
Purpose: The purpose of this assignment is to allow you practice Interfaces, Inner Classes, ArrayLists and Linked Lists, as well as other previous object-oriented and File I/O concepts.

“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 succeed in these courses.

In order to make sure that you have the required skillset to succeed in a course, courses have pre-requisites and co-requisites. A prerequisite 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.

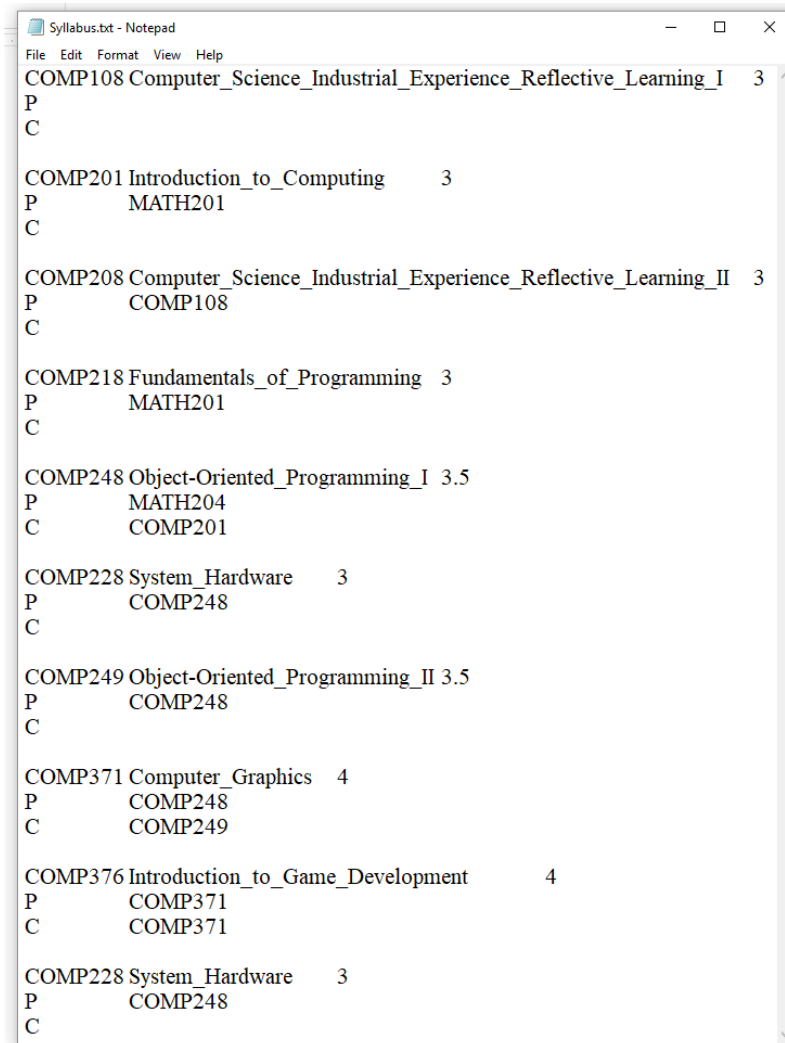
As an example, at Concordia University COMP248 is a pre-requisite for COMP249 and COMP371 is a co-requisite for COMP376. Furthermore, a course can have a same course as both pre-requisite as well as co-requisite (e.g. COMP371 is both pre-requisite and co-requisite for COMP376)

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 few files, namely, Syllabus.txt containing information about various courses, and some Request.txt files (i.e. Request1.txt, Request2.txt, etc.). Each of these files contains one student enrolment request information (courses completed so far and courses he/she wishes to enrol in). Four of these files are provided with the assignment; however, any other similar files can be given and you should expect that your program will be tested with other Request.txt files. 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 courseID, P represents courseID 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 enrolling 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 separated by _) and credits assigned to that course. Next two lines will have P and C indicating pre-requisite and co-requisite respectively for 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. A detailed description of all the details that you have to implement for this assignment is available after the two figures.

