Name:

Student ID: **Cohort:**

ISTD 50.001 Mid-term

Date: 2-Nov, 2018 (Friday) 2.00pm to 4.00pm (2hrs)

- Write your name and ID at the top of this page.
- Answer all 7 questions. Total 100 points.
- For questions 1-5, submit your solutions to Vocareum.
- For questions 6-7, write your answers on this exam paper.
- All answers will be manually graded. You may be able to earn partial credit for questions.
- **You are not allowed to use any Internet accessing or communicating device during the quiz.**
- **You are not allowed to consult anyone inside or outside of the classroom other than the course instructors of 50.001.**
- This exam is open book. You may refer to the course slides / notes and your personal notes.
- You can use Android Studio / other IDE to test your programs.
- Good luck!

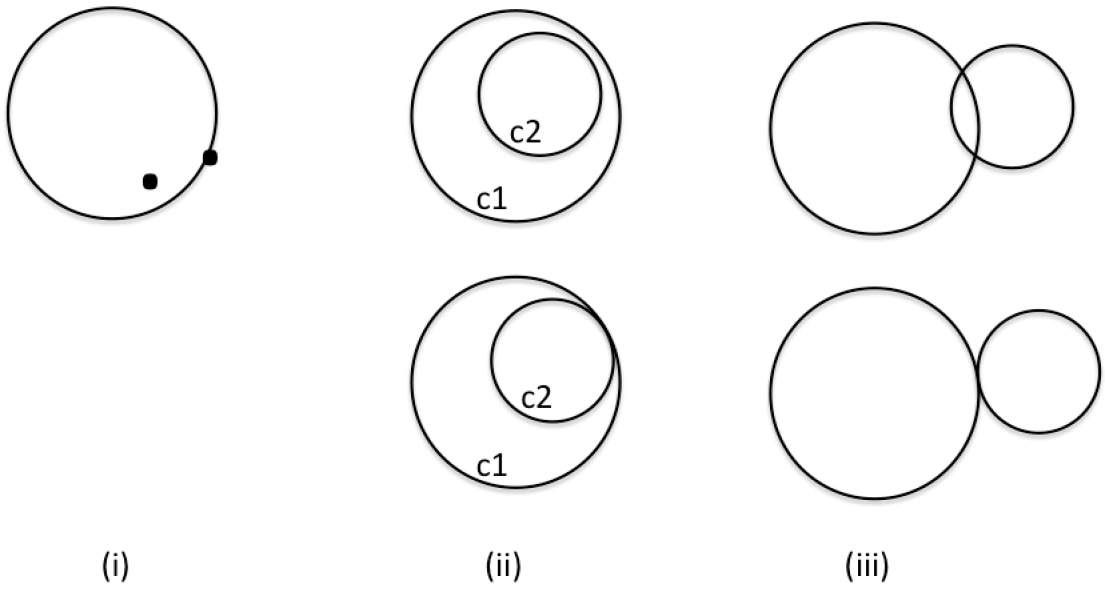
6	
7	
SubTotal	

Q1. [10 points]

Circle class

Define the **Circle** class that contains:

- Two private double variables named **x** and **y** that specify the center of the circle.
- The methods **getX()** and **getY()** to return **x** and **y** of the center respectively.
- A double private variable **radius**.
- A method **getRadius()** to return **radius**.
- A no-argument constructor that creates a default circle with (0,0) for center (x,y) and 1 for radius.
- A constructor that creates a circle with the specified center (x,y) and radius.
- A method **getArea()** that returns the area of the circle.
- A method **getPerimeter()** that returns the perimeter of the circle.
- A method **contains(double px, double py)** that returns true if the specified point (px,py) is inside this circle (see Figure below).
- A method **contains(Circle c)** that returns true if the specified **Circle c** is inside this circle (see Figure below).
- A method **overlaps(Circle c)** that returns true if the specified **Circle c** overlaps with this circle (see Figure below).



(i) The points are inside the circle; (ii) Circle c1 contains c2; but Circle c2 does not contain c1; (iii) Circle overlaps with another one.

