this includes writing code for the test cases.** * **To prevent hard-coding, test cases used in Vocareum *may* be different from those provided here and will not be given to you.** |

1. [10 points] Title: Comparable interface

Given the following Octagon class, which represents a regular octagon (all sides equal):

**public class** Octagon {  
 **private double side**;  
 **public** Octagon(**double** side){  
 **this**.**side** = side;  
 }  
 **public double** getSide() {  
 **return side**;  
 }  
}  
  
Modify this **Octagon** class to implement the **Comparable<Octagon>** interface to allow sorting of **Octagon** objects based on their perimeters.

An **Octagon** class implementing the **Comparable<Octagon>** interface allows a **List** of **Octagon** objects to be sorted by a natural ordering (in this case, the perimeter). This is demonstrated in the following Test code:

**public class** Test {  
 **public static void** main(String[] args) {  
 ArrayList<Octagon> l = **new** ArrayList<Octagon>();  
 l.add(**new** Octagon(2));  
 l.add(**new** Octagon(3));  
 l.add(**new** Octagon(1));  
 Collections.*sort*(l);  
 **for** (Octagon o:l)  
 System.***out***.println(o.getSide());  
 }  
}

Results:  
1.0  
2.0  
3.0

2. [10 points] Title: Comparator interface

Given the following Octagon class, same code and assumptions as previous question:

**public class** Octagon {  
 **private double side**;  
 **public** Octagon(**double** side){  
 **this**.**side** = side;  
 }  
 **public double** getSide() {  
 **return side**;  
 }  
}

Implement a **OctagonComparator** class to implement the **Comparator<Octagon>** interface to allow sorting of **Octagon** objects based on their perimeters.

In this case, the **Octagon** does not implement any interface. You design a separate class called **OctagonComparator** which implements the **Comparator<Octagon>** interface. The **OctagonComparator** class specifies how a **List** of **Octagon** objects is sorted (in this case, the perimeter) and an object of this class is passed to the **Collections.sort()** method. This is demonstrated in the following Test code:

**public class** TestOctagonComparator {  
 **public static void** main(String[] args) {  
 ArrayList<Octagon> l = **new** ArrayList<Octagon>();  
 l.add(**new** Octagon(2));  
 l.add(**new** Octagon(3));  
 l.add(**new** Octagon(1));  
 Collections.*sort*(l, **new** OctagonComparator());  
 **for** (Octagon o:l)  
 System.***out***.println(o.getSide());  
 }  
}

Results:

1.0  
2.0  
3.0

3. [Total: 20 points]

Use the Observer design pattern to develop an air pollution alert system that sends the air pollution index to all the subscribed users if the air pollution index is more than 100.

The starting code has been provided. Modify AirPollutionAlert class to implement interface Subject. You only need to modify AirPollutionAlert class according to the Observer design pattern. Do not modify Subscriber class.

If your implementation of Observer Design Pattern is correct, you should see following results.

Example test code:

public class Test{

public static void main(String[] args) {

AirPollutionAlert singaporeAlert = new AirPollutionAlert();

Subscriber man = new Subscriber("man",singaporeAlert);

Subscriber simon = new Subscriber("simon", singaporeAlert);

singaporeAlert.setAirPollutionIndex(200);

singaporeAlert.setAirPollutionIndex(50);

singaporeAlert.setAirPollutionIndex(120);

singaporeAlert.unregister(man);

singaporeAlert.setAirPollutionIndex(300);

}

}

Output:

