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

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

Model Solutions
COMP 103: Test 1

11 April, 2013

Instructions

- Time allowed: **40 minutes**
- There are 40 marks in total.
- Answer **all** the questions.
- Write your answers in the boxes in this test paper and hand in all sheets. Ask for additional paper if you need it.
- If you think some question is unclear, ask for clarification.
- Brief Java documentation is supplied on the last page.
- This test will be converted to 15% 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, and calculators without a full set of alphabet keys.
- You may write notes and working on this paper, but make sure it is clear where your answers are.

Questions	**Marks**	
1. Collections	[10]	
2. Programming with Collections	[14]	
3. Using an iterator	[7]	
5. Costs, and Simple sorts	[9]	
	TOTAL:	

Question 1. Collections [10 marks]

For each of these programming tasks, state what kind of collection you would use, and justify why you chose that kind of collection.

(a) [2 marks] A collection recording the coffee orders in a cafe. The coffees are made in the sequence they were requested, and the program needs to provide only the next coffee order to be made.

Collection Type: Queue
Justification: It's a queue of coffee orders, so duplicates are allowed (not a Set). But they are made in the order requested, and only the most pressing order needs to be displayed, hence a Queue is the right Collection to use.

(b) [2 marks] A collection recording the people currently attending an event. New people may arrive at any time, or existing attendees may depart, but at any time the collection needs be able to provide a list of all the people currently at the event.

Collection Type: Set
Justification: Attendees form a Set because duplicates should not be allowed. However there is no requirement for ordering, just attendance. Hence a Set.

(c) [2 marks] A collection recording the current inventory of a store. Upon request, the program needs to be able to provide the number of items of a certain type of stock that the store currently has.

Collection Type: Map.
Justification: The user will look up a type of item ("Socks") and get a number (119), so what is required is a Map from Strings to Integers.

(Question 1 continued on next page)

(d) [2 marks] A Set is a collection that has no ordering, and we have looked at an implementations of this that store the items in an array. It can still be a good idea to implement it by using a *sorted* array, rather than an unsorted one. Explain why.

If the Set is stored in sorted form, we can use the binary search algorithm to search it. This is much (much) faster than the linear search required if the array is not sorted.

(e) [2 marks] In the box below, fix the error(s) in the following declaration for a Set of strings:

```
Set localSet = new Set<String> ;
```

corrected version:

```
Set<String> localSet = new HashSet<String> ();
```

(or some other valid *Set* implementation).

Question 2. Programming with Collections [14 marks]

This question is about a program called SongsOrganiser, that manages information about a playlist of Songs. Consider the following code for a Song class, which does nothing except store information:

```
public class Song {
    private int year;
    private String artist, title;

    // constructor
    public Song(String artist, String title, int yr) {
        this.year = yr;
        this.artist = artist;
        this.title = title;
    }
    // some 'get' methods to provide information.
    public String getArtist() { return artist; }
    public String getTitle()  { return title; }
    public Integer getYear()  { return year; }
}
```

A real SongsOrganiser program would probably read in data from a large file. The demo version shown here just creates some Song objects explicitly, and stores them in a List of Songs:

```
public class SongsOrganiser {
    // constructor
    public SongsOrganiser() {
        List <Song> songlist = new ArrayList <Song> ();

        songlist.add(new Song("Phoenix Foundation", "Hitchcock",2005));
        songlist.add(new Song("Phoenix Foundation", "Buffalo",2010));
        songlist.add(new Song("Dresden Dolls", "Shores of California",2007));
        songlist.add(new Song("Tiny Ruins", "Priest with balloons",2011));
         :
        songlist.add(new Song("Azelia Banks", "212",2011));
        songlist.add(new Song("SJD", "Beautiful haze",2007));

        // Set up a map from artist names to Songs
        Map <String, List<Song>> songMap;
        songMap = makeMapOfSongs(songlist);      // to be written, question 2a.
        sortEachArtist(songMap);                 // to be written, question 2b.
    }

    // methods
    :
}
```

The questions here ask you to write the methods mentioned in the last two lines above, which would go into the SongsOrganiser class.

(a) [9 marks] Write the method makeMapOfSongs, which takes a List of Song objects, and returns a Map from artist names to Songs.

You may assume that each song appears only once in the original List.

