COMP 1020 Winter 2016 Assignment 2

Due Date: Friday, Feb. 26, before midnight

1 MATERIAL COVERED

- File input/output and exceptions
- Two dimensional arrays
- Binary search (review) and object comparison

2 NOTES AND INSTRUCTIONS—PLEASE FOLLOW *EXACTLY*

- Follow the programming standards posted on the course website to avoid losing marks.
- You must complete the "Blanket Honesty Declaration" checklist on the course website, before you can submit any assignment.
- To submit the assignment you will upload the required files, as specified for each question, to the Dropbox for Assignment 2 on the course website.
- You will also put the output of your test runs from Dr. Java into .txt files, and upload them to the Dropbox in the same way.
- Name your output files <LastName><FirstName>A2Qn-output.txt unless some other name is specifically asked for.
- Do not manipulate your output in any way.
- Complete instructions for handing in your assignment, including an easy way to create the output files, can be found in the "Handin instructions" document in the Quick Links on the website.
- To be eligible to earn full marks, your Java programs must compile and run upon download, without requiring any modifications.
- Do not hand in zipped or otherwise compressed files. Hand in only and exactly those files specified in this assignment PDF, in the specified formats. Failure to do so will result in lost marks.

3 QUESTION 1: SIMPLE DATABASE CLASS

3.1 GENERAL DESCRIPTION

For this question, you will write a simple `Database` class that will enable the `CourseListQ2` class from Assignment 1 to load course information from a file and save the updated list back into a file. You may use your own versions of `Course.java` and `CourseListQ2.java`, or you may wait for the sample solution to be posted and use those. If you use the sample solutions, you MUST leave the first 4 comment lines in the file.

3.2 THE DATABASE CLASS

Create a new Java file, and name it `Database.java`. (Do not prefix it with your name.) Name the class appropriately to match. You have been provided with a main program file, called `TestA2Q1.java`.

Create two *instance* variables: `String` variables for the input and output file names. Both variables should be private. Please keep in mind that the input file and output file could be the same.

Create only a single constructor that accepts two `String` parameters (the input and output file names) and initializes the two instance variables accordingly.

You have been provided with a sample input file called `courses.txt`. Each line in the input file contains all of the information for one course: a CRN, title, department, term and year, separated by commas. A sample input file looks like this:

```
20161001,Advanced YouTube Commenting,Comp Sci,1,3
20162006,Zookeeping,Animal Science,2,1
20162003,Intermediate Cereal Cooking,Culinary Arts,2,2
20163004,Potato Trees,Agriculture,3,2
20161005,Paper Airplanes,Mechanical Engineering,1,4
20162003,Intermediate Cereal Cooking,Culinary Arts,2,2
20161002,Basket Weaving,Fine Arts,1,1
```

Write the following two methods for this class:

- `public CourseListQ2 load()` This method will read the comma separated input file (specified in the constructor), line by line, create a `Course` object for each line, and add it to a new `CourseListQ2` object that will be returned as the result.
- `public void save(CourseListQ2 courseList)` This method will store the CRN, title, department, term and year of each course in the `courseList` object, as a comma separated line, in the previously-specified output file, replacing anything that was in that file previously. Each course will be in a separate line.

To write the abovementioned methods some changes will be need to be made to your `Course` and `CourseListQ2` classes from Assignment 1.

3.2.1 Changes to the `Course` class

- Add a class variable called `nextCRN`, an integer, to store the sequence number that should be used for the CRN if a new course is created. As before, the sequence begins at 1, and it should always be adjusted so that it is equal to the largest sequence number that has ever been used, plus 1, to guarantee that it is a CRN number that has never been used before. You will no longer use the `numCourses` variable to generate CRN numbers.
- Add a new constructor to the `Course` class that accepts five parameters: the CRN, title, department, term and year of a course.

- Modify the other constructor (the one accepting four parameters: title, department, term and year) to use the `nextCrn` class variable to generate a course CRN.
- Both constructors must update the `nextCrn` class variable as described above.
- Add a `public String toCsv()` method to return an appropriate String to write into a file: the CRN, title, department, term and year separated by commas.

