Family Name: ..... **Other Names:** .....

Student ID: ..... **Signature** .....

COMP 103 : Test 2

2022, Jan 26 ** WITH SOLUTIONS **

Instructions

- Time allowed: **50 minutes**
- Attempt **all** questions. There are 30 marks in total.
- Write your answers in this test paper and hand in all sheets.
- If you think some question is unclear, ask for clarification.
- Brief Java documentation is provided with the test
- This test contributes 10% of your final grade
- You may use dictionaries.
- You may write notes and working on this paper, but make sure your answers are clear.

Questions	Marks	
1. Using Queues	[10]	
2. Using Collections	[10]	
3. CompareTo, Equals, and HashCode	[10]	
	TOTAL:	

Question 1. Using Queues **[10 marks]**

This question is about a program that a company needs to help their delivery coordinator keep track of the boxes to be delivered.

The program uses a queue of Boxes that are waiting for delivery.

When the warehouse staff have boxed up the items for a customer request, the box is weighed and taken to the delivery area. The coordinator uses the program to record the box, with its boxID and weight.

When a delivery van is ready for a new trip, boxes are loaded into the van **in the order that they arrived at the delivery area**.

Each van has a maximum total weight capacity. The coordinator uses the program to print out a list of the boxIDs to be loaded that will not overload the van. No box will weigh more than a single van capacity.

Part of the program is given below. You are to complete two methods.

```
public class Box{
    private int ID;
    private double weight;

    public Box(int id, double wt){
        this.ID = id;
        this.weight = wt;
    }
    public double getWeight(){
        return weight;
    }
    public String toString(){
        return String.format("#%d (%.1fkg)", ID, weight);
    }
}
```

```
import java.util.*;
import ecs100.*;

public class DeliveryManager{
    private Queue<Box> deliveryQueue = new ArrayDeque<Box>(); // The queue of boxes waiting for delivery

    public static void main(String[] args){
        new DeliveryManager().setupGUI();
    }
    public void setupGUI(){
        UI.addButton("New Box", ()->{
                            int id = UI.askInt("ID");
                            double wt = UI.askDouble("Weight");
                            recordBox(id, wt);                          // for question (a)
                        });
        UI.addTextField("Van weight", (String w)->{
                            double maxCapacity = Double.valueOf(w);
                            chooseBoxesForDelivery(maxCapacity);    // for question (b)
                        });
    }
    ...
```

(Question 1 continued)

(a) **[4 marks]** Complete the recordBox(int ID, double weight) method to record a new Box on the delivery queue, and print out the current number of Boxes waiting to be delivered. The parameters are the id and the weight of the new box.

```
public void recordBox(int id, double weight){
    Box box = new Box(id, weight);
    deliveryQueue.offer(box);
    UI.println("queue has "+ deliveryQueue.size() + " items");









}
```

(b) **[6 marks]** Complete the chooseBoxesForDelivery(double capacity) to remove boxes from the delivery queue to be loaded into a van, and print out all the id numbers. The total weight of the boxes must not exceed the specified capacity.

```
public void chooseBoxesForDelivery(double capacity){
    UI.println("Deliver these boxes");
    double total = 0;
    while (!deliveryQueue.isEmpty() && total+deliveryQueue.peek().getWeight()<capacity){
        Box b = deliveryQueue.poll();
        total += b.getWeight();
        UI.println(b);
    }









    UI.println("---------");
}
```

Question 2. Using Collections **[10 marks]**

This question concerns a program to help an office administrator manage the tasks requested by people in the organisation.

When the administrator receives a new task, they must first check that it is a valid task for the office.

The administrator then allocates a priority to the task (a number from 1 to 5), and puts the task on a priority queue.

When a staff member becomes available, the adminstrator allocates them the highest priority task from the queue.