HINT: write out the necessary steps as "pseudocode" first.

```
public Map <String, List<Song>> makeMapOfSongs( List<Song> allSongs ) {



    // Make a new map.
    Map <String, List<Song>> myMap = new HashMap <String, List<Song>> ();

    // Build up a map of artists and lists first , in any order .
    for (Song b : allSongs) {
        String art = b. getArtist ();
        if (myMap.keySet().contains(art)) {
            myMap.get(art).add(b);              // add to an existing list
        }
        else {
            List <Song> songlist = new ArrayList <Song> (); // a new list
            songlist .add(b);                   // add to the new list
            myMap.put(art,songlist); // list is the map's value for this key
        }
    }
    return myMap;









}
```

(Question 2 continued on next page)

(b) [5 marks]

Write the method sortEachArtist, which takes the current Map, and sorts the List of Songs associated with each artist into ascending order by year.

A comparator is provided, and you should consider using the Collections.sort() method described in documentation at the end of this test.

```
public void sortEachArtist(Map <String, List<Song>> myMap) {



// EITHER THIS                    ..........................................................

        for (String art : myMap.keySet()) {
            List <Song> songlist = myMap.get(art);
            Collections.sort(songlist, new SongComparator());
            myMap.put(art,songlist);   // put it back ...
            // NB. final line not essential, as Collections.sort() is 'in place'
        }

// OR THIS, which works because Collections.sort() is 'in place      '.................

        for (List <Song> sl : myMap.values())
            Collections.sort(sl, new SongComparator());

    }


    /** Comparator that will order Songs based on their year. */
    private class SongComparator implements Comparator <Song> {
        public int compare (Song song1, Song song2) {
            return (song1.getYear() - song2.getYear());
        }
    }
```

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. Using an iterator [7 marks]

Consider the following code for a random number generator:

```
public class RandNumIterator implements Iterator <Integer> {
    private int num = 1;
    public boolean hasNext() {
        return true;
    }
    public Integer next() {
        num = (num * 92863) % 104729 + 1;
        return num;
    }
    public void remove(){ throw new UnsupportedOperationException(); }
}
```

(a) [2 marks] Why is the remove method included here, when all it does is throw an exception?

It is required for the interface Iterator

(b) [5 marks] Complete the following method, that first creates a RandNumIterator (an instance of the class given above) and then uses it to print out N random numbers.

```
private void printNNumbers(int N) {
        Iterator <Integer> lottery = new RandNumIterator();
        for (int i=1; i<N; i++)
            System.out.println( lottery .next ());




}
```

Question 4. Costs and Simple sorts [9 marks]

(a) [5 marks] Give the cost, in "big-O" notation, of the following ArraySet operations (assuming the array is *not* sorted), for a Set of size n. Include a justification in each case.

The first one is done for you as an example.

Cost of contains: $\mathcal{O}(n)$
REASON:
The array is unsorted, so you have no option but to search it element-by-element. With n items, you will find the item after looking at $n/2$ on average, which we say is of order n.

Cost of add: $\mathcal{O}(n)$
REASON: because you have to first check it's not in the Set already

Cost of remove: $\mathcal{O}(n)$
REASON: because you have to find it first

Now do the same for a SortedArraySet implementation, in which the array containing the data items is maintained in a sorted order.

Cost of contains: $\mathcal{O}(\log n)$
REASON: binary search can find it in a mere $\log n$ comparisons!

Cost of add: $\mathcal{O}(n)$
REASON: although you can check it's not there, adding it is harder: can't go at end

Cost of remove: $\mathcal{O}(n)$
REASON: you can find it quickly, but removal requires moving everything down to fill the gap

(b) [4 marks] In words, explain why the InsertionSort algorithm has cost $\mathcal{O}(n^2)$ on lists that are randomly ordered to start with, but has cost $\mathcal{O}(n)$ on lists that are nearly sorted to start with.

Cost on randomly ordered Lists is $\mathcal{O}(n^2)$, because...

InsertionSort repeatedly takes next item and finds where to put it in the List up to that point. The outer loop goes around n times, and each time runs an inner loop going backwards to figure out where the new item goes: this is linear in n, hence $\mathcal{O}(n^2)$.

Cost on nearly-ordered Lists becomes $\mathcal{O}(n)$, because...

The inner loop now only has one comparison to do (not $n/2$ or so), hence $\mathcal{O}(n)$.

****

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>)
```