CS1027: Foundations of Computer Science II
Assignment 4
Due: April 2, 11:55pm.

Purpose

To gain experience with
- The solution of problems using circular arrays and queues
- The design of algorithms in pseudocode and their implementation in Java.

1. Introduction

For this assignment you will be given a map and you are required to write a program that finds a path from a starting location to a destination, if such a path exists. This assignment is similar to Assignment 2, except that this time you are required to find a shortest path from the starting location, the Middlesex College building, to the destination, your house. As in Assignment 2 the map is divided into the same set of rectangular cells:

- A starting map cell, where the Middlesex College building is located.
- A destination map cell where your house is situated.
- Map cells containing blocks of houses or parks, where cars cannot drive.
- Map cells containing roads, where cars can drive. There are several types of road cells:
    - Road intersections; up to four different roads can converge into a road intersection. A car coming into an intersection can turn in any direction where there is a road
    - North road, south road, east road, and west road cells. All these roads are one-way roads.

Figure 1 shows an example of a map divided into cells and a shortest path from the starting map cell to the destination.

Each map cell has up to 4 neighboring cells indexed from 0 to 3. Given a cell, the north neighboring cell has index 0, the east neighboring cell has index 1, the south neighbor has index 2, and the west neighbor has index 3.

2. Classes that You Need to Implement

A description of the classes that you need to implement in this assignment is given below. You can implement more classes, if you want. You **cannot** use any static instance variables. The data structure that you must use for this assignment is an ordered list implemented using a circular array, as described in the next section.

You **cannot** use any of the other java classes from the Java library that implement collections (i.e. you cannot use any java class that can store a set of 2 or more values).

2.1 CellData.java

This class represents the data items that will be stored in the circular array. Each object of this class stores two things: an identifier and a value. This class will be declared as follows:

public class CellData<T>

This class will have two instance variables:

- *private T id*. A reference to the identifier stored in this object/
- *private int value*. This is the value of the data item stored in this object.

You need to implement the following methods in this class:

- *public CellData(T theId, int theValue)*. Constructor for the class. Initializes *id* to *theId* and value to *theValue*.
- *public int getValue()*. Returns instance variable *value*.
- *public T getId()*. Returns instance variable *id*.
- *public void setValue(int newValue)*. Assigns newValue to instance variable *value*.
- *public void setId(T newId)*. Assigns the value of *newId* to instance variable *id*.

