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

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

COMP 103 : Test

2019, Sept 3 ** WITH SOLUTIONS **

Instructions

- Time allowed: **50 minutes**
- Attempt **all** questions. There are 50 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 18% of your final grade
  (But your mark will be increased to your exam mark if that is higher.)
- You may use dictionaries.
- You may write notes and working on this paper, but make sure your answers are clear.

Questions	Marks	
1. Properties of Collections	[12]	
2. Using Queues in Simulations	[13]	
3. Using Collections	[10]	
4. Cost of Algorithms	[7]	
5. Recursion	[8]	
	TOTAL:	

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.

Question 1. Properties of Collections **[12 marks]**

For these questions, circle "true" or "false" for each property. (Note some properties may be true for more than one type.)

(a) **[4 marks]** For a Map, state whether each property is true or false.

true/**false**	:	The order of values in the collection is important
true/false	:	To remove a value, you must specify its key
true/false	:	The collection cannot contain any duplicate keys
true/**false**	:	The collection records the number of copies of each value

(b) **[4 marks]** For a Stack, state whether each property is true or false.

true/false	:	The order of items in the collection is important
true/**false**	:	Items must have a natural ordering to be stored in the collection.
true/false	:	The first item added is the last item to be removed
true/false	:	The collection can contain duplicate items

(c) **[4 marks]** For a List, state whether each property is true or false.

true/**false**	:	An item can be only be removed by specifying its index
true/false	:	The order of items in the collection can be changed
true/**false**	:	The collection cannot contain any duplicate items
true/false	:	The time an item is added does not affect when it is removed

Marking:
4 marks for getting all four properties right;
2.5 marks for getting three properties right;
0 marks for less than three right.

Question 2. Using Queues in Simulations **[13 marks]**

For this question, you are to extend a simple simulation program.

The program simulates jobs in a factory being processed by a machine. The machine has a queue of Jobs waiting to be processed. On each time tick:

- a new Job may be created and added to the queue.
- the Job at the head of the queue is advanced one tick
- if the Job at the head of the queue is complete, it is removed from the queue.
- the queue is redisplayed.

You must extend the program for a factory with multiple machines, each with its own queue.

- There are five machines, named "Lathe", "Drill", "Cutter", "Sander", "Painter".
- Each Job contains a machine name; the Job should be added to the queue of the machine with that name. (The getMachineName() method returns the name.)
- On every time tick, the first job on every queue should be advanced.

Here is the one-machine program:

```
public class Simulation{
    private boolean running = false;

    private Queue<Job> queue;

    public void setup(){
        running = false;
        queue = new ArrayDeque<Job>();
    }

    public void run(){
        int time = 0;
        running = true;
        while (running){
            time++;
            if (Math.random()<0.2){
                Job j = new Job();
                queue.offer(j);
            }
            if (!queue.isEmpty()){
                Job topJob = queue.peek();
                topJob.advanceJobByOneTick();
                if (topJob.isCompleted()){
                    queue.poll();
                }
            }
            display(queue, 1);    // redisplay queue on row 1 of the window
        }
    }
}
```

(Question 2 continued on next page)

(Question 2 continued)

Extended simulation with multiple machines, each with its own queue:

```
public class Simulation{
    public static final String[ ] MACHINE_NAMES =
                    new String[ ]{"Lathe","Drill","Cutter","Sander","Painter"};
    private boolean running = true;

    private Map<String, Queue<Job>> queues;          //new

    public setup(){
        running = false;
        queues = new TreeMap<String,Queue<Job>>(); //new
        for (String name : MACHINE_NAMES){          //new
            queues.put(name, new ArrayDeque<Job>()); //new
        }
    }


    public void run(){
        int time = 0;
        running = true;
        while (running){
            time++;
            if (Math.random()<0.2){
                Job j = new Job();
                queues.get(j.getMachineName()).offer(j);        //new
            }
            int row = 1;
            for (Queue<Job> queue : queues.values()){           //new
                if (!queue.isEmpty()){
                    Job topJob = queue.peek();
                    topJob.advanceJobByOneTick();
                    if (topJob.isCompleted()){
                        queue.poll();
                    }
                }
                display(queue, row++);                          //new
            }
        }
    }




}
```

Question 3. Using Collections **[10 marks]**

Complete the following countTwice method which is given a List of words (Strings), and returns the number of words that occur exactly twice in the List.

