Concordia University
Comp 249 – Winter 2018
Object-Oriented Programming II
Assignment 4 – Due 11:59 PM - Sunday April 8, 2018

Purpose: The purpose of this assignment is to allow you practice Interfaces, Inner classes, LinkedList, and ArrayList, as well as other previous Object-Oriented concepts such as Exception Handling and File I/O.

Problem statement

"Education is what remains after one has forgotten what one has learned in school."

-- Albert Einstein

Starting a new phase in education can be an exciting time. The time when you decide step into the world of post-secondary education (college or university). There are a lot of interesting and challenging subjects that you can take, however you should have the right foundation to succed in these courses.

In order to make sure that you have the required skillset to succeed in a course, courses have *pre-requites* and *co-requisites*. A *pre-requisite* is one or more courses that you must complete before taking a course at higher grade level. To be accepted into some courses, you will have to prove that you have completed a similar course in the same or a related subject, at a lower grade level. Prerequisites are usually in the same or a related subject, at a lower grade level. A *co-requisite* on the other hand is a slightly relaxed situation where you may take a specific course if you have finished the co-requisite earlier or even if you register for the co-requisites in the same term. Courses can have both pre-requisites and co-requisites.

As an example, at Concordia University COMP248 is a pre-requisite for COMP249 and COMP371 is a co-requisite for COMP376. Furthermore, a course can have a same course as both pre-requisite as well as co-requisite (e.g. COMP371 is both pre-requisite and co-requisite for COMP376)

In this assignment, you will design and implement a tool which will determine if a student can enrol in a specific course based on courses he/she has taken so far. You are given two files, namely, *Syllabus.txt* containing information about various courses, and *Request.txt* which contains student enrollement request information (courses completed so far and courses he/she wishes to enrol in). You will parse these files to extract course information and will produce an outcome for each of the course student wants to enrol in. The outcome for each course could be one of the below mentioned options where X represents the course ID, P represents coursID for pre-requisite and C represents courseID for co-requisite.

a) Student can enrol in X course as he/she has completed the pre-requisite P.
b) Student can enrol in X course as he/she is enroling for the co-requisite C.
c) Student can't enrol in X course as he/she doesn't have sufficient background needed.

You can assume that every course can have minimum of zero and maximum of one pre-requisite as well as co-requisite. You can also assume that the courses will be listed from basic to advanced to help parse them easily. *Request.txt* contains a word "Finished" on first line, followed by courseID of those courses. Another word "Requested" after the above information followed by the respective courseIDs. *Syllabus.txt* contains courseID along with course name (one word seperated by _) and credits assigned to that course. Next two lines will have P and C indicating pre-requisite and co-requisite respectively fo this course. This set of information is repeated for all the available courses. A sample *Syllabus.txt* file is depicted in Figure 1 and a sample *Request.txt* is depicted in Figure 2.