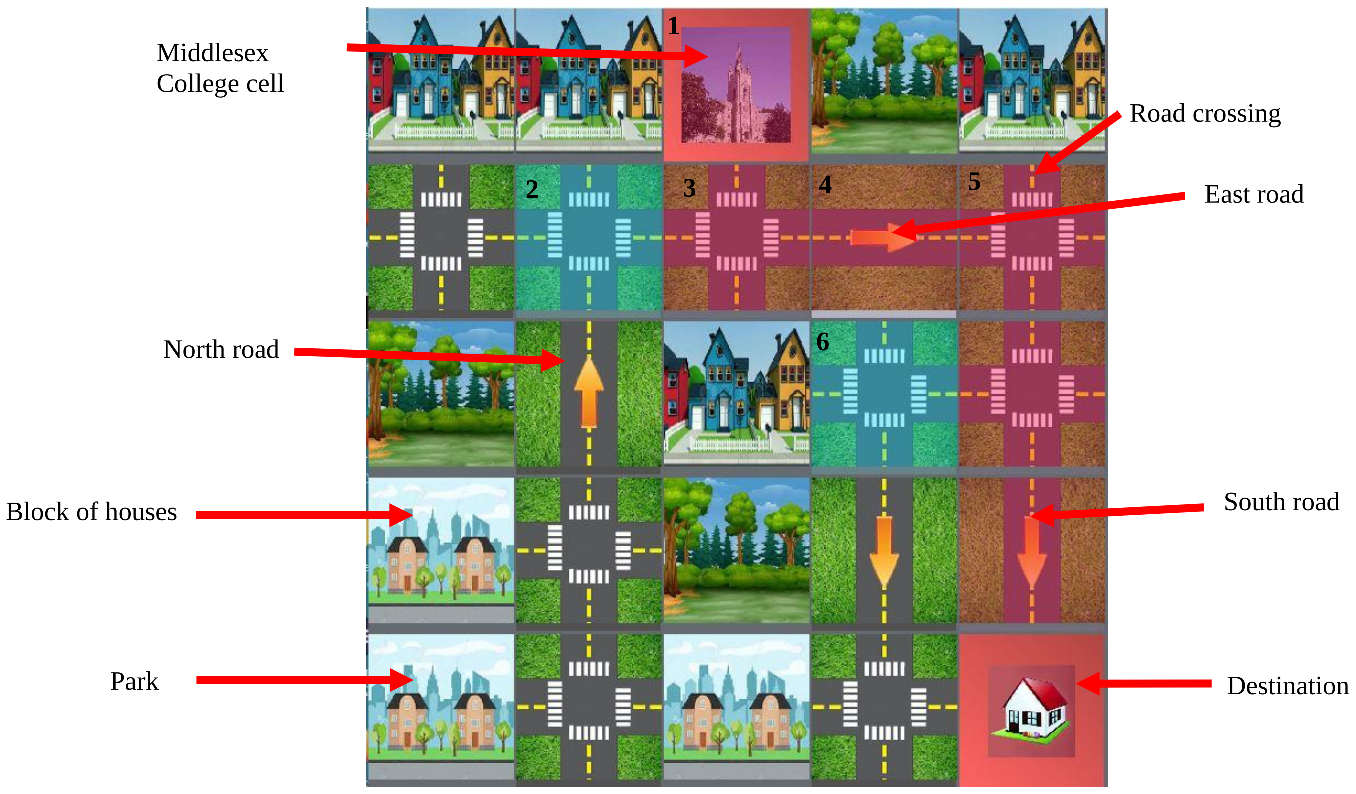


Figure 1.

2.2 OrderedCircularArray.java

This class implements an ordered list using a circular array. This class must implement the interface *SortedListADT.java*. The header of this class must be this:

public class OrderedCircularArray<T> implements SortedListADT<T>

You can download *SortedListADT.java* from the course's website. This class must have the following private instance variables:

- *private CellData*[] *list*. This array will store the data items of the ordered list.
- *private int front*. This variable stores the **position of the first data item** in the list; this is the data item with the smallest value. In the constructor of this class this variable must be initialized to 1.

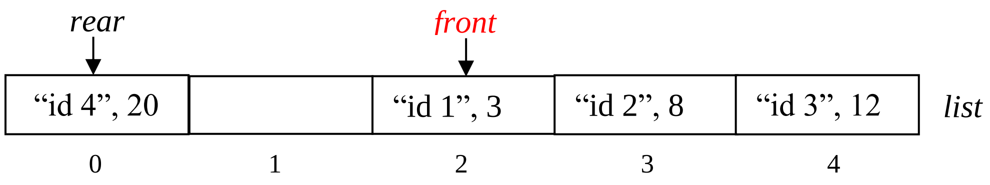
Note that this is different from the way in which the variable *front* is initialized in the lecture notes.
private int rear. This is the index of the **last** data item in the ordered list; this is the data item with the largest value. In the constructor for this class this variable must be initialized to 0. Again, note that this is different from the way in which variable *rear* is used in the lecture notes.

- *private int count*. The value of this variable is equal to the number of data items in the list.

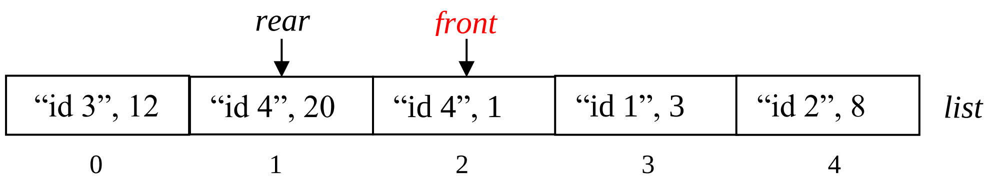
Circular array *list* stores objects of the class *CellData*. These objects are kept sorted in the array according to their integer *value* attributes. An example of the array *list* storing 3 *CellData* objects in order of their integer values is shown in the next page.

This class needs to provide the following public methods.