The program first reads the valid task categories from the file "`taskCategories.txt`" and storing them in an appropriate collection. *The file has one category (a word) per line.*

You must complete the TaskManager class on the next page:

- the field declarations with the collections (one field called "tasks" and one field called "categories"),
- the loadCategories method, which will load the file "taskCategories.txt" into the collection of valid categories (use "Files.readAllLines"),
- the receiveTask method, which is given a new task with a category, but no priority, and checks if it is a valid category for the office.
  If so, it asks the user for a priority, sets the priority of the task, and adds it to the priority queue.
  Otherwise print out a message containing the invalid category

You should use the Task class to represent tasks, and you should ensure that receiveTask is efficient, even if the collection of categories is very large.

```
class Task implements Comparable<Task>{ // comparable based on priority
    private String category;
    private priority priorityLevel ;
    private long timeStamp;

    public Task(String cat){
        this.category = cat;
    }

    public void setPriority (int pri){
        this. priority = pri;
        this.timeStamp = System.currentTimeMillis();
    }
    public String getCategory(){
        return this.category;
    }

    public String toString(){
        return this.category +" (pri:"+this.priority+")";
    }
    :
}
```

(Question 2 continued on next page)

```
public class TaskManager{
    private Queue<Task> tasks = new PriorityQueue<Task>();                    //FIELDS
    private Set<String> categories = new HashSet<String>();



    public static void main(String[] args){
        TaskManager rm = new TaskManager();
        rm.loadCategories();
        UI.addButton("Receive", ()->{rm.receiveTask(rm.getNewTask());});
        UI.addButton("Allocate", rm::allocateTask);
    }
    public void loadCategories(){
        try{
            List<String> allWords = Files.readAllLines(Path.of("taskCategories.txt"));
            for (String wd : allWords){
                categories.add(wd);
            }
        }
        catch(IOException e){UI.println("Fail: " + e);}









    }
    public void receiveTask(Task task){

        if (categories.contains(task.getCategory())){
            task.setPriority(UI.askInt("Enter priority"));
            tasks.offer(task);
        }
        else {
            UI.println(task.getCategory()+" is not a valid category");
        }









    }
    public void allocateTask(){ // Removes and prints highest priority task*/
        if (!tasks.isEmpty()){UI.println("Allocate "+tasks.poll());}
        else                 {UI.println("No tasks ");}
    }
```

Question 3. CompareTo, Equals, and HashCode **[10 marks]**

Suppose you are writing a program to keep track of the albums that you own. You have defined an Album class that contains several fields (given below) such as the title, artist, releaseYear, and trackListing.

(a) **[5 marks]** You would like to be able to sort a List that contains Album objects, and for the Album object to have a natural ordering.

Add the approprate interface declaration to the Album class so that it is comparable. Then, complete the compareTo(...) method for the Album class so that:

- Album will be sorted alphabetically by artist first, and then by release year.
- You can assume that none of the fields are null.

```
public class Album implements Comparable<Album>        {

    private String  title ;
    private String  artist ;
    private int  releaseYear ;
    private List<String> trackListing;

    public int compareTo(Album other) {
        if ( artist .equals(other. artist ) {
            return releaseYear  - other. releaseYear ;  // or equivalent
        }
        return artist .compareTo(other. artist );









    }
```

(Question 3 continued on next page)

(Question 3 continued)

(b) **[5 marks]** It is not sufficient to just define a compareTo(...) method by itself–we also need to define the equals(...) and hashcode() methods.

Complete the equals(...) and hashcode() methods for the Album class below.

- Two albums are equal if and only if they have the same title, artist, and release year.
- You can assume that neither title nor artist will be null.
- Ensure that the equals(...) and hashcode() methods are consistent:
  - with each other (if two albums are equal, they must have the same hash code), **and**
  - with the compareTo(...) method (if two albums are equal, compareTo must return 0)

```
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Album other = (Album) obj;
        if (! artist .equals(other. artist ))
            return false;
        if (! title .equals(other. title ))
            return false;
        return releaseYear == releaseYear;


    }

    public int hashCode() {
        int prime = 97;
        int result = 1;
        result = prime * result + artist .hashCode();
        result = prime * result + title .hashCode();
        result = prime * result + releaseYear;

        return result ;
    }


}
```

* * * * * * * * * * * * * * *

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Documentation

Brief, simplified specifications of some relevant Java collection types and classes.