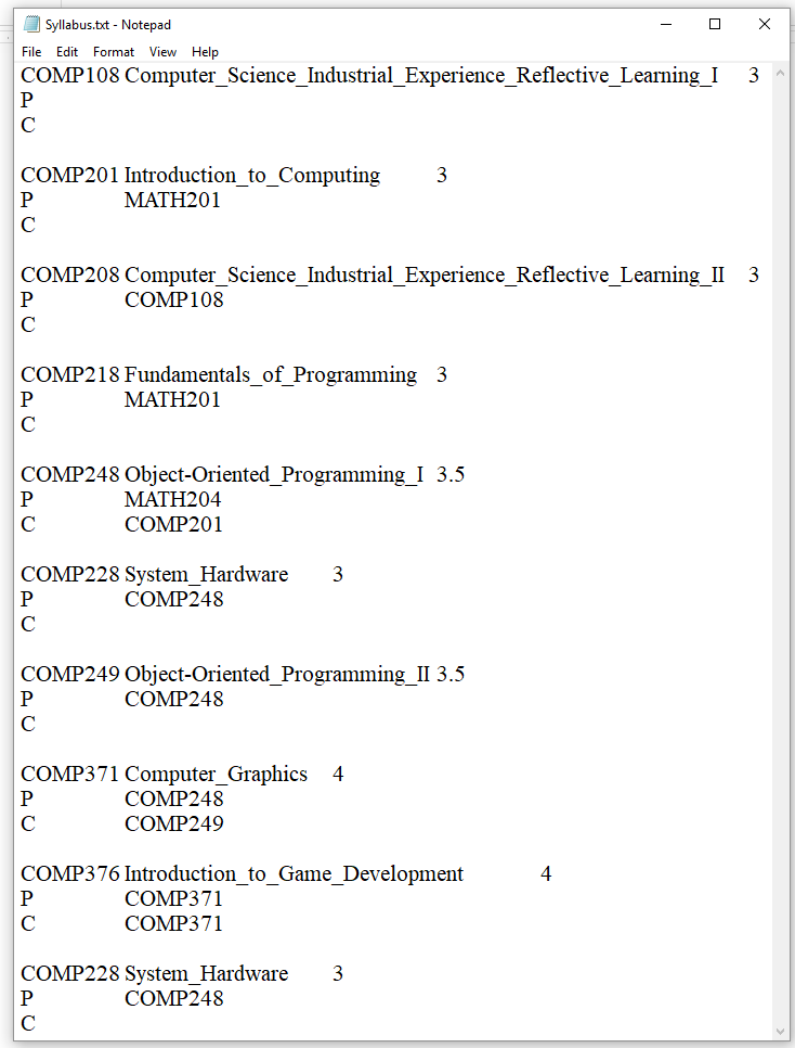

Figure 1.

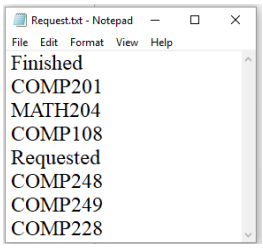

Figure 2.

Figure 1. Illustration of Syllabus.txt where P represents pre-requisite and C represents co-requisite
Figure 2. Illustration of Request.txt

Requirements

Task1: Create an interface class named **DirectlyRelatable** which has a boolean method named:

isDirectlyRelated(Course C) where *C* is a object of type Course described in the next task.

Task2: Create the **Course** class has the following five attributes: a *courseID (String type)*, a *courseName (String type)*, a *credit (double type)*, a *preReqID (String type)*, *coReqID (String type)*. It is assumed that course name is always recorded as a single word (_ is used to combine multiple words). It is also assumed that no two courses can have the same courseID.
You are required to write the implementation of the Course class. Besides the usual mutator and accessor methods (i.e. *getCourseID()*, *setCoursename()*) the class must also have the following:

1. a parameterized constructor that accepts five values and initializes *courseID, courseName, credit, preReqID, coReqID* to these passed values.
2. a copy constructor, which takes in two parameters, a Course object and a String value. The newly created object will be assigned all the attributes of the passed object, with the exception of the courseID. courseID is assigned the value passed as the second parameter to the constructor. It is always assumed that this value will correspond to the unique courseID rule.

3. *clone()* method. This method will prompt the user to enter a new courseID, then creates and returns a clone of the calling object with the exception of the courseID, which is assigned the value entered by the user.
4. additionally, the class should have a *toString()* and an *equals()* methods. Two courses are equal if they have the same attributes, with the exception of the courseID, which could be different.
5. This class needs to implement the interface DirectlyRelatable from task1. The method *isDirectlyRelated* that takes in another Course object C and should return `true` if C is pre-requisite or co-requisite of the current course object.

Task3: Create the **CourseList** class that has the following:

