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

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

COMP 103 : Test

2018, Sept 12//(Corrected)

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 a Stack for Undo	[13]	
3. Using Collections	[15]	
4. Cost of Algorithms	[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. Properties of Collections **[12 marks]**

The list at the bottom of the page gives 17 possible properties of Collection Types. State the properties that are true of Map's, Queue's, and Set's. (Just write the numbers of the properties.)

(a) **[4 marks]** Properties of a Map:

(b) **[4 marks]** Properties of a Queue:

(c) **[4 marks]** Properties of a Set:

Possible Properties:

1. The first item added is the last item to be removed
2. The order of items in the collection is important
3. The time an item is added does not affect when it is removed
4. The order of items can be reversed
5. Items can only be removed from one end.
6. Items can be added at a specified position
7. Items must have a natural ordering to be stored in the collection.
8. Items must be added with a key
9. The last item added is the first item to be removed
10. An item can be removed by specifying its index
11. A key can be added without any item
12. Items are added and removed from opposite ends.
13. To remove an item, you must specify its key
14. The collection cannot contain any duplicate items
15. The collection records the number of copies of each item
16. The first item added is the first item to be removed
17. The order of items in the collection is unimportant

Question 2. Using a Stack for Undo **[13 marks]**

For this question, you are to add an "undo" feature to a program for a simple maze search game.

The game allows the user to move a token *up*, *down*, *left*, and *right* on a grid-based maze. Part of the program for the game is given.

Modify the code below so that the "Undo" button allows the player to undo their movements and get the token back to where it was at an earlier point. Each press of should reverse one more move.

You will need to add a new collection, complete the doUndo method, and modify some of the other methods.

```
import ecs100.*;
import java.util.*;
public class MazeGame{
    private Cell [][] maze = new Cell[20][20];   // the maze
    private int row;    //current position of the token
    private int col;



    /** Constructor */
    public MazeGame(){
        setupGUI();
        reset();
    }
    /** Set up the buttons */
    public void setupGUI(){
        UI.addButton("Reset", this::reset);
        UI.addButton("Left", this::goLeft);
        UI.addButton("Right", this::goRight);
        UI.addButton("Up",    this::goUp);
        UI.addButton("Down", this::goDown);
        UI.addButton("Undo", this::doUndo);
    }

    /** Make a new maze and put the token at cell (1,1) */
    public void reset(){
        buildMaze();   // builds the maze. All cells on the edge will be walls
        row = 1;       // starts the avatar at position (1,1)
        col = 1;
        redraw();      // redraws the maze and the avatar.

    }
```

(Question 2 continued on next page)

(Question 2 continued)

```
    /** Move the token to the left if the cell to the left is empty */
    public void goLeft(){
        if (maze[row][col-1].isEmpty()){
            col--;

        }
        redraw();
    }
    /** Move the token to the right if the cell to the right is empty */
    public void goRight(){
        if (maze[row][col+1].isEmpty()){
            col++;

        }
        redraw();
    }
    /** Move the token up if the cell above is empty */
    public void goUp(){
        if (maze[row-1][col].isEmpty()){
            row--;

        }
        redraw();
    }
    /** Move the token down if the cell below is empty */
    public void goDown(){
        if (maze[row+1][col].isEmpty()){
            row++;

        }
        redraw();
    }
    /** Undo one more action */
    public void doUndo(){









    }
}
```

Question 3. Using Collections **[15 marks]**

This question involves part of a program for managing an automated factory. The factory has many different machines that can perform different kinds of tasks. Each machine has a unique name.

The program has a Queue of Tasks for each machine, stored in a Map of Queues. The keys of the map are the names of the machines.

It also has a Set of Machine objects.

```
private Map<String, Queue<Task>> machineQueues = new HashMap<String, Queue<Task>>();
private Set<Machine> allMachines = new HashSet<Machine>();
```

The Task class includes the following method:

```
/** Returns the name of the Machine that should process this task */
public String getMachineName(){...}
```

(a) **[7 marks]** Complete the following assignTasks(...) method which is given a list of Tasks that must be perfomed, and must put each Task on the correct queue in the machineQueues map.

- Each Task contains the name of the machine to perform the Task (along with additional details of the Task).
- If a Task has a machine name that is not in the Map, the Task should be ignored.

```
public void assignTasks( List<Task> tasks){




}
```

(Question 3 continued on next page)

(Question 3 continued)

The Machine class contains the following three methods

```
/** Return the name of the machine */
public String getName(){....}

/** Return true if the Machine is currently free (not processing a Task) */
public boolean isFree (){....}

/** Make the Machine process a Task. */
public void process(Task task ){....}
```

(b) **[8 marks]** Complete the following performTasks(...) method which will repeatedly check each machine in allMachines to find any machines that are currently free. If a machine is free and has any Tasks on its queue, then the method should dequeue the first Task on its queue and make the machine process it.

You may assume that there is a queue in machineQueues for every Machine in allMachines.

```
public void performTasks(){
  while (true){









      UI.sleep(1000);  // pause for one second before checking all the Machines again
  }
}
```

Question 4. Cost of Algorithms **[10 marks]**

Consider the following two methods that look for a pair of Patients in a list that have the same name. The methods print out the name of the first pair of Patients they find and then stop.

Analyse the worst-case cost of these methods. (The worst case is when every Patient in the list has a different name).

Assuming that the List contains n patients, for each method:

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

(a) **[3 marks]**

```
public void findPair1(List<Patient> patients){
    for (Patient p1 : patients){
        for (Patient p2 : patients){
            String n1 = p1.getName();          // cost= O(    )  times=
            String n2 = p2.getName();          // cost= O(    )  times=
            if (p1 != p2 && n1.equals(n2)) {   // cost= O(    )  times=
                UI.println(n1);                // cost= O(    )  times=
                return;
            }
        }
    }
}
```

Total Cost = $O($ $)$

(b) **[3 marks]**

```
public void findPair2(List<Patient> patients){
    //Assume patients can be ordered by name
    Collections.sort(patients);                // cost= O(    )  times=

    for (int i=0; i<patients.size()-1; i++){
        String n1 = patients.get(i).getName();     // cost= O(    )  times=
        String n2 = patients.get(i+1).getName();   // cost= O(    )  times=
        if (n1.equals(n2)) {                       // cost= O(    )  times=
            UI.println(n1);                        // cost= O(    )  times=
            return;
        }
    }
}
```

Total Cost = $O($ $)$

(c) **[4 marks]** Write a third algorithm (in Java or in pseudocode) for the same task that uses a Set and has cost $O(n)$.

```
public void findPair3 ( List<Patient> patients){




}
```

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

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
```