Homework 8

This assignment provides a solid hands-on experience in object-oriented programming, linked lists, and system design. It also illustrates what can be described as "programming at the large", i.e. working with multiple classes and packages.

A few days ago we published Part I of this assignment, and asked that you submit it. After a second thought we decided to treat Part I as a self-study exercise, preparing you to work with linked lists. Therefore, there is no need to submit Part I, but rather view it as an essential hands-on introduction to the current project.

1. College Management System

You were hired to assist a system architect in building a *College Management System*, designed to help academic institutions run their operations. A typical college maintains information about various entities like *courses*, *students*, *instructors*, *rooms*, and so on. In addition, the college maintains various records, e.g. the fact that a student took a course and got a grade in it. To keep things simple, in this project we'll focus on three entities only: *courses*, *students*, and *courseTaken* records. This information will be represented by the three classes described below.

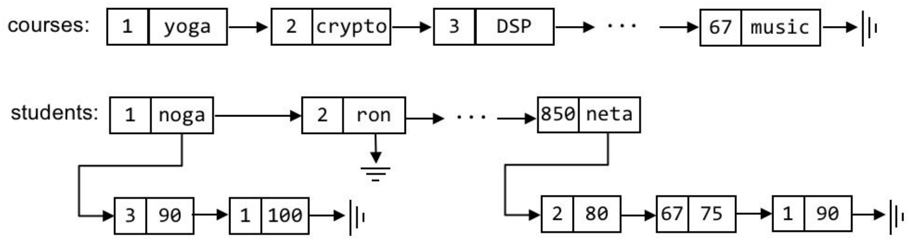
Course (`Course.java`): A typical college offers hundreds of courses, and each can be represented as an instance of this class. Typically, a course is characterized by a course id, course title, course instructor (an instance of an `Instructor` class), capacity, room (an instance of a `Room` class), etc. In this project though, the `Course` class features two fields only: course id, and title.

Student (`Student.java`): Represents students who are enrolled in the college. Each student has a student id, name, address, year, major, course list, and so on. In this project though, the `Student` class features three fields only: student id, name, and course list. The latter is a list of (course id, grade) pairs, each being an instance of the `CourseTaken` class.

CourseTaken (`CourseTaken.java`): Represents pairs of the form (*course id*, *grade*). It would make sense to also keep track of year and semester data, but, to keep things simple, we'll focus on course id and grade only.

Take a few minutes to inspect the classes `Course.java`, `Student.java`, and `CourseTaken.java`.

The system architect decided to base the system on a list of `Course` objects and a list of `Student` objects. Each student has a list of `CourseTaken` objects. For example:



Abstract (client) view of the courses and students lists

In this example, the college offers 67 courses, and enrolls 850 students. Noga got 90 in DSP and 100 in Yoga, and Neta got 80, 75, and 90 in Crypto, Music, and Yoga, respectively. Ron did not take any courses yet.

After interviewing representative administrators and prospective users from various college departments, the system architect made a list of the most required college management operations:

- Add / delete courses.
- Add / delete students.
- Add / delete courses and grades of students.
- Student report: Student name, followed by all the courses that s/he took, grades that s/he got, and the student's grade point average.
- Course report: Course name, followed by all the students who took the course.
- Enrollment report: The number of students who took a given course.
- Top performer report: The name of the student who got the highest grade in the course.

In order to deliver this functionality, the system architect decided to write a `College` class, and represent the entire college as an object. The `College` object will have *courses* and *students* properties (fields), each implemented as a linked list of `Course` and `Student` objects, respectively. Each college management operation will be implemented by a separate `College` method. The architect also wrote a `CollegeDemo` class, designed to create a college and test its various methods.

Appendix A shows an example of the system execution and output. Take a look.

2. Generic Linked Lists

A generic linked list can be used for creating and managing lists of any type of object. For example, a list of courses (instances of a `Course` class), a list of students (instances of a `Student` class), and so on. This versatility can be achieved by implementing three generic classes: a generic `Node` class, a generic `LinkedList` class, and a a generic `ListIterator` class.

