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

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

# COMP 103 : Test 2

2021, Jan 15 ** WITH SOLUTIONS **

### 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.

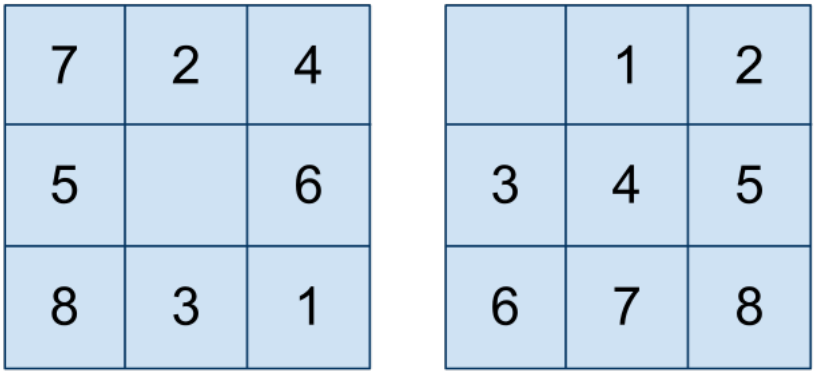
| Questions | Marks | |
|---|---|---|
| 1. Using A Stack for Undo | [10] | |
| 2. Using Collections | [10] | |
| 3. CompareTo, Equals, and HashCode | [10] | |
| | TOTAL: | |

**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 A Stack for Undo** **[10 marks]**

For this question, you are to add an "undo" feature to a program for a simple sliding puzzle game, similar to the one shown below.




The game allows the user to slide tiles around, with the goal of moving all of the numbers into the correct order. The puzzle tiles can slide *up*, *down*, *left*, and *right*, and only into the empty space.

Modify the code below so that the "Undo" button allows the player to undo their movements and slide the tiles back to where they were at an earlier point. Each press of the undo button should reverse one move.

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

```
public class SlidingPuzzleGame{
    private Tile [][] puzzle = new Tile[3][3];   // the puzzle board
    private int row;    //current position of the empty space
    private int col;
    private Stack<String> undoStack = new Stack<String>();

    /** Set up the buttons */
    public void setupGUI(){
        UI.addButton("Reset", this::reset );
        UI.addButton("Left", this :: slideLeft );
        UI.addButton("Right", this:: slideRight );
        UI.addButton("Up",    this :: slideUp );
        UI.addButton("Down", this :: slideDown);
        UI.addButton("Undo", this :: doUndo);
    }
    /** Make a new puzzle and put the empty space at cell (1,1) */
    public void reset (){
        buildPuzzle (); // shuffles the puzzle, leaving the space in the center
        row = 1;        // starts the empty space at position (1,1)
        col = 1;
        redraw();       // redraws the puzzle
        undoStack.clear ();      // OR undoStack = new Stack<String>();
    }
```

(Question 1 continued on next page)

**(Question 1 continued)**

```
/** Slide the tile to the left of the of the space into the space */
public void slideLeft (){
    if (col-1 >= 0){
        puzzle[row][col] = puzzle[row][col-1]; //swap the empty space
        puzzle[row][col-1] = null;             //with the tile to the left
        col--;
        undoStack.push("left");
        // or "right" if the undo method doesn't do the reverse
    }
    redraw();
}
/** Slide the tile to the right of the of the space into the space */
public void slideRight (){
    if (col+1 < 3){
        puzzle[row][col] = puzzle[row][col+1]; //swap the empty space
        puzzle[row][col+1] = null;             //with the tile to the right
        col++;
        undoStack.push("right");
        // or "left" if the undo method doesn't do the reverse
    }
    redraw();
}
/** Slide the tile above the space into the space */
public void slideUp(){
    if (row-1 >=0){
        puzzle[row][col] = puzzle[row-1][col]; //swap the empty space
        puzzle[row-1][col] = null;             //with the tile above it
        row--;
        undoStack.push("up");
        // or "down" if the undo method doesn't do the reverse
    }
    redraw();
}
/** Slide the tile below the space into the space */
public void slideDown(){
    if (row+1 < 3){
        puzzle[row][col] = puzzle[row+1][col]; //swap the empty space
        puzzle[row+1][col] = null;             //with the tile below it
        row++;
        undoStack.push("down");
        // or "up" if the undo method doesn't do the reverse
    }
    redraw();
}
```

