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

**ID Number:** ..... **Signature** .....

# Model Solutions
## COMP 103: Mid-term Test

21st of August, 2015

### Instructions

- Time allowed: **45 minutes**
- There are 45 marks in total.
- Answer **all** the questions.
- Write your answers in the boxes in this test paper and hand in all sheets.
- Brief Java documentation is supplied on the last page.
- This test will be converted to 20% of your final grade
  (but your mark will be boosted up to your exam mark if that is higher.)
- You may use paper translation dictionaries.
- You may write notes and working on this paper, but make sure it is clear where your answers are.

| **Questions** | | **Marks** | |
|---|---|---|---|
| 1. | Various | [10] | |
| 2. | Programming with Collections | [25] | |
| 3. | Costs, sorts, and recursion | [10] | |
| | TOTAL: | 45 | |

**Question 1. Various questions** [10 marks]

**(a)** [4 marks] By drawing a circle around the correct answer, indicate whether the following statements are TRUE or FALSE:

- List is an interface, that implements Collection: TRUE / FALSE — False
- The extends relationship is only allowed between classes: TRUE / FALSE — False
- A for-each loop can be used on all Iterable types: TRUE / FALSE — True
- Insertion Sort is a stable sort: TRUE / FALSE — True

**(b)** [2 marks] Suppose you use a variable myPatients of type *Queue* to store information about patients in a waiting room, with patients being represented as objects of type Patient. Write code to declare and initialise an empty instance of such a queue.

```
Queue <Patient> myPatients = new ArrayQueue<Patient> ();
```

**(c)** [2 marks] Java has interfaces *Comparable* and *Comparator*. In words, describe the difference between these two interfaces.

Comparable ensures that objects can be compared to each other, and that they have a natural ordering criterion (ie. ensures they have a default ordering criterion, if you like). Comparator enables objects to be compared even when they are not comparable, and supports multiple sorting criteria.

**(d)** [2 marks] The class ArrayList does not implement *List* directly. Name the class that ArrayList extends and briefly explain to motivation behind this design.

AbstractList.
The idea is that many List methods can be generically be expressed in terms of a few basic methods and should be inherited by a common abstract class rather than rewritten time and again.

## Question 2. Programming with Collections [25 marks]

**(a)** [3 marks] What will the following code print?

```
public static void main(String[] a) {
    Stack<String> ss = new Stack<String>();
    ss.push("A");
    UI.print(ss.peek());
    ss.push("B");
    ss.push("C");
    ss.pop();
    ss.push("D");

    while (!ss.isEmpty())
        UI.print(ss.pop());
}
```

ADBA

**(b)** [4 marks] By circling the number at the left, indicate clearly which of the following are **valid**. If there are invalid ones, explain the problem using the line below them:

1. `Collection<Shape> mycollection = new List<Shape> ();`

---

2. `List<ArrayList> mycollection1 = new ArrayList<ArrayList> ();`

---

3. `Collection<Shape> mycollection = new ArrayList<Shape> ();`

---

4. `Set <Map<String, Object>> mycollection3 = new HashSet<Map<String, Object>> ();`

---

**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.

**(c)** [10 marks] Suppose you are working on a program that deals with two *List*s of words, where each word is represented with a String .

Write a method called "uniqueInReversed" that takes the two lists as arguments. The method should check whether the first list contains exactly the same entries as the second list but in reverse order. If the lists are not reversed versions of each other, the method should return -1. Otherwise, it should return the number of unique words in the first list, i.e., number of words, discounting duplicates.

*Example:* if the lists being passed in were

```
list1:  dog, cat, cat, eel
list2:  eel, cat, cat, dog
```

then the method should return 3.

You do not need to index the lists in order to write this method. In fact, to obtain full marks, do not use get(...). However, you must not use a predefined method such as Collections.reverse either.

```
int uniqueInReversed(List<String> l1, List<String> l2){
    static int uniqueInReversed(List<String> l1, List<String> l2) {
        if (l1 == null || l1 == null) return -1;
        if (l1.size() != l2.size()) return -1;

        Stack<String> stack = new Stack<String>();
        Set<String> unique = new HashSet<String>();

        for (String word : l1)
            stack.push(word);

        for (String word : l2) {
            if (!word.equals(stack.pop())) return -1;

            unique.add(word);
        }

        return unique.size();
    }




}
```

**(d)** [6 marks] You have written a *WaitingList* class and now want to enable waiting lists to be compared against each other. Extend the below code so that the natural ordering of two waiting lists puts smaller lists (with less elements) before longer lists.