1) An inner class called **CourseNode**. This class has the following:
   i. two private attributes: an object of Course and a pointer to a CourseNode object.
   ii. a default constructor, which assigns both attributes to null.
   iii. a parameterized constructor that accepts two parameters, a Course object and a CourseNode object, then initializes the attributes accordingly.
   iv. a copy constructor.
   v. a *clone()* method.
   vi. other mutator and accessor methods.
2) A private attribute called head, which should point to the first node in this list object.
3) A private attribute called size, which always indicates the current size of the list (how many nodes are in the list).
4) A default constructor, which creates an empty list.
5) A copy constructor, which accepts a CourseList object and creates a copy of it;
6) A method called *addToStart()*, which accepts one parameter, an object from Course class and then creates a node with that passed object and inserts this node at the head of the list;
7) A method called *insertAtIndex()*, which accepts two parameters, an object from Course class, and an integer representing an index. If the index is not valid (a valid index must have a value between 0 and size-1), the method must throw a *NoSuchElementException* and terminate the program. If the index is valid, then the method creates a node with the passed Course object and inserts this node at the given index. The method must properly handle all special cases.
8) A method called *deleteFromIndex()*, which accepts one integer parameter representing an index. Again, if the index is not valid, the method must throw a *NoSuchElementException* and terminate the program. Otherwise; the node pointed by that index is deleted from the list. The method must properly handle all special cases.
9) A method called *deleteFromStart()*, which deletes the first node in the list (the one pointed by head). All special cases must be properly handled.
10) A method called *replaceAtIndex()*, which accepts two parameters, an object from Course class, and an integer representing an index. If the index is not valid, the method simply returns; otherwise the object in the node at the passed index is to be replaced by the passed object.
11) A method called *find()*, which accepts one parameter of type String representing a courseID. The method then searches the list for a CourseNode with that courseID. If such an object is found, then the method returns a pointer to that CourseNode; otherwise, method returns null. The method must keep track of how many iterations were made before the search finally finds the course or concludes that it is not in the list.
12) A method called *contains()*, which accepts one parameter of type String representing a *courseID*. The method returns `true` if the a course with that *courseID* is in the list; otherwise, the method returns `false`;

13) A method called *equals()*, which accepts one parameter of type CourseList. The method returns `true` if the two lists contain similar courses; otherwise the method returns `false`. Recall that two Course objects are equal if they have the same values with the exception of the *courseID*, which can, and actually is expected to be, different.

A sample CourseList is demonstrated in Figure 3.

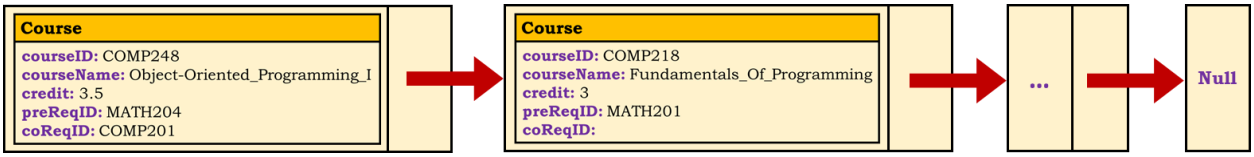

Figure 3. A sample CourseList (LinkedList implementation)

Finally, here are some general rules that you must consider when implementing the above methods:

- Whenever a node is added or deleted, the list size must be adjusted accordingly.
- All special cases must be handled, whether or not the method description explicitly states that.
- All *clone()* and copy constructors must perform a deep copy; no shadow copies are allowed.
- If any of your methods allows a privacy leak, you must clearly place a comment at the beginning of the method (a) indicating that this method may result in a privacy leak (b) explaining reason behind the privacy leak. Please keep in mind that you are not required to implement these proposals.

Task4: Write a public class called **EnrolmentResults** that has the *main()* method, you must do the following:

1) Create at least two empty lists from the CourseList class (needed for copy constructor, see task 3.4).
2) Open the *Syllabus.txt* file, and read its contents line by line. Use these records to initialize one of the CourseList objects you created above. You can use the *addToStart()* method to insert the read objects into the list. However, the list should not have any duplicate records, so if the input file has duplicate entries, which is the case in the file provided with the assignment for instance, your code must handle this case so that each record is inserted in the list only once.
3) Open *Request.txt* (prompt the user to enter the name of the file that need to be processed: e.g. *Request3.txt*) and create an ArrayList from the contents then iterate through each of the courses the student wants to enroll in. Process each of the courses and print the outcome whether student will be able to enroll or not. A sample output for a given file is mentioned below. Your program should ask for the file names as your program will be tested against similar input files.
4) Prompt the user to enter a few *courseIDs* and search the list that you created from the input file for these values. Make sure to display the number of iterations performed.
5) Following that, you must create enough objects to test each of the constructors/methods of your classes. The details of this part are left as open to you. You can do whatever you wish as long as your methods are being tested including some of the special cases.

```
Student can enrol in COMP248 course as he/she has completed the pre-requisite MATH204.
Student can't enrol in COMP249 course as he/she doesn't have sufficient background needed.
Student can't enrol in COMP228 course as he/she doesn't have sufficient background needed.
```

