



CMSC131 - OBJECT-ORIENTED PROGRAMMING I

SECTION 010X, 030X, AND 040X

DR. ROGER EASTMAN AND DR. ILCHUL YOON

Project 5

Grades Analysis

Assigned: May 4th, 2018

Due: May 9th, 2018 11:00 PM

Late Due: May 10th, 2018 11:00 PM — 20% penalty

Good Faith Attempt Due: May 10th, 2018 11:00 PM

Drop Option: You can drop this project.

1 Overview

In this project you will be analyzing grade data for courses at the University of Maryland. Completing this project will require the use of many topics that you learned in the beginning of the semester, as well as some new topics that you have learned over the past couple weeks.

The grade data was obtained through a Public Information Act Request to the University by the website [PlanetTerp](#), and is from the Office of Institutional Research, Planning & Assessment. It was slightly modified for this project. The hope is that you will find this project interesting to write, as the data, and the analyses you will do, are directly related to you. Upon completion of the project, you will be able to look up grade information given certain constraints.

Use Eclipse to work on this project. To begin the project, check out the `GradesAnalysis` project from the CVS repository and then write code in the provided classes. You may add more methods if you think they will be helpful, but you are not allowed to modify the provided method signatures in the starter file.

2 Requirements

There are four classes you must implement. All of the instance variables you need to complete the project are included in the starter files. If you need any other instance variables, declare those as `private` variables. The requirements for each class are described below.

1. The `Professor` class represents a professor. It must have the following:

Instance variables:

- `String name`, representing a Professor's name

Constructors:

- A constructor that takes a `name` and sets the instance variable to it

Methods:

- `String getName()` that returns the `name` of the `Professor`
- `boolean equals(Object o)` that returns `true` if two `Professors` have the same name and `false` otherwise

2. The `Course` class represents a course. It must have the following:

Instance variables:

- `String department`, representing a Course's department
- `int courseNumber`, representing a Course's course number
- `char special`, representing a Course's "special" number. For example, for CMSC389A, the 'A' would be stored in `special`
- `String section`, representing a Course's section

Constructors:

- A constructor that takes a `department`, `courseNumber`, `special`, and `section`, and sets each instance variable to the respective parameter. See the instance variable description above for the types. If the `section` parameter has 3 characters, add a "0" before it. This is because in the grade data file, section "0101" is written as "101".

Methods:

- `String getDepartment()` that returns the Course's department
- `String getCourse()` that returns the full name of the Course. For example, a course with department "CMSC", courseNumber 131, and s '\0' (explained below) would return "CMSC131". Note that (1) there are no spaces between department, course and special and (2) the '\0' character should always be added to the return String, even if it is not displayed.
- `String getSection()` that returns the Course's section
- `boolean equals(Object o)` that returns `true` if two `Courses` have the same name (as defined in `getCourse()`) and `false` otherwise. Note that two `Courses` are equal even if they have different sections.

3. The `CourseGrade` class "connects" a `Professor` and a `Course` with grades. It must have the following:

Instance variables:

- `Professor professor`, representing the CourseGrade's professor
- `Course course`, representing the CourseGrade's course
- `Map<String, Integer> grades`, which maps a grade to the number of students who received that grade. The key is a `String`, rather than a `char`, because there are grades such as `A+` and `Other` (explained below) that are more than one character.

Constructors:

- A constructor that takes a `Professor` and `Course` and sets each instance variable to the respective parameter. It should also initialize `grades` to a new, empty `HashMap`.

Methods:

- `Professor getProfessor()` that returns the `CourseGrade`'s professor
- `Course getCourse()` that returns the `CourseGrade`'s course
- `Map<String, Integer> getGrades()` that returns the `CourseGrade`'s grades
- `void updateGrade(String grade, int numStudents)` that sets `grades` for the key `grade` to `numStudents`. It should overwrite any previously stored grade.
- `float getAverage()` that returns the average GPA for `CourseGrade`. A grade can be converted to a GPA using this table (found [here](#)):

A+	A	A-	B+	B	B-	C+	C	C-	D+	D	D-	F	W	Other
4.0	4.0	3.7	3.3	3.0	2.7	2.3	2.0	1.7	1.3	1.0	0.7	0.0	0.0	Ignore

Note that this does not follow the way UMD calculates grades exactly. For example, this project counts a W as a 0, whereas UMD does not include Ws in GPA calculations.

