Mid-Semester Examination (R) – Object Oriented Programming (CS F213)
First Semester: 2023-24
Dr. Asish Bera & 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 **4 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 Generics (Recommended Time → 30 minutes)** **Question 1: 25 pts**

   (a) Complete the given code skeleton of a generic ***max*** method that computes the greatest value in a collection of elements of an unknown type. [05]

```
1 class Collections {
2   public static max(______xs) { // Adapt appropriately [01
        point]
3     Iterator xi = xs.iterator();
4     // Complete the code [04 points]
5     return _____;
6   }
7 }
```

   (b) Explain the working of the following code snippet. Would this work? Is the logic correct? Give justifications and if necessary alter the code as a work-around. [08]

```
class ObjectStore<T extends Comparable<T> & Comparable<
    String> > {
  private Set<T> theObjects = new TreeSet<T> ();
  ...
  public boolean equals(ObjectStore<String> other) {
    if (theObjects.size() != other.size()) return false;
    Iterator<T> iterThis = theObjects.iterator();
    Iterator<String> iterOther = other.theObjects.iterator
       ();
    while (iterThis.hasNext() && iterOther.hasNext()) {
      T t = iterThis.next();
      String string = iterOther.next();
      if ( t.compareTo(string) != 0)  return false;
    }
    return true;
  }
}
```

(c) Explain with an example why the below code snippet would fail. Modify it to eliminate the error and explain your solution. [04]

```
public class Collections {
  public static <T> void copy( List<T> dest, List<T> src)
     {
    for (int i=0; i<src.size(); i++)
      dest.set(i,src.get(i));
  }
}
```

(d) Write short notes with suitable examples (if relevant) for the following. [06]

1. Reification **[02 points]**
2. Bridge Methods **[02 points]**
3. Type Erasure **[02 points]**

(e) If we have to declare a method called swap then which of the given signature would you use? What are the implications of both? Explain with relevant code examples to illustrate your scenario. [02]

```
public static <E> void swap(List<E> list, int
   i, int j);
public static void swap(List<?> list, int i,
   int j);
```

2. Java Collections & Lambdas (Recommended Time → 25 min.) **Question 2: 15 pts**

(a) Study the given code snippet. Find errors, fix them and complete the logic for the methods of the class. [06]

```java
public class Stack<E> {
    // Find errors and fix them -- [01 point]
    private E[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY =
       16;

    public Stack() {
    elements = new E[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(_______) { // Check signature and
       implement [01 point]
        // Implement the logic
    }

    public E pop(_______) { // Check signature and
       implement [01 point]
        // Implement the logic
        return result;
    }

    // a method that takes a sequence of elements and
    // pushes them all onto the stack
    public void pushAll(_______) { // Check signature and
       implement [01 point]
        // Implement the logic
    }

    // popAll method pops each element off the stack and
       adds the elements to the given
    // collection
    public E popAll(_______) { // Check signature and
       implement [01 point]
        // Implement the logic
    }

    public boolean isEmpty() { // Check signature and
       implement [01 point]
    }
}
```

(b) Modify the following **Date** class so that it implements the interface **Comparable<T>**. [04]
The ordering on objects of type Date should be the *natural, chronological ordering.*

```java
class Date {

```

```
3     int month; // Month number in range 1 to 12.
4     int day; // Day number in range 1 to 31.
5     int year; // Year number.
6     Date(int m, int d, int y) { // Convenience constructor
          .
7         month = m;
8         day = d;
9         year = y;
10    }
11 }
```

(c) The below table lists some of the important functional interfaces in Java. Write [03]

Interface Name	Arguments	Return Type
Predicate<T>	T	boolean
Consumer<T>	T	void
Function<T,R>	T	R
Supplier<T>	None	T
BinaryOperator<T>	(T, T)	T
UnaryOperator<T>	T	T

suitable lambda code snippets for the following Functional interfaces

1. Predicate<T> **[01 point]**
2. BinaryOperator<T> **[01 point]**
3. Function<T,R> **[01 point]**

(d) Consider a debug() inside a Logger class whose usage is given below: [02]

```
1             Logger logger = new Logger();
2             if (logger.isDebugEnabled()) {
3                 logger.debug("Look at this: " +
                      expensiveOperation());
4             }
```

Alter the method implementation of debug() to support lambda-based invocation like:

```
1         Logger logger = new Logger();
2         logger.debug(() -> "Look at this: " +
              expensiveOperation());
```

3. **Java Fundamentals (Recommended Time → 20 minutes)** **Question 3: 10 pts**

(a) Write a class called **Spaceship** that models some characteristics of a Spaceship. Specifically, a Spaceship keeps track of the amount of food on board (in pounds), a list of names of visited planets (in order of visit), and the name of each crew member on board, as well as how much food (in pounds) each crew member consumes each day. Note that you should store the crew members and their food consumption in a HashMap; not doing so will result in a substantial deduction. You create a Spaceship by specifying the amount of food initially on board, like [05]

```
Spaceship myShip = new Spaceship(50); // 50 pounds of food initially
```

A Spaceship should have the following public methods:

```
/** Boards a crew member with the given food intake. This crew member will
now consume food during trips. */
public void board(String crewMemberName, int foodPerDay);
/** Unboards a crew member with the given name from the ship. This crew
member is no longer on the ship and no longer consumes food. */
public void unboard(String crewMemberName);
/** Returns a String of visited planets, in order of visit. The string
should be formatted like "[Earth, Mars, Venus]" */
public String getPlanetsVisited();
/** Attempts to fly to a planet, which takes the given number of days. */
public boolean flyTo(String planetName, int daysRequired);
```

The most involved public method is **flyTo**. This method should do the following:

- It should first calculate if there is enough food on board to feed all crew members for the number of days required to get to this planet.
- If there is enough food, it should update the amount of food onboard to reflect that the ship has travelled to this planet, and it should add this planet to its list of visited planets. The method should also return true to indicate the trip was successful.
- If there is not enough food, then the method should return false to indicate the trip failed. It should not modify either the onboard food or the list of visited planets.