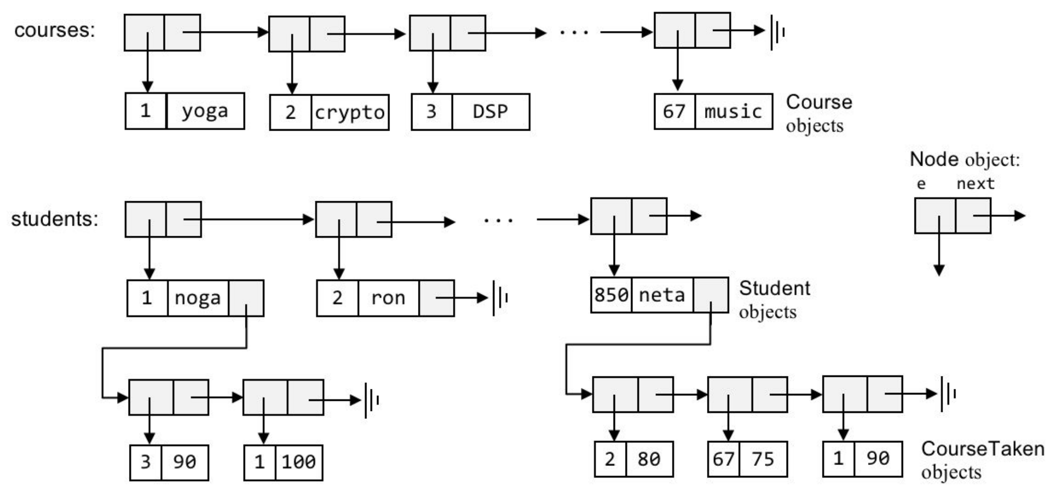
Node.java: In lecture 10-1 and in Part I of this homework, the `Node` class featured two fields: an `int` field named `data`, and a `Node` field named `next`. In the generic version of the `Node` class, the type and name of the `data` field were changed to `E` and `e`, respectively. We've chosen these symbols to denote that this generic node is designed to hold (refer to) an object that we call `e`, of any given type that we call `E`. Take a look at the supplied `Node.java` class.

So, now that we have a generic `Node` class, we can create `Node` objects that hold (refer to) objects of any desired type. For example, consider the following code:

```
Course c = new Course(1,"Introduction to CS");
Node<Course> node = new Node<Course>(c);
System.out.println(node);  // Prints "Course 1: Introduction to CS"
```

The first statement declares and constructs a `Course` object. Nothing new here. The second statement declares a new `Node` object in which the generic `E` type is set to the specific type `Course`, and calls the generic `Node` constructor, passing to it the new `Course` object that was just created. The result will be a single `Node` object in which `e` contains a reference to the course object referred to by `c`, and `next` contains `null`. Do you understand how the last statement ends up printing the course information? To be sure, take a look at the supplied `Course.java` and `Node.java` classes. Note: we sometimes refer to the generic field `e` as "element", since it plays the role of a list element.

LinkedList.java: Now that we have a generic `Node` class, we can write a generic `LinkedList` class. This class will manage linked lists of any desired type. For example, the abstract view of the data model presented in page 1 can be realized as follows:



Implementation of the courses and students lists

The courses list shown above can be constructed using the following client code, that can appear in any class:

```
LinkedList<Course> courses = new LinkedList<Course>();
courses.add(new Course(1,"yoga"));
courses.add(new Course(2,"crypto"));
courses.add(new Course(3,"DSP"));
...
System.out.println(courses);
```

The first statement in the client code creates a new linked list of `Course` objects. The next three statements create three new `Course` instances, and add them to the list. The last statement calls the `toString` method of the `LinkedList` class. It's important to note how the client code doesn't ever mention `Node` objects. `Node` objects are created and managed only by the `LinkedList` class.