3.2.2 Changes to the `CourseListQ2` class

- Modify the `public void addCourse(Course)` method to throw an exception if the course that is being added has a duplicate CRN. Catch this exception in the `load()` method of the Database class (it should simply print the message stored in the Exception and continue running).
- Add two "accessor" or "getter" methods: `public int getNumCourses()` and `public Course getCourse(int i)` which will return the number of courses in the list, and the course with index (position) `i`. The `getCourse` method will throw an ArrayIndexOutOfBounds exception if the argument is not a valid array index (that is, it is < zero or ≥ numCourses).

3.3 SAMPLE OUTPUT (produced by the supplied TestA2Q1.java program)

```
Duplicate CRN: 20162003
20163008,Aristotle,Department of Philosophy,3,2
20162006,Zookeeping,Animal Science,2,1
20162003,Intermediate Cereal Cooking,Culinary Arts,2,2
20163004,Potato Trees,Agriculture,3,2
20161005,Paper Airplanes,Mechanical Engineering,1,4
20161002,Basket Weaving,Fine Arts,1,1
TestA2Q1: java.lang.ArrayIndexOutOfBoundsException: getCourse received out of bounds index of 6
End of processing.
Programmed by Stew Dent.
```

3.4 HAND-IN

Submit your `Database.java, Course.java,` and `CourseListQ2.java` files (but **not** the main program you were given) and also the `coursesOut.txt` file that is created by the main method (which uses your `save` method).

4 QUESTION 2: REMOVING DUPLICATES FROM A FILE

For this question, you will write the `A2Q2` class that will allow you to read a file, remove duplicate lines, and store the unique lines in a separate file. You have been given a sample input file called `words.txt`. To make it simple, each line in the input file will contain a single word. The output file should keep the words *in the same order* as the original input file.

4.1 THE A2Q2 CLASS

Create a new Java file, and name it `A2Q2.java`. (Do not prefix it with your name.) Name the class appropriately to match. You have been provided with a main program file, called `TestA2Q2.java`.

Create the following *instance* variables: an uninitialized array of type `String`, two `String` variables to store input and output file names, and an integer `numWords` that we will use to specify how many words are currently in the array. Make all instance variables `private`.

Define only one constructor that accepts two `String` parameters to initialize the input and output file names. It should also initialize the array of type `String` to a size of 20 (remember to use constants as appropriate), and set `numWords` to 0, because the list will initially contain no words.

Define the following methods:

- `private void insert(String newWord)` This method will insert `newWord` into the array, keeping the array sorted in ascending order. This method should expand the array if required. To do this, you will need to use an "ordered insertion" algorithm, as covered in COMP 1010. Remember that to compare two `String` values you cannot use < or >. You must use the `compareTo` method: `string1.compareTo(string2)` will give a negative number if `string1` is less than `string2`, a positive number if `string1` is greater than `string2`, and a 0 if they're identical.
- `private int indexOf(String newWord)` This method will perform a ***binary*** search on the array (remember that you keep the array sorted so that you can use the binary search algorithm) and return the index of `newWord` in the array, or -1 if `newWord` does not exist in the array.
- `public void removeDuplicateLines()` This method will read the input file line by line and if the current line has not been seen before, it will be written to the output file. Otherwise it will print a message saying that this word has been seen before.

4.2 SAMPLE INPUT FILE

```
hello
foo
teacher
comp1020
bar
hello
teacher
java
repeat
java
```

4.3 SAMPLE OUTPUT FILE (newWords.txt – produced by TestA2Q2)

```
hello
foo
teacher
comp1020
bar
java
repeat
```

4.4 SAMPLE OUTPUT

```
hello has been seen before!!
teacher has been seen before!!
java has been seen before!!
End of processing.
Programmed by Stew Dent.
```

4.5 HAND-IN

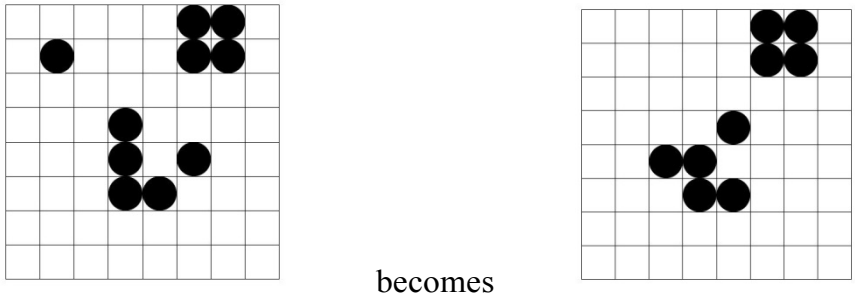
Submit your `A2Q2.java` file (but **not** the main program you were given) and also two text files: one containing the output of your program, named as specified in Section 2 (`YourNameA2Q2-output.txt`) and also the `newWords.txt` file generated by the main program using your `removeDuplicateLines()` method.