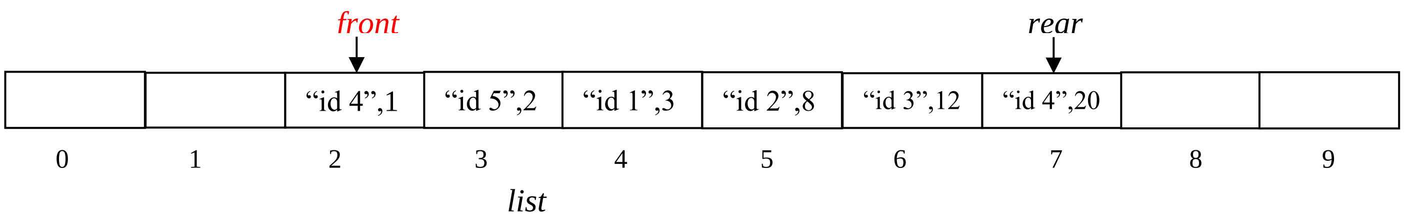
- *public OrderedCircularArray()*. Initializes the instance variables as specified above. Array *list* must initially have length 5.
  Note that when initializing array *list* as *list = new CellData<T>*[5]; the compiler will issue an error message because type *T* is not known at compilation time. To solve the problem use casting (create an array of type *CellData*[] and then cast it to the type that you need).
- *public void insert (T id, int value)*. Adds a new *CellData object* storing the given *id* and *value* to the ordered list. You must ensure that after adding this new *CellData* object the list remains sorted. Note that **the value of front must not change** when adding a new *CellData* object into the list.
  For example, if the array *list* is the following



  and we invoke *insert*("id 4", 1), then the new array must be as follows



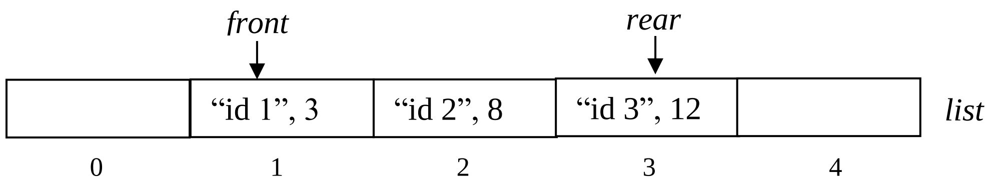
  If the array is full when the *insert* method is invoked, a new array of size twice as large as the original one must be created, and the information from the old array must be copied there. **You must ensure that the value of front does not change**. So, if in the above array with 5 elements we invoke *insert*("id 5",2), the new array must be as follows



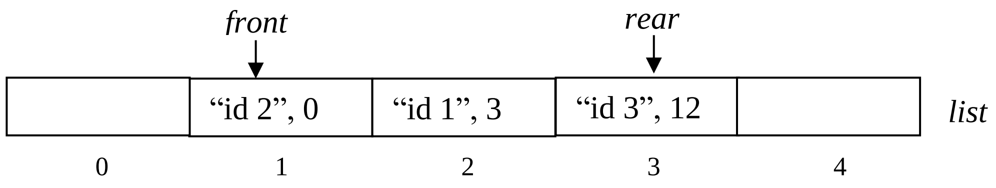
- *public int getValue(T id) throws InvalidDataItemException*. Returns the integer value of the *CellData* object with the specified *id*. An *InvalidDataItemException* is thrown if no *CellData* object with the given *id* is in the ordered list. In this assignment we will assume that no two *CellData* objects in the ordered list have the same *id*.

  Note that to check whether an object with the given *id* is in the ordered list you need to scan the ordered list using linear search. To check if the *CellData* object stored in some position of the ordered list has the same id attribute as *id* you must use the *equals* method, not the "==" operator.
- *public void remove(T id) throws InvalidDataItemException*. Removes from the ordered list the *CellData* object with the specified *id*. An *InvalidDataItemException* is thrown if no *CellData* object in the ordered list has the specified *id*. Note that the value of front must not change when removing a data item from the ordered list using this method.
- *public void changeValue (T id, int newValue) throws InvalidDataItemException*. Changes the *value* attribute of the *CellData* object with the given *id* to the specified *newValue*. An *InvalidDataItemException* is thrown if no object in the ordered list has the given *id*. After changing the value of the *CellData* obejct, you need to ensure that the list is still sorted by value, so you might have to reorder some of the information stored in the circular array.