As an example, let's say on myShip we have onboard Nolan, who consumes 3 pounds of food daily, and Nick, who consumes 4 pounds of food daily. myShip.flyTo("Venus", 7) should return true because Nick and Nolan consume 7 pounds of food per day in total, which over 7 days is 7*7=49 pounds of food (and our ship has 50). We should

therefore add "Venus" to our list of visited planets, and set the food onboard to now be 50-49=1.

On the other hand, myShip.flyTo("Mars", 10) should return false because we require 7*10=70 pounds of food to get to Mars, but our ship only has 50. We should not add "Mars" to our list of visited planets, nor change our onboard food. Use the given solution snippet:

```
1  public class Spaceship {
2      // The number of pounds of food on board
3      private int foodOnBoard;
4      // A map from crew member names to the pounds of food
          they eat per day
5      private HashMap<String, Integer> crewMemberMap;
6      // A list of visited planet names, in order of visit
7      private ArrayList<String> planetsVisited;
8      // Fix the signature and complete the code
9      public Spaceship(______) {
10         // COmplete the code snippet [01 point]
11     }
12     /* Boards a crew member with the given food intake.
          This
13     * crew member will now consume food during trips.
14     */
15     public void board(String crewMemberName, int
          foodPerDay) {
16         // Complete the code snippet [0.5 point]
17     }
18     /* Unboards a crew member with the given name from the
           ship. This
19     * crew member is no longer on the ship and no longer
          consumes food.
20     */
21     public void unboard(String crewMemberName) {
22      // write the logic [0.5 point]
23     }
24     /* Returns a String of visited planets, in order of
          visit. The string
25     * should be formatted like "[Earth, Mars, Venus]"
26     */
27     public String getPlanetsVisited() {
28         return planetsVisited.toString();
29     }
30     /* Attempts to fly to a planet, which takes the given
          number of days.
31     * Returns true if we had enough food, and false
          otherwise.
32     */
```

```
33      public boolean flyTo(String planetName, int
           daysRequired) {
34      // write the logic [3 points]
35  }
```

(b) Write a method called **visitablePlanets** that returns a list of planet names that we could visit given a certain amount of food. Specifically, the method accepts three parameters: the amount of food our spaceship starts with (int), a map of names of crewmembers to the amount of food they each consume daily, and a map from planet names we want to visit to the number of travel days required to visit that planet. You should first create a new Spaceship with the given amount of food. Then, you should use the public methods of Spaceship to board all crew members and attempt to travel to the planets in the provided planet map. When iterating over the planet map's keys, you should assume that the keys (planet names) are in the order in which you want to travel to the planets. You should return the list of planets that the Spaceship can travel to given the amount of food it starts with. [05]

For example, if the parameters had the values

```
1               startingFood = 50,
2               crewMap = {Rishi=2, Guy=2, Aleksander=3},
3               planetMap = {Venus=2, Mars=3, Saturn=10}
```

then **visitablePlanets** should return "[Venus, Mars]" because we have enough food to get to Venus (7*2 = 14 pounds required) and then Mars (another 7*3 = 21 pounds required) but not Saturn after that (we have only 50-14-21=15 pounds remaining, but need 7*10=70).

4. **Inheritance in Java (Recommended Time → 15 minutes.)** **Question 4: 10 pts**

(a) Please refer to Figure 1 to understand the hierarchy of classes and interfaces. The interface **Parent** provides an abstract method welcome () with the signature: **void welcome (String msg)**. Both the interfaces provide default implementation for the abstract method. **Tasks:** [10]

1. Provide code-snippet to for the entire hierarchy. **[07 points]**
2. Draw a tabular structure and show which version of welcome() gets invoked from each of the classes. Justify your answer. **[03 points]**

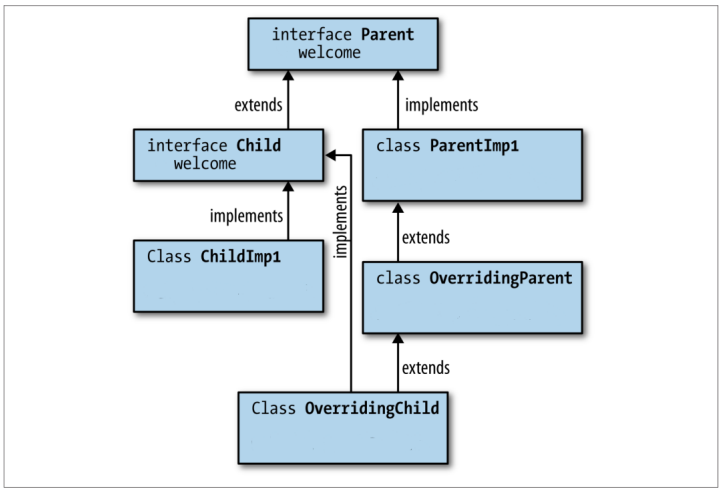

Figure 1: Figure for Question 4b

Instruction for writing answers

Please answer sub-parts of the same question sequentially. For example, if you start with Question 1, then please answer Question 1 (a), (b), (c) and so on before moving to a different question, for instance, Question 2.

****

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