Note: E stands for the type of the item in the collection.

```
interface Collection<E>
   public boolean isEmpty()                    // cost: O(1) for all standard collection classes
   public int size()                           // cost: O(1) for all standard collection classes
   public void clear()
   public boolean add(E item)
   public boolean contains(Object item)
   public boolean remove(Object element)

interface List<E> extends Collection<E>
   // Implementations: ArrayList
   public boolean isEmpty()
   public int size()
   public void clear()
   public E get(int index)                     // cost: O(1)
   public E set(int index, E element)          // cost: O(1)
   public boolean contains(Object item)        // cost: O(n)
   public void add(int index, E element)       // cost: O(n) (unless index is close to the end.)
   public E remove(int index)                  // cost: O(n) (unless index is close to the end.)
   public boolean remove(Object element)       // cost: O(n)

interface Set extends Collection<E>
   // Implementations: HashSet, TreeSet
   public boolean isEmpty()
   public int size()
   public void clear()
   public boolean add(E item)                  // cost: O(1) for HashSet, O(log(n)) for TreeSet
   public boolean contains(Object item)        // cost: O(1) for HashSet, O(log(n)) for TreeSet
   public boolean remove(Object element)       // cost: O(1) for HashSet, O(log(n)) for TreeSet

interface Map<K, V>
   // Implementations: HashMap, TreeMap
   public V get(K key)                 // cost: O(1) for HashMap, O(log(n)) for TreeMap
   public V put(K key, V value)        // cost: O(1) for HashMap, O(log(n)) for TreeMap
   public V remove(K key)              // cost: O(1) for HashMap, O(log(n)) for TreeMap
   public boolean containsKey(K key)   // cost: O(1) for HashMap, O(log(n)) for TreeMap
   public Set<K> keySet()              // cost: O(1)
   public Collection<V> values()       // cost: O(1)
   // (get returns null if the key is not present)
   // (get put and remove return the old value, if any)
```

```
interface Queue<E> extends Collection<E>
   // Implementations: ArrayDeque, LinkedList, PriorityQueue
   public boolean isEmpty()
   public int size()
   public void clear()
   public E peek()                  // cost: O(1) for ArrayDeque, LinkedList, O(1) for PriorityQueue
   public E poll()                  // cost: O(1) for ArrayDeque, LinkedList, O(log(n)) for PriorityQueue
   public boolean offer(E element)  // cost: O(1) for ArrayDeque, LinkedList, O(log(n)) for PriorityQueue
   // (peek and poll return null if the queue is empty)

class Stack<E> implements Collection<E>
   public boolean isEmpty()
   public int size()
   pubic void clear()
   public E peek()                  // cost: O(1)
   public E pop()                   // cost: O(1)
   public E push(E element)         // cost: O(1)
   // (peek and pop return null if the queue is empty)


class Collections
   public void sort(List<E> list);                          // cost = O(n log(n)), but O(n) if almost sorted
   public void sort(List<E> list, (E e1, E e2)->{...});     // cost = O(n log(n)), but O(n) if almost sorted
   public void swap(List<E> list, int i, int j);            // cost = O(1)
   public void reverse(List<E> list);                       // cost = O(n)
   public void shuffle(List<E> list);                       // cost = O(n)


interface Comparable<E> // All Comparable objects have a compareTo method:
   public int compareTo(E other);
          // returns
          //     -ve if this comes before other;
          //     +ve if this comes after other,
          //       0 if this and other are the same
```