Test code:

```
public class TestCircle {
   public static void main(String[] args) {
      Circle c = new Circle();

      System.out.println(c.getArea());
      System.out.println(c.getPerimeter());

      System.out.println(c.contains(1,0));

      System.out.println(c.contains( new Circle(0.5, 0, 0.3)));

      System.out.println(c.overlaps(new Circle(0,2,0.5)));

      Circle c2 = new Circle(1,1,1);
      System.out.println(c.getArea());

   }
}
```

Return:

```
3.141592653589793
6.283185307179586
true
true
false
3.141592653589793
```

Q2. [10 points]

Comparable and Comparator interfaces

Design a class named **Account** that contains:

- Private String data field (variable) named **id** for the account identity
- Private double data field named **balance** for the account
- Private Date data field named **dateCreated** that stores the date when the account was created
- A constructor that creates an account with the specified **id** and initial **balance**
- The accessor method for **id**
- The accessor method for **balance**
- The accessor method for **dateCreated**
- Override the method **toString()** to return a description of the account (see example code)
- Implement the **Comparable<Account>** interface to allow sorting of **Account** objects based on their balance (see example code)

In addition, design an **AccountComparator** class to implement the **Comparator<Account>** interface to allow sorting of Account objects based on their **id**. In particular, the Account **id** are sorted in alphabetical order. You can assume all characters in **id** are in lower case.

Hint: String is comparable.

Test code:

```
import java.util.ArrayList;
import java.util.Collections;

public class TestAcc {

   public static void main(String[] args) {

      Account a = new Account("simon", 20);
      System.out.println(a.getId());
      System.out.println(a.getBalance());
      System.out.println(a);

      ArrayList<Account> l = new ArrayList<>();
      l.add(new Account("man", 30));
      l.add(new Account("eric", 100));
      l.add(new Account("norman", 10));

      Collections.sort(l);
      System.out.println(l);

      Collections.sort(l, new AccountComparator());
      System.out.println(l);
   }
}
```

Output:

simon
20.0
Account:simon
[Account:norman, Account:man, Account:eric]
[Account:eric, Account:man, Account:norman]

(Note that there is no space in “Account:simon”)

Q3. [10 points]

Observer design pattern

Use the Observer design pattern to develop a temperature alert system that sends the temperature alert to all the subscribed users if the temperature is above 35 degrees or below 10 degrees.

The starting code has been provided. According to the Observer design pattern:

- Modify and complete the interface **Subject**
- Modify and complete the **TemperatureAlert** class to implement interface **Subject**
- Modify and complete the **Student** class to implement interface **Observer**, such that you obtain the outputs below
- Modify and complete the **Fish** class to implement interface **Observer**, such that you obtain the outputs below

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

```
public class TestTemperatureAlert {
   public static void main(String[] args) {
      TemperatureAlert westCoast = new TemperatureAlert();
      Student s1 = new Student("s1", westCoast);
      Student s2 = new Student("s2", westCoast);

      westCoast.setTemperature(40);
      westCoast.setTemperature(25);
      westCoast.setTemperature(5);

      westCoast.unregister(s1);
      Student s3 = new Student("s3", westCoast);
      Fish f1 = new Fish("f1", westCoast);

      westCoast.setTemperature(2);

   }
}
```

Output:

```
Student s1 receives temperature alert: 40
Student s2 receives temperature alert: 40
Student s1 receives temperature alert: 5
Student s2 receives temperature alert: 5
Student s2 receives temperature alert: 2
Student s3 receives temperature alert: 2
Fish f1 receives temperature alert: 2
```

Q4. [Total: 10 points]
Recursion

a) [3 points] Fibonacci

Write a recursive method, **recurFib(int idx)**, to calculate the Fibonacci number for a given index. For example, **recurFib(3)** returns an integer value 2, **recurFib(5)** returns an integer value 5 and **recurFib(10)** returns an integer value 55. Note: No points will be given for a non-recursive method.

```
public class Fib {

   public static void main(String[] args) {
      System.out.println(recurFib(3));
      System.out.println(recurFib(5));
      System.out.println(recurFib(10));
   }

   public static int recurFib(int idx) {
     //TODO: implement recursive method

   }
}
```

Output:

```
2
5
55
```

b) [7 points] Compute all sequences

Given a set of unique characters and an integer k >=1, write a recursive method to compute all the possible sequences of length k that can be formed from the given set of characters. Return the set of all possible seqeunces in an ArrayList<String>, see example below. Note: No points will be given for a non-recursive method.

```
public class AllSeq {
   public static void main(String[] args) {
      String[] s = {"p", "q"};
      System.out.println(compAllSeq(s, 3));
      String[] s2 = {"1", "2", "3", "4"};
      System.out.println(compAllSeq(s2, 1));
   }

   public static ArrayList<String> compAllSeq(String[] s, int k){
      //TODO: implement a recursive method to return all possible sequences of length k
   }
}
```

Output:

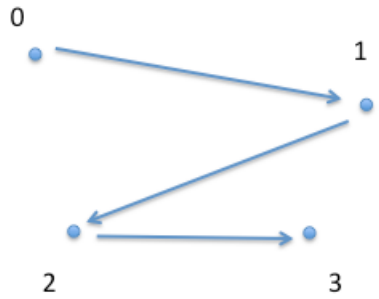
```
[ppp, qpp, pqp, qqp, ppq, qpq, pqq, qqq]
[1, 2, 3, 4]
```

Q5. [20 points]

Strongly Connected Graph

Computer networks are often modeled as graphs. A graph to a theoretician is a set of vertices and has edges connecting those vertices. A vertex might represent, for example, a computer on a network, while an edge may represent a physical or logical connection between two computers. If a graph is directed, then each edge has a specific direction. An example of a directed graph is shown below.