A screenshot of a Notepad window titled 'Syllabus.txt - Notepad'. The window contains a list of computer science courses with their prerequisites or co-requisites. Each entry consists of a course ID, name, and credit value, followed by a line with 'P' for pre-requisite or 'C' for co-requisite and the prerequisite course ID. The courses listed are: COMP108 Computer_Science_Industrial_Experience_Reflective_Learning_I (3 credits), COMP201 Introduction_to_Computing (3 credits), COMP208 Computer_Science_Industrial_Experience_Reflective_Learning_II (3 credits), COMP218 Fundamentals_of_Programming (3 credits), COMP248 Object-Oriented_Programming_I (3.5 credits), COMP228 System_Hardware (3 credits), COMP249 Object-Oriented_Programming_II (3.5 credits), COMP371 Computer_Graphics (4 credits), COMP376 Introduction_to_Game_Development (4 credits), and another entry for COMP228 System_Hardware (3 credits).

```
COMP108 Computer_Science_Industrial_Experience_Reflective_Learning_I 3
P
C

COMP201 Introduction_to_Computing 3
P MATH201
C

COMP208 Computer_Science_Industrial_Experience_Reflective_Learning_II 3
P COMP108
C

COMP218 Fundamentals_of_Programming 3
P MATH201
C

COMP248 Object-Oriented_Programming_I 3.5
P MATH204
C COMP201

COMP228 System_Hardware 3
P COMP248
C

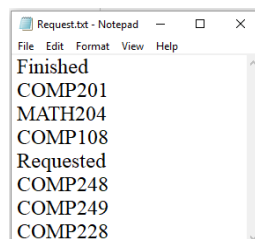
COMP249 Object-Oriented_Programming_II 3.5
P COMP248
C

COMP371 Computer_Graphics 4
P COMP248
C COMP249

COMP376 Introduction_to_Game_Development 4
P COMP371
C COMP371

COMP228 System_Hardware 3
P COMP248
C
```

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

A screenshot of a Notepad window titled 'Request.txt - Notepad'. The window contains a list of course IDs: COMP201, MATH204, COMP108, COMP248, COMP249, and COMP228. The word 'Finished' is at the top, and 'Requested' is below it.

```
Finished
COMP201
MATH204
COMP108
Requested
COMP248
COMP249
COMP228
```

Figure 2. Illustration of Request.txt

I) Create an interface named **DirectlyRelatable** which has a boolean method called `isDirectlyRelated` (Course C) where C is a object of type Course described in the next part.

II) The **Course** class, which must implement the **DirectlyRelatable** interface, has the following 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:

- a) Parameterized constructor that accepts five values and initializes courseID, courseName, credit, preReqID, coReqID to these passed values;
- b) 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;
- c) 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;
- d) 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.
- e) This class needs to implement the interface from part I. 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, or vice versa (hence the courses are directly related); otherwise it returns false.

III) The CourseList class has the following:

- (a) 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.
- (b) A private attribute called head, which should point to the first node in this list object;
- (c) A private attribute called size, which always indicates the current size of the list (how many nodes are in the list);
- (d) A default constructor, which creates an empty list;
- (e) A copy constructor, which accepts a **CourseList** object and creates a copy of it;
- (f) 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;
- (g) 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;
- (h) 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;
- (i) A method called deleteFromStart(), which deletes the first node in the list (the one pointed by head). All special cases must be properly handled.
- (j) 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;

- (k) 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;
- (l) A method called `contains()`, which accepts one parameter of type `String` representing a `courseID`. The method returns `true` if a course with that `courseID` is in the list; otherwise, the method returns `false`;
- (m) A method called `equals()`, which accepts one parameter of type `Syllabus`. 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.