Figure 4. A sample outcome of the enrolment request

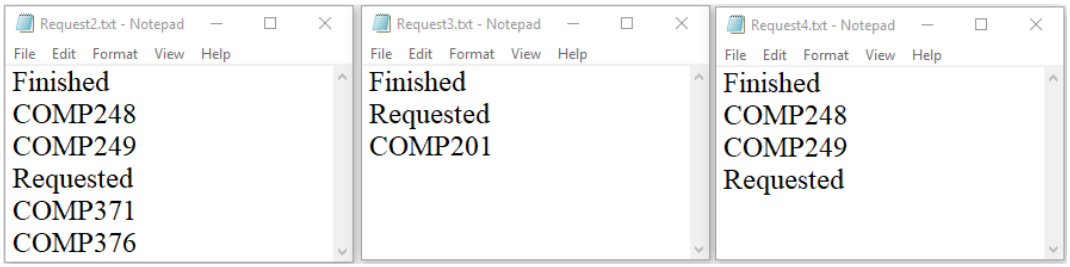

Figure 5. *Sample Request.txt* files

Outputs for other sample *request.txt* files shown in Figure 5 are depicted below.

```
Student can enrol in COMP371 course as he/she has completed the pre-requisite COMP248 and COMP249.
Student can enrol in COMP376 course as he/she is enrolling for co-requisite COMP371.
```

(a)

```
Student can't enrol in COMP201 course as he/she doesn't have sufficient background needed.
```

(b)

```
No enrollment courses found.
```

(c)

Figure 6. Outputs for (a) *Request2.txt*, (b) *Request3.txt*, and (c) *Request4.txt* respectively

General Guideline information:

- You should open and close the *Syllabus.txt* file only once; a better mark will be given to this;
- Do not use any external libraries or existing software to produce what is needed; that will directly result in a 0 mark!
- Your program must work for any input files. The files provided with this assignment are only a possible version, and must not be considered as the general case when writing your code.

General Guidelines When Writing Programs

- Include the following comments at the top of your source code (each .java file)
  - `// ------------------------------------------------------`
  - `// Assignment# (include number)`
  - `// Question: (include question/part number, if applicable)`
  - `// Written by: (include student name and id if assignment done`
    `//by one student or both students if assignment done by two`
    `//students`
  - `// ------------------------------------------------------`
- In a comment, give a general explanation of what your program does. As the programming questions get more complex, the explanations might get a bit longer.
- Display clear prompts for users when you are expecting the user to enter data from the keyboard.
- All output should be displayed with clear messages and in an easy to read format.
- End your program with a closing message so that the user knows that the program has terminated.

Assignment#4 Submission

- For this assignment, you are allowed to work individually, or in a group of **2 students maximum.** You and your teammate must however be in the same section of the course.
- If working in a group, **only one** of the two members submit/upload the assignment.
- Only electronic submissions will be accepted. Zip together the source code files.
- Students will have to submit their assignment (one copy per group) using Moodle.
- Assignments must be submitted in the right DropBox of the assignment.
- **Naming convention for zip file:** Create one zip file, containing all source files and produced documentations for your assignment using the following naming convention:
  - The zip file should be called *a#_StudentName_StudentID*, where # is the number of the assignment and *StudentName/StudentID* is your name and ID number respectively. Use your "official" name only - no abbreviations or nick names; capitalize the usual "last" name. Inappropriate submissions will be heavily penalized. For example, for the first assignment, student 12345678 would submit a zip file named like: *a4_Mike-Simon_123456.zip.* If working in a group, the name should look like: *a4_Mike-Simon_12345678-AND-Linda-Jackson_98765432.zip.*
- Submit only ONE version of an assignment. If more than one version is submitted the first one will be graded and all others will be disregarded.
- **Finally, notice that a demo is required for the assignment. Failure to do your demo, or missing you reserved demo time, will result in a zero mark for the assignment regardless of your submission.** ***There will be no substitution for a missed demo time.*** **Please see course outline for further details.**

Evaluation Criteria

Task #1	1 pt
Task #2	2 pt
Task #3	3 pt
Task #4	3 pt
JavaDoc documentations and general quality of your software application	1 pt
Total	**10 pts**