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

    public int compareTo (WaitingList other) {
      return this.size() - other.size();
    }



}
```

**(e)** [2 marks] Describe in words which steps you would have to undertake so that it becomes possible to iterate through the elements of a *WaitingList* using the "for-each" syntax.

The class would have to implement Iterable. It would have to return an iterator that implements Iterator and and the implements the methods hasNext() and next(), keeping track of the iteration progress. .

**Question 3. Costs, sorts, and recursion** [10 marks]

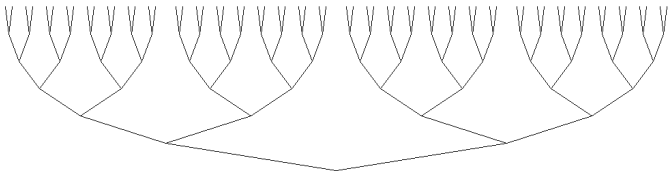
**(a)** [3 marks] Adding an item to a SortedArraySet still has complexity $\mathcal{O}(n)$ just like with ArraySet. First explain why and then describe when it makes sense to prefer SortedArraySet over ArraySet.

Searching for a potential duplicate is quick but any addition requires shifting up elements. SortedArraySet is still preferrable when there a lot of "contains" lookups compared to adds/removes.

**(b)** [2 marks] Suppose InsertionSort takes 1 second to sort 1000,000 almost sorted items on your new desktop machine. Approximately how long will it take to sort 8000,000 almost sorted items?

The second problem is 8 times larger than the first. InsertionSort scales as $\mathcal{O}(n)$ with nearly sorted lists, so by that reasoning it should take about 8 seconds..

**(c)** [5 marks] Write a recursive method that draws the stylised tree below.



Assume the starting point (root of the tree at the bottom) to be at (x=500, y=300). Each layer of nodes is separated vertically by 40 pixels from its upper and lower neighbours respectively. At the layer above the root, the horizontal "span", i.e. , the distance between two sibling nodes, is 500 pixels (= twice the horizontal distance between the root and one of the nodes). At the next layer up it is half of that, and so on. No more branches should be drawn, if the span has become less than four (4). You may assume your method to be called like this: drawTree(500, 300, 250);.

```
static public void drawTree(int x, int y, int span) {
  static public void drawTree(int x, int y, int span) {
      if (span < 4) return;

      int newY = y − 40;
      int newX1 = x−span;
      int newX2 = x+span;

      UI.drawLine(x, y, newX1, newY);
      UI.drawLine(x, y, newX2, newY);

      drawTree(newX1, newY, span/2);
      drawTree(newX2, newY, span/2);
  }
}
```

**********

# Appendix

Some brief and truncated documention that may be helpful:

```
interface Collection<E>
    public boolean isEmpty()
    public int size()
    public boolean add(E item)
    public boolean contains(Object item)
    public boolean remove(Object element)
    public Iterator<E> iterator()

interface List<E> extends Collection<E>
    // Implementations: ArrayList, LinkedList
    public E get(int index)
    public E set(int index, E element)
    public void add(int index, E element)
    public E remove(int index)
    // plus methods inherited from Collection

interface Set extends Collection<E>
    // Implementations: ArraySet, HashSet, TreeSet
    // methods inherited from Collection

interface Queue<E> extends Collection<E>
    // Implementations: ArrayQueue, LinkedList
    public E peek ()                    // returns null if queue is empty
    public E poll ()                    // returns null if queue is empty
    public boolean offer (E element)    // returns false if fails to add

class Stack<E> implements Collection<E>
    public E peek ()                    // returns null if stack is empty
    public E pop ()                     // returns null if stack is empty
    public E push (E element)           // returns element being pushed

interface Map<K, V>
    // Implementations: HashMap, TreeMap, ArrayMap
    public V get(K key)                 // returns null if no such key
    public V put(K key, V value)        // returns old value, or null
    public V remove(K key)              // returns old value, or null
    public boolean containsKey(K key)
    public Set<K> keySet()

public class Collections
    public void sort(List<E>)
    public void sort(List<E>, Comparator<E>)
    public void shuffle(List<E>, Comparator<E>)
```