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

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

COMP 103 : Test

2019, Jan 9 ** 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 a Stack for Bracket Matching	[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 properties of four different collections are given below. For each list of properties, please identify which type of collection is being described.

(a) **[3 marks]**

- The order of items in the collection is important
- Items can only be removed from one end.
- Items are added and removed from the same end.
- The last item added is the first item to be removed

A stack

(b) **[3 marks]**

- The order of items in the collection is important
- Items can only be removed from one end.
- Items are added and removed from opposite ends.
- The first item added is the first item to be removed

A queue

(c) **[3 marks]**

- The time an item is added does not affect when it is removed
- Items must be added with a unique key
- The order of items in the collection is unimportant
- Items are stored as key-value pairs

A map

(d) **[3 marks]**

- The time an item is added does not affect when it is removed
- The collection cannot contain any duplicate items
- The order of items in the collection is unimportant

A set

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

An integral part of any good code editor is being able to tell if your {'s and }'s match up, or to catch that missing).

For this question, you are to implement an algorithm that can be used to check if all of the brackets in a segment of code match up correctly, and in the right order. You must use a **Stack<Character>**.

There are fours kinds of bracket:

- { and } - Curly brackets
- (and) - Round brackets
- [and] - Square brackets
- < and > - Angle brackets

Some notes:

- The code that your method is checking is represented using a CharSequence, which is a more general way of representing a String-like object. Documentation has been provided in the appendix.
- You may ignore any character in the input that is not an opening or closing bracket.
- Brackets can appear inside each other, but they must be closed in the correct order. E.g: ([ ]) is valid, but ([ )] is *not*.
- All open brackets must be closed. E.g: "([ ] " is not valid because the first (is never closed.
- A closing bracket without an opening bracket is invalid.
- If the input is valid, the method should return **true**. Otherwise it should return **false**.
- Use a Stack<Character> to hold all the opening brackets.
- Some useful methods have been provided

```
/** Returns true if the argument is a valid open bracket */
public boolean isOpenBracket(char c) {
    return ( c == '{' || c == '(' || c == '[' || c == '<' );
}
/** Returns true if the argument is a valid close bracket */
public boolean isCloseBracket(char c) {
    return ( c == '}' || c == ')' || c == ']' || c == '>' );
}

/** Returns true if the closing bracket matches the opening bracket */
public boolean matches(char opening, char closing) {
    return ( ( opening == '{' && closing == '}') ||
             ( opening == '(' && closing == ')') ||
             ( opening == '[' && closing == ']') ||
             ( opening == '<' && closing == '>') );
}
```

(Question 2 continued on next page)

(Question 2 continued)

```
/** Check for matching brackets.
 *  Returns true if all of the brackets match,
 *  or false if there is any sort of mismatch
 *
 * The input is sequence of characters
 * You need to use a Stack<Character>
 * if a character is not an open or close bracket, skip it
 */
public boolean checkBrackets(CharSequence input) {
    Stack<Character> stack = new Stack<>();
    for(int i=0; i<input.length(); i++) {
        char c = input.charAt(i);
        if (isOpenBracket(c)) {
            stack.push(c);
        }
        else if (isCloseBracket(c)) {
            if (stack.isEmpty())
                return false;
            if (!matches(c, stack.pop()))
                return false;
        }
    }
    if (!stack.isEmpty())
        return false;
    return true;




}
```

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

This question involves part of a program for managing the sales representatives of a company that need to make visits to the clients of the company.

The program has a field to store a Map of Queues of Clients, one Queue for each sales representative:

```
private Map<String, Queue<Client>> clientQueues = new HashMap<String, Queue<Client>>();
```

The Client class includes the following method:

```
/** Returns the name of the Sales Representative for this client */
public String getSalesRep (){...}
```

(a) **[7 marks]** Complete the following assignVisits(...) method which is given a list of Clients that need to be visited in the next week, and must put each Client on the correct queue in the clientQueues map.

- The getSalesRep method of Client will return the name of the sales representative who is responsible for the client.
- If a sales representative is not already a key in the clientQueues Map, assignVisits should add the sales representative to the Map, along with a new Queue.

```
public void assignClients ( List<Client> clients ){
    for ( Client client : clients ){
        String name = client.getSalesRep();
        if (! clientQueues.containsKey(name)){
            clientQueues.put(name, new ArrayQueue<Client>());
        }
        clientQueues.get(name).offer( client );
    }




}
```

(Question 3 continued on next page)

(Question 3 continued)

(b) **[8 marks]** Complete the following listDailyVisits() method which will print out the name of each sales representative and the clients that they must visit today. Each representative can visit at most four clients in one day.

- For each sales representative in the clientQueues Map, the first four clients on their queue should be removed from the queue and printed.
- If there are fewer than four clients on the queue, a "–" should be printed in place of the missing clients.
- The Client class has a getName method that will return a String that is the name of the client.

For example, if clientQueues had just two sales representatives, and the second had only two clients in their queue, the method might print:

```
Jamie McCarty:
   Sun Enterprises
   Smith Booster Engines
   Planetary expeditions
   Space Robotics
Lindsay Young:
   Launch Pad Logistics
   Hydrogen Fuel Systems
   --
   --
```

```
public void listDailyVisits (){
    for (String name : clientQueues.keySet()){
        UI.println(name+": ");
        for(int i=0; i<4; i++){
            if (clientQueues.get(name).isEmpty()){
                UI.println("  --");
            }
            else {
                Client client = clientQueues.get(name).poll();
                UI.println( client.getName());
            }
        }
    }




}
```

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

The two fragments of code below are very similar, and both of them reverse the input list. Assume the size of the list is n.

For each fragment, work out the cost (in Big-O notation) by

- working out the cost of performing each line once
- working out the number of times each line will be performed
- computing the total cost

Pay close attention to the differences between the two fragments.

(a) **[3 marks]**

```
public void reverse1(List<String> input){

    Stack<String> stack = new Stack<String>();
    while (!input.isEmpty()) {
        String s = input.remove(0);          // cost = O( n )   times= n
        stack.push(s);                       // cost = O( 1 )   times= n
    }
    while(!stack.isEmpty()) {
        String s = stack.pop();              // cost = O( 1 )   times= n
        input.add(s);                        // cost = O( 1 )   times= n
    }
}
```

Total Cost = $O(\ n^2\ )$

(b) **[3 marks]**

```
public void reverse2(List<String> input){

    Queue<String> queue = new ArrayDeque<String>();
    while (!input.isEmpty()) {
        String s = input.remove(input.size()-1);   // cost = O( 1 )   times= n
        queue.offer(s);                            // cost = O( 1 )   times= n
    }
    while(!queue.isEmpty()) {
        String s = queue.poll();                   // cost = O( 1 )   times= n
        input.add(s);                              // cost = O( 1 )   times= n
    }
}
```

Total Cost = $O(\ n\ )$

(Question 4 continued on next page)

(Question 4 continued)

(c) **[4 marks]** A program uses a HashMap to store all the tax details for citizens.

When the Map has 100,000 citizens, the program takes 7 microseconds to retrieve the tax details for a given citizen.

If there were 100,000,000 citizens, how long would you expect the program to take to retrieve the tax details for a citizen? Briefly explain why.

It would still take 7 microseconds, as a HashMap is O(1) for retrieval

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

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)
   public Set<Map.Entry<K,V>> entrySet()  // 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

interface CharSequence
    public char charAt(int index);   // Returns the char value at the specified index
    public int length();             // Returns the length of this CharSequence
```