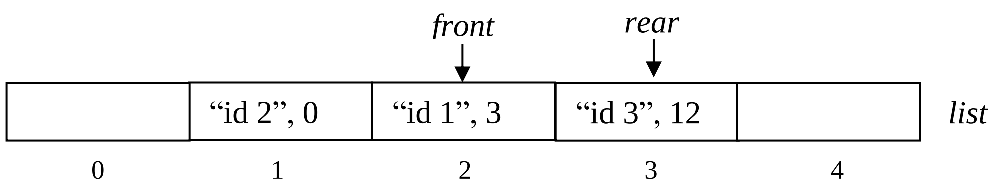
  Hint. First remove the *CellData* object with the given *id* and then add a new *CellData* item with the given *id* and *newValue*.
- *public T getSmallest() throws EmptyListException*. Removes and returns the *id* or the *CellData* object in the ordered list with smallest associated value. Note that this is the *CellData* object stored in the position of the circular array specified by *front*. An *EmptyListException* is thrown if the list is empty. Note that for this method the value of *front* must change once the *CellData* object with smallest value has been removed.
- *public boolean isEmpty()*. Returns *true* if the ordered list is empty and it returns *false* otherwise.
- *public int size()*. Returns the number of data items in the ordered list.
- *public int getFront()*. Returns the value of instance variable *front*.
- *public int getRear()*. Returns the value of instance variable *rear*.



To make sure you understand the above methods, consider the above ordered list where each entry stores a *CellData* object in which the *id* attribute is of type String. Initially the list was empty, and the value of *front* was 1, then method *insert* was invoked 3 times to add the 3 *CellData* objects shown. Note how *front* did not change and the entries are sorted by the integer attribute *value* of the *CellData* objects. If we invoke on this *list* method *getDataValue* ("id 2"), this method should return the value 8, while invoking *getDataValue*("id 4") should throw an *InvalidDataItemException*. Invoking *changeValue*("id 2", 0) will set the integer value stored in *list*[2] to 0. Since now the *CellData* objects are not in increasing order of value, the *CellData* object "id 2", 0 needs to move to the front of the list as shown below.



If now we invoke *getSmallest*(), this will return "id 2" (as this data item has the smallest value) and then the value of *front* will increase to 2.



You can implement other methods in this class, if you want to, but they must be declared as private.

2.3 ShortestPath.java

This class will have an instance variable

CityMap cityMap;

This variable references the object representing the city map where your program will try to find a shortest path, a path with the minimum number of map cells, from the starting cell to the destination cell. Class *CityMap*, described below, is given to you. You must implement the following methods in this class:

- *public ShortestPath (CityMap theMap).* This is the constructor for the class. It receives as input a reference to a *CityMap* object representing the city map. In this method you must initialize instance variable *cityMap*: *cityMap = theMap*.
- *public void findShortestPath().* This method will look for a path with the minimum number of map cells from the starting cell to the destination cell. If a path is found, this method must print a message indicating how many cells there were in the path (including the starting cell and the destination cell). If no path was found, this method must print the message "No path found".
  You are encouraged to design your own algorithm to look for a shortest path from the starting cell to the destination. Read the description of the classes *Map* and *MapCell* below to see how to select the starting cell and the destination cell, and how to mark cells as visited. If you cannot figure out how to find a path, please read the description of an algorithm given in the next section.
- *private MapCell nextCell(MapCell cell).* The parameter of this method is the current map cell. This method returns the first unmarked neighboring map cell that can be used to continue the path from the current one. Use method *isMarked* from class *MapCell* (described below) to determine whether a neighboring map cell has been marked or not. This method is similar to the *nextCell* method from Assignment 2, except that now we do not prefer the destination or a road intersection cell over the other cells.

  If there are no unmarked cells adjacent to the current one that can be used to continue the path, this method must return `null`.

Your program must catch any exceptions that might be thrown. For each exception caught an appropriate message must be printed. The message must explain what caused the exception to be thrown.

You can write more methods in this class, if you want to, but they must be declared as private.

3. An Algorithm for Computing a Shortest Path

This section describes an algorithm for computing a shortest path between two map cells. You do not have to implement this algorithm if you do not want to. You are encouraged to design your own algorithm, but the algorithm must use an ordered list from class *OrderedCircularArray*.

The algorithm starts at the starting cell and as it traverses the map it will store in the ordered list the map cells that it might visit next. Each entry of the circular array will store a *CellData* object containing

- a map cell and
- an integer value equal to the distance from that cell to the starting cell where the Middlesex College building is.

As the algorithm looks for a shortest path to the destination cell it will keep track of the distance from the current cell to the destination cell. A description of the algorithm and an example of how the algorithm works are given below.