In order to provide this clean interface, we must encapsulate all the code that handles the `Node` objects inside the code of `LinkedList`, without exposing it to the clients. Take a look at the `LinkedList.java` class, focusing on (i) the field definitions, (ii) the constructor, and (iii) the `add` method. Notice how the class code "funnels" the `E` parameter down to the `Node` level, causing the system to create node objects that hold references to objects of type `E`. Make sure that you understand how the class executes the client code shown above.

ListIterator.java: This class provides iteration services over generic linked lists. Normally, clients don't access this class directly. Rather, they can access the `iterator()` method of the `LinkedList` class. This method returns a `ListIterator` object that can be used for iterating any list.

It’s important to compare the abstract/client view of the lists, shown in page 1, with the lists implementation shown in page 3. The abstract view is used by programmers who create and manipulate lists using the services of the generic `LinkedList` class. These programmers are completely oblivious of (know nothing about) the `Node` objects. They don’t even need to know that the `Node` class exists. All they know, and care about, is that they can create and manipulate lists using `LinkedList` methods, which are documented in the `LinkedList` API. With that in mind, note that the code shown at the bottom of page 2 is not client code. Rather, it illustrates what can be done within the `LinkedList` class.

3. Packaging

The College Management System consists of eight classes. In terms of usage, these classes fall into three categories:

Data: The `Course`, `Student`, and `CourseTaken` classes serve as templates for creating data. The instances of these classes are the “atoms” from which the *courses* and *students* lists are built. These classes are simple, except for the `Student` class that has some unimplemented methods.

Infrastructure: Taken together, the `Node`, `LinkedList`, and `ListIterator` classes form a package that can be used for creating and managing linked lists of objects of any type. We named this package `linkedList`. Users of these classes need know nothing about their implementation; All they need is the API of the `LinkedList` class.

Control: The `College` and `CollegeDemo` classes “run the show”. In particular, they feature methods that create and manipulate the courses and students lists. They do so by using the services of the `LinkedList` class.

In general, if some class `Foo` wants to create and manage linked lists using our implementation, it needs to have the statement `import linkedlist.*` at the top of the class code. This will allow any method in `Foo`‘s code to call any method in the imported classes. In our system, both the `College` class and the `Student` class need to use the services of the `linkedList` package.

4. Exceptions

The final version of the College Management System must be “bullet proof”, meaning that no user action should make it crash. For example, if a user tries to get en element from an empty list, the system should display a gentle error message, and continue running as usual. In order to realize this behavior, the system architect must anticipate every problematic action, and then protect against this contingency by planting exception throwing and handling (`try/catch`) statements in strategic points in the code.

In this project, we don’t expect you to handle all the possible exceptions. Instead, we’ll illustrate exceptions throwing and handling using two examples, both in the `LinkedList` class. In particular, notice that the API of the `add(e,index)` method documents that if the given index is negative, or greater than the list size, the method throws an exception. Similarly, the APIs of the `getFirst()` and `getLast()` methods document that if the list is empty, the method throws an exception.

In order to test these exceptions, we wrote some testing code in the `main` method of `LinkedList`. As part of the homework requirements, you have to complete the exception throwing and handling code, in these examples only.

5. What to Do

Complete all the missing code in the classes `LinkedList.java`, `Student.java`, and `College.java`. The missing code is marked by the comments "`// Put your code here`" and "`Replace the following statement with your code`".

Implementation and testing guidelines

We recommend proceeding in the following order:

1. Start by completing the implementation of the `LinkedList` class. Note that the class constructor and the `add` method are fully implemented. Therefore, you can execute the class and watch how its `main` method creates and prints a list of `Integer` objects.

2. Next, implement all the `LinkedList` methods, in any order that suits you. After implementing each method, test it by writing and executing testing code in the `main` method of `LinkedList`. You can create and play with lists of `Integer` objects, as we did.

4. Implement and test the `College` methods, in the order in which they appear in the `College` class, up to, and including, the `getStudent` method (we'll implement the remaining methods later). Notice that the private `getCourse` and `getStudent` methods are quite handy, and can help in the implementation of other `College` methods.