(Question 1 continued on next page)

**(Question 1 continued)**

```
    /** Undo one action */
    public void doUndo(){
        if (!undoStack.isEmpty()){
            String action = undoStack.pop();
            if (action.equals("left"))       { slideRight(); }
            else if (action.equals("right")) { slideLeft(); }
            else if (action.equals("up"))    { slideDown(); }
            else if (action.equals("down"))  { slideUp(); }
            redraw();
        }

        /* ----- OR ------*/
        // (if they push the opposite action onto the stack, instead of the actual action)
        if (!undoStack.isEmpty()){
            String action = undoStack.pop();
            if (action.equals("left"))       { slideLeft(); }
            else if (action.equals("right")) { slideRight(); }
            else if (action.equals("up"))    { slideUp(); }
            else if (action.equals("down"))  { slideDown(); }
            redraw();
        }
    }
}
```

**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. Using Collections** **[10 marks]**

Complete the following findMostCommon(...) method which is given a List of words (Strings), and returns the most common word in that list (i.e. the word that occurs the most often).

```
public String findMostCommon(List<String> words) {

    Map<String,Integer> wordCounts = new HashMap<String,Integer>();

    for(String word : words) {
        if (!wordCounts.containsKey(word))
            wordCounts.put(word,1);
        else
            wordCounts.put(word, wordCounts.get(word)+1);
    }
    String mostCommon = null;
    int count = 0;
    for (Map.Entry<String,Integer> e : wordCounts.entrySet()) {
        if ( e.getValue() > count) {
            mostCommon = e.getKey();
            count = e.getValue();
        }
    }
    return mostCommon;









}
```

**Hint:** You will want to use a Map to keep track of how many times each word appears.

**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. CompareTo, Equals, and HashCode** **[10 marks]**

Suppose you are writing a program to keep track of the books that you own. You have defined a Book class that contains several fields (given below) such as the title, author, and yearFirstPublished.

(a) **[5 marks]** You would like to be able to sort a List that contains Book objects, and for the Book object to have a natural ordering.

Add the approprate interface declaration to the Book class so that it is comparable. Then, complete the compareTo(...) method for the Book class so that:

- Books will be sorted alphabetically by author first, and then title.
- You can assume that neither title nor author will be null.

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

    private String  title ;
    private String  author;
    private int  yearFirstPublished ;

    public int compareTo(Book other) {
        if ( author.equals(other.author) ) {
            return  title .compareTo(other. title );
        }
        return author.compareTo(other.author);




    }
```

(Question 3 continued on next page)

**(Question 3 continued)**

(b) **[5 marks]** It is not sufficient to just define a compareTo(...) method by itself–we also need to define the equals(...) and hashcode() methods.

Complete the equals(...) and hashcode() methods for the Book class below.

- Two books are equal if and only if they have the same title and author.
- You can assume that neither title nor author will be null.
- Ensure that the equals(...) and hashcode() methods are consistent:
  - with each other (if two books are equal, they must have the same hash code), **and**
  - with the compareTo(...) method (if two books are equal, compareTo must return 0)

```
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Book other = (Book) obj;
        if (!author.equals(other.author))
            return false;
        if (! title .equals(other. title ))
            return false;
        return true;


    }

    public int hashCode() {
        int prime = 31;
        int result = 1;
        result = prime * result + author.hashCode();
        result = prime * result + title .hashCode();


        return result ;
    }


}
```

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

# 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
```