1. First, create an empty ordered list using class *OrderedCircularArray*.
2. Get the starting cell where the Middlesex College building is located by using method *getStart()* from class *Map*. Each map cell is represented with an object of the class *MapCell*, described in Section 5.
3. Insert the starting cell in the ordered list with a distance value of zero (as the distance from the starting cell to itself is zero). Mark this cell as *in the ordered list* (use method *markInList()* from class *MapCell*).
4. **While** the list is not empty, **and** the destination cell has not been reached, perform the following steps:

    - Remove from the ordered list the map cell *S* with smallest distance value and mark it as *out of the ordered list* (to mark the cell use method *markOutList* from class *MapCell*).
    - If *S* is the destination cell, then the algorithm exits the while loop.

      Otherwise, the algorithm considers each one of the neighbouring cells of *S* that

        - is not null,
        - is not a block of houses or a park,
        - it has not been marked, and
        - it can continue the path from *S*

      To find these cells you will use your method *nextCell* described above. For each one of these neighbouring cells *C* perform the following steps:

        - Set *D* = 1 + distance from *S* to the starting cell.
        - If the distance between *C* and the starting cell (to find this distance use method *getDistanceToStart* from class *MapCell*) is larger than *D* then:
            - set the distance of *C* to the starting cell to *D* (to do this use method *setDistanceToStart* from class *MapCell*).
            - Set *S* as the predecessor of *C* in the path to the starting cell (use method *setPredecessor* from class *MapCell*); this is necessary to allow the algorithm to reconstruct the path from the initial cell to the destination once the destination has been reached.
        - Set *P* = distance from *C* to the starting cell.
        - If *C* is marked as *in the ordered list* and *P* is smaller than the distance value stored in the entry of the circular array storing *C* (to find this value use method *getValue*

from class *CellData*) then use the *changeValue* method from class *OrderedCircularArray* to update the distance value of *C* to *P*.

- If *C* is not marked as *in the ordered list*, then add it to the ordered list with distance value equal to *P*. Then mark *C* as *in the ordered list*.

Note. Recall that you are expected to use meaningful names for your variables when implementing this algorithm. So, do not use *C, P* and *S* as names for your variables; use more meaningful names.

To understand how the algorithm works, consider the map given in page 2. The algorithm starts at cell 1 and sets the distance from this cell to the starting cell to 0. Then it examines the neighboring cells; since the east and west neighbors are blocks of houses they are ignored. Cell 3 is added to the ordered list with a distance value of 1 and the predecessor of cell 3 is set to cell 1.

In the next iteration of the while loop, the algorithm removes the map cell 3 from the ordered list. Then two *CellData* objects are added to the ordered list containing map cells 2 and 4; for each one of these map cells the associated distance value is 2. To understand why the distance value for map cell 2 is 2, note that the shortest path between map cells 2 and 1 is the path 1, 3, 2; this path has length 2 (the distance between 1 and 3 is 1 and the distance between 3 and 2 is 1. Similarly, the shortest path between map cells 4 and 1 is the path 1, 3, 4, which also has length 2.

Next the algorithm removes from the ordered list the *CellData* object with smallest value, namely cell 4 (because map cells 2 and 4 have the same distance value and map cell 4 was added to the ordered list before map cell 2). From here the algorithm will examine the neighboring unmarked cells of map cell 4, namely 5 and 6. *CellData* objects storing map cells 5 and 6 are added to the ordered list, each with associated distance value 3; map cell 4 is set as the predecessor of map cells 5 and 6. In the next iteration map cell 2 will be removed from the ordered list.

The algorithm keeps examining cells in this manner until it reaches the destination cell, or it determines that the destination cannot be reached. When the destination cell is found, the code given to you will automatically highlight in red the path that your algorithm has found.

4. Command Line Arguments

The *main* method is in the provided class *CityMap*. The program will read the name of the input map file from the command line. From the console you can run the program as follows:

```
java CityMap name_of_map_file
```

where `name_of_map_file` is the name of the file containing the city map. To run the program from Eclipse you need to indicate what the name of the input file is. To do this, in Eclipse select "Run → Run Configurations...". Make it sure that "Java Application → CityMap" is the active selection on the left-hand side. Select the "Arguments" tab. Enter the name of the file for the map in the "Program arguments" text box. For more details check the tutorial posted in the course's website:

http://www.csd.uwo.ca/courses/CS1027b/passingParameters.html

5. Classes Provided

You can download from the course's webpage several java classes that allow your program to display the map on the screen. You are encouraged to study the given code to you learn how it works. Below is a description of some of these classes.

