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

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

COMP 103 : Test 3

2021, Jan 27

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 in Simulations	[10]	
2. Big-O	[10]	
3. Recursion	[10]	
	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. Using Queues in Simulations **[10 marks]**

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

The program simulates customers using the self-checkout machines at the local supermarket. Currently, the simulation only supports one self-checkout machine, which is modelled as a single queue.

- Customers all join the back of the queue.
- Each tick, the customer at the front of the queue processes one item from their basket.
- When all their items have been processed, the customer leaves the queue.

At each tick:

1. a new Customer may be created and added to the queue.
2. the Customer at the head of the queue is advanced one tick
3. if the Customer at the head of the queue has finished, they are removed from the queue.
4. Finally, the queue is re-displayed.

The owner of the supermarket wants to improve the simulation so that it can handle multiple self-checkout machines. You must extend the program to allow for multiple machines.

- There can be any number of self-checkout machines (controlled by a field).
- All customers will queue in a **single** queue.
- If a checkout machine does not have a customer using it, the first customer in the queue is removed from the queue and placed at that machine.
- Each machine can only process one customer at a time.
  - The self-checkout machines are represented as an array of Customer.
  - A `null` value indicates that there is no Customer currently using that machine.
- On every time tick, if a machine has a customer using it, that customer should be advanced one tick. If that customer is finished, they should be removed from the simulation.

(Question 1 continued on next page)

(Question 1 continued)

Here is the one-machine program:

```
public class Simulation{
    private boolean running = false;
    // Our single queue that all customers join
    private Queue<Customer> selfCheckoutQueue;

    /** Sets up the simulation. The queue is emtpy to begin with. */
    public void setup(){
        running = false;
        selfCheckoutQueue = new ArrayDeque<Customer>();
    }

    /** Runs the simulation */
    public void run(){
        int time = 0;
        running = true;
        while (running){
            time++;
            // Every tick, there is a 20% chance of a new customer arriving.
            if (Math.random()<0.2){
                Customer c = new Customer();
                selfCheckoutQueue.offer(c);
            }
            // If the queue isn't empty, process the first customer.
            if (!selfCheckoutQueue.isEmpty()){
                Customer firstCustomer = Customer.peek();
                firstCustomer.advanceOneTick();
                // If they are finished, remove them from the queue
                if (firstCustomer.isFinished()){
                    selfCheckoutQueue.poll();
                }
            }
            updateDisplay();
        }
    }
}
```

(Question 1 continued on next page)

(Question 1 continued)

(a) **[10 marks]** Complete the run method for the simulation with multiple machines.

```
public class Simulation{
    public static final int NUM_MACHINES = 5; // new
    private boolean running = true;

    private Queue<Customer> selfCheckoutQueue ;
    private Customer[] selfCheckoutMachines; // new

    public void setup() {
        running = false;
        selfCheckoutQueue = new ArrayDeque<Order>();
        selfCheckoutMachines = new Customer[NUM_MACHINES]; // new
    }

   public void run() {
        int time = 0;
        running = true;
        while (running){
            time++;
            if (Math.random()<0.2){
                Customer c = new Customer();
                selfCheckoutQueue.offer(o);
            }
            /*# YOUR CODE HERE */

            updateDisplay();
        }
    }
}
```

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 2. Big-O **[10 marks]**

(a) **[6 marks]** What is the Big-O cost of the following method (in the worst case)?
- 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 List<String> uniqueWordsList(List<String> words){
    List<String> unique = new ArrayList<String>();

    for(String word : words) {
        if ( !unique.contains(word) ) {          // cost= O(      )   times=
            unique.add(word);                    // cost= O(      )   times=
        }
    }
    return unique;
}                                                Total Cost = O(      )
```

(b) **[4 marks]** What is the Big-O cost of the following method, which replaces the List used above with a HashSet?

```
public Set<String> uniqueWordsSet(List<String> words){
    Set<String> unique = new HashSet<String>();

    for(String word : words) {
        if ( !unique.contains(word) ) {          // cost= O(      )   times=
            unique.add(word);                    // cost= O(      )   times=
        }
    }
    return unique;
}                                                Total Cost = O(      )
```

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 3. Recursion **[10 marks]**

(a) **[5 marks]** Complete the following recursive drawLine(x, y, length) method to draw a "bridge" made of lines that form into recursive arches.

- You must use a "first-and-rest" recursion: drawLine(...) should draw the first line and then call itself recursively to draw the rest of the lines.
- The first line should start at (x,y) and end at (x+length,y) – i.e. it is a horizontal line that starts at (x,y) and is length pixels long.
- Then, it should recursively draw two lines that:
  - Have 1/3 the length of the current line
  - Are 10 pixels below the current line
  - Are positioned at either end of the current line (as shown below)
- It should continue to draw lines as long as the length is greater than 5.

For example, drawLine(0, 0, 500) should output the following:

```
public void drawLine(double x, double y, double width){




}
```

Hint: you can use the method UI.drawLine(x1, y1, x2, y3) to draw a line.

(Question 3 continued on next page)

(Question 3 continued)

(b) **[5 marks]** Suppose the variable letters contains the following List of 9 Strings:

"A"	"B"	"C"	"D"	"E"	"F"	"G"	"H"	"I"
0	1	2	3	4	5	6	7	8

What will search(letters, "C", 0, 8) print out?

```
/** Search for something */
public void search( List<String> letters, String letter, int start, int end){
    if ( start <= end) {
        int mid = (start + end)/2;
        UI.print( letters.get(mid) + " ");

        if ( letter.equals( letters.get(mid))) {
            UI.println("Found!");

        } else if ( letter.compareTo(letters.get(mid)) < 0 ) {
            search( letters, letter, start, mid-1);

        } else {
            search( letters, letter, mid+1, end);

        }
    } else {
        UI.println("Not found");
    }
}
```

Answer:

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

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)
   public void sort((E e1, E e2)->{...});      // cost: O(n log(n)), but O(n) if almost sorted

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