5 QUESTION 3: CONWAY'S GAME OF LIFE

John Conway invented his "Game of Life" in 1970. It is an example of a "finite automaton" where simple rules are used to create complex behavior. See

http://en.wikipedia.org/wiki/Conway's_Game_of_Life

The "game" (actually more like a simulation) is played on a board consisting of a square grid of "cells". Each cell can be "alive" or "dead" (empty). Therefore, a 2-dimensional `boolean` array is perfect to represent the board, with `true`=alive and `false`=dead. We'll draw small black circles for living cells, and leave dead cells blank. So two "generations" of a small board might look like this:



A new generation of cells is created from the current generation as follows. The eight cells surrounding a given cell (vertically, horizontally, or diagonally) are its "neighbours". The fate of any cell on the board is controlled by how many *living* neighbours it has, using these rules:

(Survival rule) If a cell is alive, then it stays alive in the next generation if it has either 2 or 3 living neighbours. (It dies of loneliness if it has 0 or 1, and dies of overcrowding if it has 4 or more.)

(Birth rule) If a dead (empty) cell has exactly 3 living neighbours, then it becomes a living cell in the next generation.

The living cell in the top left died of loneliness. The four in the top right survived, because they each have 3 living neighbours. In the middle, two cells died of loneliness, three survived, and two new cells were born because there were exactly 3 living neighbours in those places.

Implement a simulation of Conway's Game of Life.

1. Download the file `ConwaysLifeTemplate.java`. Save it as `ConwaysLife.java` before using it. You will also need `StdDraw.java`. The instance variables are already declared for you: the size of the board, the current board, and the next generation board.
2. Implement the two constructors whose headers are given in the file.
    a. Both constructors should initialize the `boardSize` variable, and also create two arrays – the board array will hold the current generation, and the `nextBoard` array will be used to calculate the next generation.

b. The constructor `ConwaysLife(int size, double density)` should fill the board with random cells. The parameter `density` will be a number from 0 to 1, giving the probability that each cell on the board will be alive (If 0, they'll all be dead. If 1, they'll all be alive. If 0.5 then each cell has a 50% chance of being alive.) Use the `Math.random()` method to determine whether each cell is alive or not. The `nextBoard` array must be created, but nothing else needs to be done to it yet.

c. The constructor `ConwaysLife(File chosenFile)` should create the initial configuration of the board from the indicated text file. The first line of this file will contain a single integer giving the size of the board (the number of rows and columns). Every other line will represent a row of the board, with one character per cell. An 'X' will be a living cell, and any other character will be a dead (empty) one. See the supplied test data files for examples. Set up the board array. The `nextBoard` array must be created, but nothing else needs to be done to it yet.

3. Implement the `int countNeighbours(int row, int col)` method, which will count the number of living neighbours of the cell in the given position in the `board` variable. Note that cells on the edge of the board, or in the corners, will have only 5 or 3 neighbours instead of 8. In other words, watch out for invalid subscripts on the edges. (In the real game, the board is considered to be infinite, but that's not practical to implement.)

4. Implement the `void nextGeneration()` method. This method should fill up the `nextBoard` instance variable to represent the next generation, according to the rules given previously. At the bottom of this method, code is supplied which will swap the two arrays, so that the next generation becomes the current generation, allowing the process to be repeated indefinitely.

5. A `toString()` method is already supplied, and so is a `draw()` method which will draw the board using `StdDraw` graphics.

Test your class by using the supplied `TestLife.java` program. It will allow you to choose which constructor to use, select a file, control how quickly the simulation runs, and produce some printed output for debugging purposes, and to hand in.

5.1 HAND-IN:

Submit your completed `ConwaysLife.java` file. Also hand in **one** file, named as specified in Section 2, containing the results of the following **two** test runs: 1) show 5 generations produced by the file `LifeSample.txt` (which contains the example shown earlier), and 2) show 5 generations of a random board of size 12. (Three other files are supplied which you can experiment with just for fun: `GliderGun.txt` , `GlidersCollide.txt`, and `Harvester.txt` You do not need to run these or hand them in.)