man received notification: 200.0

simon received notification: 200.0

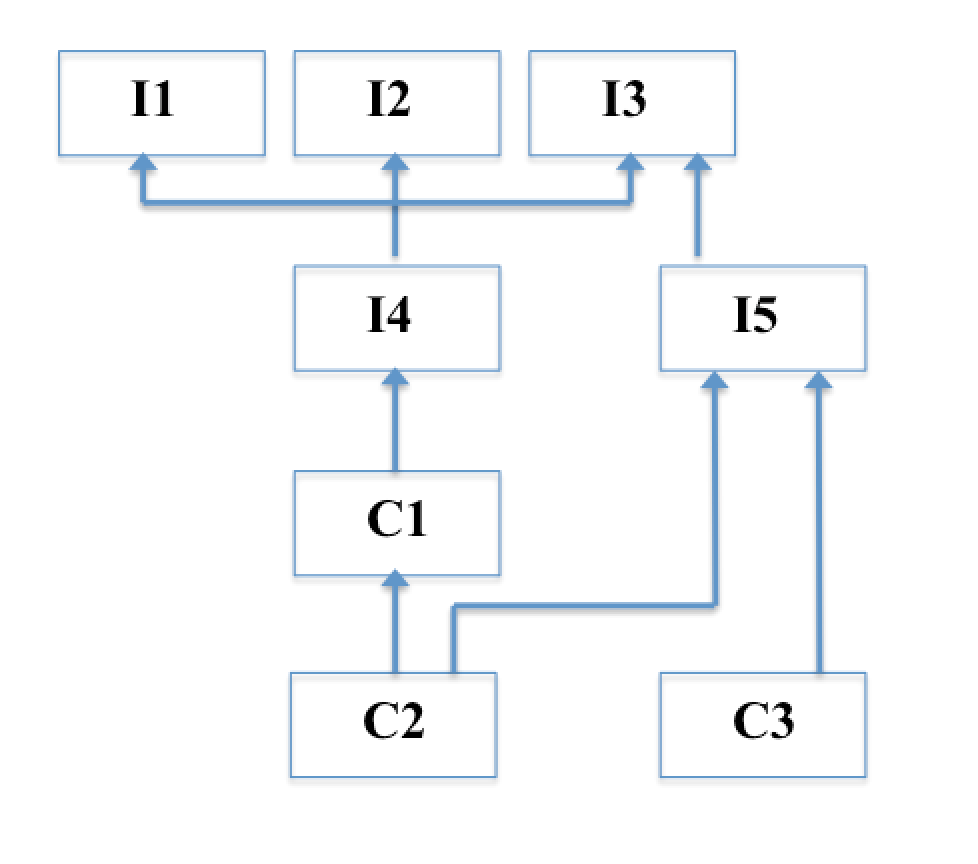
man received notification: 120.0

simon received notification: 120.0

simon received notification: 300.0

4. [Total: 20 points]

Develop the software using inheritance and interfaces as shown in the following figure. Note that an arrow from P to Q means that P is a subclass/sub-interface of Q, or P implements Q.



Interface I1 contains a method int p1();

Interface I2 contains a method int p2();

Interface I3 contains a method int p3();

Interface I4 contains a method int p4(); Note that I4 also inherits accessible programming constructs from I1, I2, I3, as shown in the figure.

Interface I5 contains a method int p5();

Class C1 contains only an abstract method int q1(); Note that C1 also implements I4 as shown in the figure.

Class C2 is a concrete class and provides all the needed implementations of methods. All the implementations simply return 0.

Class C3 is a concrete class and provides all the needed implementations of methods. All the implementations simply return 0.

5. [20 points] Title: Palindrome

https://en.wikipedia.org/wiki/Palindrome

Write a recursive method for finding palindromes. The method isPalindrome() returns true if the input string is a palindrome, and false otherwise.

Example outputs:

abba - true

adbcba - false

ZZaZZ - true

123421 - false

Note: no credit will be given for non-recursive code.

6. [Total: 30 points]

Permutation: Write a **recursive** method ‘permute()’ that computes all possible orderings of the characters in a string, i.e., all permutations that use all the characters from the original string.

For example, given the string ‘hat’, you code should produce the strings:

‘tha’, ’aht’, ’tah’, ’ath’, ’hta’, ’hat’

You can assume that the characters are distinctive and there is no repeated character.

No credit will be given for non-recursive code.

The list of permutations can be in any order.

**public class** Permutation {  
 **private final** String **in**;  
 **private** ArrayList<String> **a** = **new** ArrayList<String>();  
 *// additional attribute if needed* Permutation(**final** String str){  
 **in** = str;  
 *// additional initialization if needed* }  
  
 **public void** permute(){  
 *// produce the permuted sequences of 'in' and store in 'a', recursively* }  
  
 **public** ArrayList<String> getA(){  
 **return a**;  
 }  
}

**public class** TestPermutation {  
 **public static void** main(String[] args) {  
 ArrayList<String> v;  
  
 Permutation p = **new** Permutation(**"hat"**);  
 p.permute();  
 v = p.getA();  
 System.***out***.println(v);  
}  
}

Output:

[hat, hta, aht, ath, tha, tah]

7. **Question 7 is deleted.**

8. [10 points] (ArrayIndexOutOfBoundsException) Write a method:

**public static** String tstException(int idx, String[] y)

Given the index of the array, the method returns the corresponding element value. If the specified index is out of bounds, the method returns the message “Out of Bounds”.

Use try-catch to implement the method as a means of getting familiar with the syntax.

Recall that exceptions are for events that are generally out of the control of the programmer e.g. file not found.

In this case, notice that you can simply use an if/else to check the value of idx.   
Thus, a good coding practice is actually not to use try-catch in this question.

**public class** TestException {  
 **public static void** main(String[] args) {  
 String[] in = {**"hello"**, **"good night"**, **"good morning"**};  
  
 String ret = *tstException*(2, in);  
  
 System.***out***.println(ret);  
  
 ret = *tstException*(-1, in);  
  
 System.***out***.println(ret);  
 }  
  
  
 **public static** String tstException(**int** idx, String[] y) {  
 // to be implemented  
   
 }  
}

**Output:**

good morning

Out of Bounds