As you implement each `College` method, test it by writing testing code in the `CollegeDemo` class. You don't have to submit the `CollegeDemo` class, so feel free to do whatever you want with it.

5. Switch to the `Student` class, and implement the `addCourse` method. Go back to the `College` class, and implement and test the `addCourseTaken` method.

6. Implement all the remaining methods in the `Student` and `College` classes, and test them.

"Wet Testing:" After you submit your implemented classes, we will test them by executing our own version of the `CollegeDemo` class. This will include detailed tests of every method.

Submission

Before submitting your solution, inspect your code and make sure that it is written according to our **Java Coding Style Guidelines**. Also, make sure that each program starts with the program header described in the **Homework Submission Guidelines**. Both documents can be found in the Moodle site, it is your responsibility to read them. Any deviations from these guidelines will result in points penalty. Submit the following file only:

- ➢ `Student.java`
- ➢ `LinkedList.java`
- ➢ `College.java`

Deadline: Submit Homework 8 no later than January 8, 2019, 23:55. You are welcome to submit earlier.

Appendix A. The CollegeDemo class

```
/** Tests some of the services of the College class. */
public class CollegeDemo {

      public static void main(String []args) {

            College c = buildCollege();

            c.studentsList();    // Prints the students

            c.removeStudent(0);  // Removes the first student in the list
            c.removeStudent(4);  // Removes a student somewhere inside the list
            c.removeStudent(5);  // Removes the last student in the list

            c.studentsList();    // Prints the students after the removals

            c.coursesList();

            c.studentReport(1); System.out.println();
            c.studentReport(2); System.out.println();

            c.courseReport(1);
            c.courseReport(2);

            c.printSize(1);  System.out.println();

            c.topPerfomerReport(1);

      }

      // Builds a demo college, populated with some demo courses and students. */
      private static College buildCollege() {

            College c = new College("Berkeley School of Music");

            c.addCourse(1,"Arranging for Songwriters");
            c.addCourse(2,"Musical Theory");
            c.addCourse(3,"Basic Ear Training");
            c.addCourse(4,"Artist Management");
            c.addCourse(5,"Improvisation");

            c.addStudent(0,"Prince");
            c.addStudent(1,"Lady Gaga");
            c.addStudent(2,"Bob Dylan");
            c.addStudent(3,"Shakira");
            c.addStudent(4,"Paul McCartney");
            c.addStudent(5,"Shawn Mendes");

            c.addCourseTaken(1,1, 80);  c.addCourseTaken(1,2, 90);
            c.addCourseTaken(1,3,100);
            c.addCourseTaken(2,1,100);  c.addCourseTaken(2,2, 80);
            c.addCourseTaken(3,1, 70);  c.addCourseTaken(3,3, 70);

            return c;
      }
}
```

Resulting output:

```
Student 0: Prince
Student 1: Lady Gaga
Student 2: Bob Dylan
Student 3: Shakira
Student 4: Paul McCartney
Student 5: Shawn Mendes

Student 1: Lady Gaga
Student 2: Bob Dylan
Student 3: Shakira

Course 1: Arranging for Songwiters
Course 2: Musical Theory
Course 3: Basic Ear Training
Course 4: Artist Management
Course 5: Improvisation

Lady Gaga:
Course 1: Arranging for Songwriters, grade: 80
Course 2: Musical Theory, grade: 90
Course 3: Basic Ear Training, grade: 100
Grade average: 90

Bob Dylan:
Course 1: Arranging for Songwriters, grade: 100
Course 2: Musical Theory, grade: 80
Grade average: 90

Course report: Arranging for Songwriters
Student 1: Lady Gaga
Student 2: Bob Dylan
Student 3: Shakira

Course report: Musical Theory
Student 1: Lady Gaga
Student 2: Bob Dylan

Course 1: Arranging for Songwriters has 3 students

Top performer in Course 1: Arranging for Songwriters:
Bob Dylan
```