It is often useful to study the properties of graphs so that we can understand the properties of the real-world systems (e.g. networks) that the graph models. One property that is interesting to study is whether a graph is connected, which means that there is a path between every pair of nodes. In a directed graph, we must be more precise in what we mean by "connected." A "strongly connected" directed graph is a graph in which there is a directed path between every pair of vertices in the graph. A directed path follows edges by respecting the direction of those edges.



Design a method **testStronglyConnected( )** to test if a given directed graph is strongly connected or not. The method return **true** if the directed graph is strongly connected, **false** if the graph is not strongly connected.

```
public class StronglyConnected {

   static boolean testStronglyConnected(int nodecount, int linkcount, ArrayList<Integer> listOfLink)
   {
      //TODO
   }

}
```

Here, **nodecount** is the number of nodes (vertices) in the graph, **linkcount** is the number of links (edges), and each pair of Integer x, y in **listOfLink** is a directed edge from x to y. For example, the above graph can be represented and passed to the method as follows:

```
public class TestStronglyConnected {
   public static void main (String[] args){
      int nodecount=4;
      int linkcount=3;
      ArrayList<Integer> listOfLink = new ArrayList<Integer> ( Arrays.asList(0,1,  1,2,  2,3));
      System.out.println(StronglyConnected.isStronglyConnected(nodecount, linkcount, listOfLink));
   }
}
```

return:
false

The graph is not strongly connected because not every pair of vertices is connected by a directed path. For example, there is no directed path to go from vertex 3 to vertex 2.

Another test case:

```
public class TestStronglyConnected {
   public static void main (String[] args){
      int nodecount=5;
      int linkcount=5;
     ArrayList<Integer> listOfLink = new ArrayList<Integer> ( Arrays.asList(0,1,  1,2,  2,3,  3,4,  4,0));
     System.out.println(StronglyConnected.isStronglyConnected(nodecount, linkcount, listOfLink));
   }
}
```

return:
true

Q.6 [25 points]

Static and dynamic checking, OOP

a) [5 points] This question is about error checking in a programming language. As explained during lessons, there are three ways that programming errors are caught (i) by static checking (ii) by dynamic checking (iii) they are not caught by the compiler and the system. Consider the following code.

```
public class Checking {

   public static void main(String[] args){


   }

   public static void someMethod(int a, int b){

      System.out.println("Maths:" + a + " divided by " + "b = " + a/b );

   }

}
```

- Explain what is *static checking.* Give an Java statement that, when written in main() above, has error(s) that can be caught by static checking.

- Explain what is *dynamic checking.* Give a Java statement that, when written in main() above, has error(s) that can be caught by dynamic checking.

- Give a Java statement that, when written in main() above, has error that is not caught by static checking nor dynamic checking.

b) [5 points] Explain how Java syntax generics (e.g., ArrayList<String>) helps to eliminate errors from code. Provide a Java code snippet to illustrate your answer.

c) [5 points] Explain the use of Java keyword '**this**' and '**super**' as a method call, e.g. **this()** or **super()**. How are they different? You may provide Java code snippets to illustrate your answer.

d) [5 points] A Java programmer may prefer to write an abstract class with abstract methods. Explain the advantage of using abstract class and abstract method in Java. You may provide Java code snippets to illustrate your answer.

e) [5 points] Can an abstract class still define its constructors? What is the usage of constructors of an abstract class? You may illustrate your answer with Java code snippets.

Q.7 [15 points]

Boolean Satisfiability

a) [5 points] Explain what it means for a propositional formula of Boolean variables to be in the conjunctive normal form (CNF)? Provide one example of CNF.

b) [10 points] Is the following propositional formula of Boolean variables A, B, C, D satisfiable? Circle your answer, and give detail explanation and derivation. No credit will be given if explanation / derivation is not provided.

(¬D) ∧ **(¬A** ∨**B)** ∧ **(¬B** ∨**C)** ∧**(¬C** ∨**D)** ∧ **(A)**

Your Solution:

a. Satisfiable

b. Not satisfiable

Explanation / Derivation:

End of Paper