5.1 Class *CityMap*.java

This class represents the map of the city including the starting cell and the destination cell. The method that you might use from this class is the following:

- *public MapCell getStart()*. Returns a *MapCell* object representing the starting cell where the Middlesex College Building is located.

5.2 Class *MapCell*.java

This class represents the cells of the map. Objects of this class are created inside the class *CityMap* when its constructor reads the map file. The methods that you might use form this class are the following:

- *public MapCell getNeighbour (int i) throws InvalidNeighbourIndexException*. As explained above, each cell of the map has up to four neighbouring cells, indexed from 0 to 3. This method returns either a *MapCell* object representing the *i*-th neighbor of the current cell or *null* if such a neighbor does not exist. Remember that if a cell has fewer than 4 neighbouring cells, these neighbouring cells do not necessarily need to appear at consecutive index values. So, it might be for example, that *this.getNeighbour*(2) is null, but *this.getNeighbour*(*i*) for all other values of *i* are not null.
  An *InvalidNeighbourIndexException* is thrown if the value of the parameter *i* is negative or larger than 3.
- *public boolean* methods: *isBlock*(), *isIntersection*(), *isNorthRoad*(), *isEastRoad*(), *isSouthRoad(), isWestRoad(), isStart(), isDestination*(), return true if *this MapCell* object represents a cell corresponding to a block of houses or a park (where a car cannot drive), a road intersection, a north road, an east road, a south road, a west road, the staring cell where the Middlesex College Building is located, or the destination cell where your house is located, respectively.
- *public boolean isMarkedInList*() returns true if *this MapCell* object represents a cell that has been marked as *in the ordered list*.
- *public boolean isMarkedOutList*() returns true if ***this*** *MapCell* object represents a cell that has been marked as *out of the ordered list*.
- *public boolean isMarked()* returns true if ***this*** *MapCell* object represents a map cell that has been marked as *in* or *out of the ordered list*.
- *public void markInList*() marks ***this*** *MapCell* object as *in the ordered list*.
- *public void markOutList*() marks ***this*** *MapCell* object as *out of the ordered list*.
- *public int getDistanceToStart*(). Returns the distance from the cell represented by ***this*** *MapCell* object to the starting map cell.
- *public void setDistanceToStart(int dist)*. Sets to the specified value *dist* the distance from the cell represented by ***this*** *MapCell* object to the starting map cell.
- *public void setPredecessor(MapCell pred).* Sets the predecessor of ***this*** *MapCell* object to the specified value.
- *public Boolean equals(MapCell otherCell).* Returns true if *otherCell* points to ***this*** *MapCell* object, i.e. if **this** object and *otherCell* have the same address; otherwise it returns false.

5.3 Other Classes Provided

SortedListADT.java, CellColors.java, CellComponent.java, InvalidNeighbourIndexException.java, CellLayout.java, InvalidMapException.java, IllegalArgumentException.java, EmptyListException, InvalidDataItemException.

6. Image Files and Sample Input Files Provided

You are given several image files that are used by the provided java code to display the different kinds of map cells on the screen. You are also given several input map files that you can use to test your program. In Eclipse put all these files inside your project file in the same folder where the *src* folder is. ***Do not*** put them inside the *src* folder as Eclipse will not find them there. If your program does not display the map correctly on your monitor, you might need to move these files to another folder, depending on how your installation of Eclipse has been configured.

7. Submission

Submit all your .java files to OWL. **Do not** put the code inline in the textbox. **Do not** submit your *.class* files. If you do this and do not submit your *.java* files your program cannot be marked. **Do not** submit a compressed file with your java classes (.zip, .rar, .gzip, …). Do not put a "package" command at the top of your java classes.

8. Non-functional Specifications

1. **Assignments are to be done individually and must be your own work.** Software will be used to detect cheating.
2. Include comments in your code in **javadoc** format. Add javadoc comments at the beginning of your classes indicating who the author of the code is and a giving a brief description of the class. Add javadoc comments to the methods and instance variables. Read information about javadoc in the second lab for this course.
3. Add comments to explain the meaning of potentially confusing and/or major parts of your code.
4. Use Java coding conventions and good programming techniques. Read the notes about comments, coding conventions and good programming techniques in the first assignment.

9. What You Will Be Marked On

1. Functional specifications:
    - Does the program behave according to specifications? Does it run with the test input files provided? Are your classes created properly? Are you using appropriate data structures? Is the output according to specifications?
2. Non-functional specifications: as described above