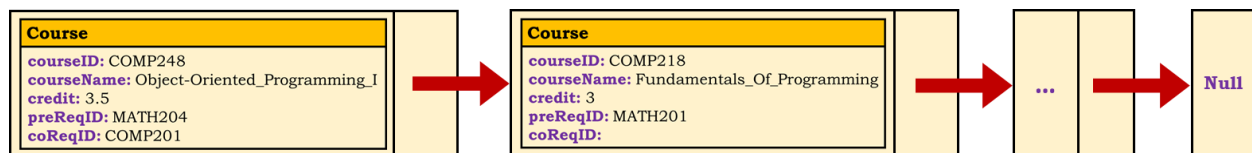


Figure 3. A sample `CourseList` (LinkedList implementation)

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 shallow copies are allowed;
- If any of your methods allows a privacy leak, you must clearly place a comment at the beginning of the method 1) indicating that this method may result in a privacy leak 2) explaining reason behind the privacy leak. Please keep in mind that you are not required to implement these proposals;

IV) Now, you are required to write a public class called **EnrolmentResults**. In the `main()` method, you must do the following:

- (a) Create at least two empty lists from the `CourseList` class (needed for copy constructor III (e));
- (b) 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;
- (c) Open a `Request.txt` file (prompt the user to enter the name of the file that need to be processed; i.e. `Request3.txt`) and create **ArrayLists** 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. Again, your program should ask for the file names as your program will be tested against similar input files;
- (d) 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;
- (e) 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. You should also make sure to test the `isDirectlyRelated()` method.

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

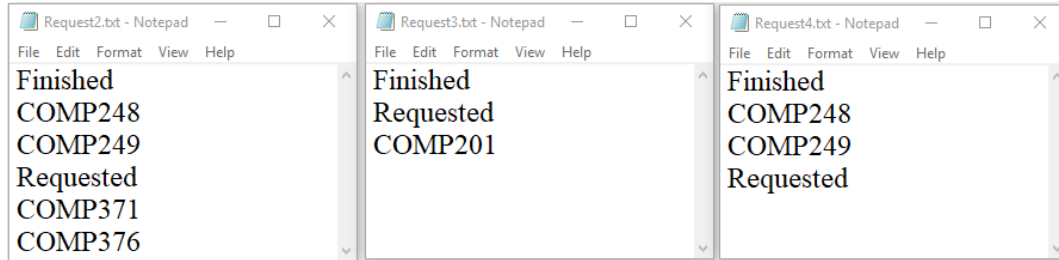


Figure 5. Sample Request.txt files

Outputs for the 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

Some general information:

- You should open and close the Syllabus.txt file only once; a better mark will be given for that;
- Do not use any external libraries or existing software to produce what is needed; that will directly result in a 0 mark!
- Again, 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 codes
// -----
// Assignment (include number)
// Question: (include question/part number, if applicable)
// Written by: (include your name and student id)
// -----
- In a comment, give a general explanation of what your program does. As the programming questions get more complex, the explanations will get lengthier.
- Include comments in your program describing the main steps in your program.
- Display a welcome message which includes your name(s).
- 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.

JavaDoc Documentation:

Documentation for your program must be written in **javaDoc**.

In addition, the following information must appear at the top of each file:

Name(s) and ID(s) (include full names and IDs)
 COMP249
 Assignment # (include the assignment number)
 Due Date (include the due date for this assignment)

Submitting Assignment 4

- For this assignment, you are allowed to work individually, or in a group of a maximum of 2 students (i.e. you and one other student). You and your teammate must however be in the same section of the course. Groups of more than 2 students = zero mark for all members!
- Only e-submissions will be accepted. Zip together the source codes. (Please use WINZIP).
- Students will have to submit their assignments (one copy per group) using the Moodle/EAS system (please check for your section submission). Assignments must be submitted in the right DropBox/folder of the assignments. **Assignments uploaded to an incorrect DropBox/folder or submitted via email will not be graded and will result in a zero mark. No resubmissions will be allowed. If using Moodle, please make sure you submit on Moodle system making use of the Assignment4 submission link under the week of April 6.**
- 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: *a1_Mike-Simon_123456.zip*. If working in group, the name should look like: *a1_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.
- If working in a team, only one of the members can upload the assignment. Do NOT upload the file for each of the members!

Evaluation Criteria for Assignment 4 (10 points)

Total	10 pts
JavaDoc documentations	1 pt
Task I	1 pt
Task II	2 pts
Task III	3 pts
Task IV	3 pt