The method should be efficient: if the List has n words, then the method should run in O(n) time.

Hint: this is similar to finding the mode of a list of numbers.

```
public int countTwice(List<String> words){
    Map<String, Integer> counts = new HashMap<String, Integer>();
    for (String word : words){
        if (counts.containsKey(word)){
            counts.put(word, counts.get(word)+1);
        }
        else {
            counts.put(word, 1);
        }
    }
    int ans = 0;
    for (String word : counts.keySet()){
        if (counts.get(word)==2){
            ans++;
        }
    }
    return ans;
//OR
    Set<String> seenOnce = new HashSet<String>();
    Set<String> seenTwice = new HashSet<String>();
    Set<String> seenLots = new HashSet<String>();
    for (String word : words){
        if (!seenOnce.contains(word)){
            seenOnce.add(word);
        }
        else if (!seenTwice.contains(word)){
            seenTwice.add(word);
        }
        else {
            seenLots.add(word);
        }
    }
    return seenTwice.size()-seenLots.size();









}
```

Question 4. Cost of Algorithms **[7 marks]**

What is the big-O cost of the following method that prints all possible matches of tennis players from a list of players?

- write the big-O cost of performing each line with a comment
- write the number of times each of the lines will be performed,
- write the total cost of the algorithm (big-O) at the bottom of the box

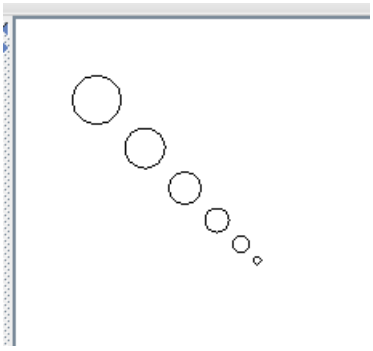
```
public void findMatches(List<Player> players){
    Collections.sort(players);                          // cost= O(nlog(n))   times= 1
    for (Player p1 : players){
        for (Player p2 : players){
            if (p1 != p2) {                             // cost= O(  1  )     times= n^2
                UI.println(p1.name()+":"+p2.name());    // cost= O(  1  )     times= n^2
            }
        }
    }                                                   Total Cost = O( n^2 )
}
```

Question 5. Recursion **[8 marks]**

(a) **[4 marks]** Complete the following recursive drawCircles(x, y, diam) method to draw a diagonal line of circles.

- You must use a "first-and-rest" recursion: drawCircles should draw the first circle and then call itself recursively to draw the rest of the circles.
- The first circle should be centered at (x,y) with diameter diam.
- The next circle should be diam pixels to the right and down from the first, and should be 5 pixels smaller.
- It should continue to draw circles as long as the diameter is larger than 3.

For example, drawCircles(100, 100, 30) should output the following:



```
public void drawCircles(double x, double y, double diam){
    if (diam > 3){
        double rad = diam/2;
        UI.drawOval(x−rad, y−rad, diam, diam);
        drawCircles(x+diam, y+diam, diam−5);
    }









}
```

Hint: you can use the method UI.drawOval(left, top, diam, diam) to draw a circle.

(Question 5 continued on next page)

(Question 5 continued)

(b) **[4 marks]** Suppose the variable letters contains the following List of 6 Strings:

"A"	"B"	"C"	"D"	"E"	"F"
0	1	2	3	4	5

What will mixUp(letters, 0, 5) print out?

```
/** mix up the letters in the list */
public void mixup(List<String> letters, int start, int end){
    if ( start <= end) {
        int mid = start + (end-start)/3;
        UI.print( letters.get(mid));
        mixup( letters, mid+1, end);
        mixup( letters, start, mid-1);
    }
}
```

Answer:

B D E F C A

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

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
```