“Other” grades are grades which were not reported for some reason. They should be ignored.

For example, a course with 3 students who received an A-, 1 who received a C, 1 who received a W, and 1 who received an Other will have an average GPA of 2.62:

$$\frac{(3 * 3.7) + (1 * 2.0) + (1 * 0.0)}{(3 + 1 + 1)} = 2.62$$

We divide by 5, not 6, because we do not include the Other grade in the average calculation. Note that you do not need to round; just return what you calculated.

- `int getNumGrade(String grade)` that returns the number of students who received the given `grade`. Using the example above, calling `getNumGrade("A-")` should return 3. You can assume that before this method is called, `updateGrade` will have been called for each grade.
- `int getNumStudents()` that returns the total number of students who received a grade (including students who received a grade of “Other”).
- `int compareTo(CourseGrade cg)`. Notice that the `CourseGrade` class implements `Comparable`, an interface in Java (<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>). This interface gives classes a way to compare objects using the `compareTo` method.

This method should return 0 if the current `CourseGrade` object has the same average GPA as the `CourseGrade` object given as the parameter, 1 if the current `CourseGrade`'s average GPA is greater than the parameter's average GPA, and -1 otherwise. Read [this](#) to see how to check if floats are roughly equal. Use 0.000001 as the “epsilon”.

4. The `Analyze` class contains methods to analyze the grade data. It must have the following:

Instance variables:

- `List<CourseGrade> courseGrades`

Constructors:

- A default constructor that initializes `courseGrades` to a new, empty `ArrayList`

Methods:

- `void readData (String fileName)`

Note that this method has already been implemented. Do not modify it.

This method reads the passed grades file, which you can assume exists and is a **Comma-Separated Values (CSV)** file. This means the data is separated by commas. The first row is a header row that shows how the data is ordered. Here are two rows from the file (the header row, and row #1925):

Course,Section,Professor>Total,A+,A,A-,B+,B,B-,C+,C,C-,D+,D,D-,Fs,Withdraw,Other
CMSC250,401,Roger D. Eastman,33,1,1,4,6,5,2,1,6,3,0,0,1,1,2,0

You can match these rows up. CMSC250 is the Course, 401 is the Section, Roger D. Eastman is the Professor, a total of 33 students received a grade, 1 student received an A+, and so on.

Observe that the method reads through the file line-by-line and adds **CourseGrades** to **gradesList** after creating a **Course** and a **Professor** object for each row. It also sets the grades of a **CourseGrade** object from the grade data given in the row. Note that the number of students who received an “Other” grade is also added to a **CourseGrade** object.

If a course does not have a “special” character, the null character is assigned to **special**. This is done by setting, for example, `char special = '\0'` or `char special = 0`.

Once this method is completely executed, **courseGrades** will have all of the data from **fileName**.

- **float getProfessorAverage (Professor p)**, which returns the average GPA a professor gave. Note that you should not calculate the average with a simple mean between the courses a professor teaches. The average should be weighted based on how many students took a course a professor taught. For example, suppose a professor teaches two courses: course *A* with 20 students and course *B* with 10 students. The average GPA in course *A* is a 3.53 and in course *B* is a 3.08. The average GPA should be calculated as:

$$\frac{(3.53 * 20) + (3.08 * 10)}{20 + 10} = 3.38$$

Think about how you can calculate this using a for loop.

This method will return the average GPA the professor gave, which should be a **float**. If the given professor did not teach any courses, or only gave “Other” grades, return -1.

- **float getCourseAverage (Course course, boolean sectionOnly)**, which returns the average GPA of a course.

It is possible that multiple **CourseGrade** objects exist for a single course (i.e., multiple sections of a course — CMSC131 might have sections 0101 and 0102, and these will be separate **CourseGrade** objects). If **sectionOnly** is **false**, the average should be weighted based on how many students took the course across all sections, again ignoring grades of “Other”. In this case, the **section** of the given **course** should not be used. In other words, if **sectionOnly** is **false**, calculate the average GPA of a course across all sections.

If **sectionOnly** is **true**, you should compute the average only for the **section** of the **course** given as a parameter.

This method will return the average GPA, which should be a **float**. If no student took the course, or if the only grades are “Other”, return -1.

- **float getDepartmentAverage (String department)**, which returns the average GPA of the courses offered by the **department**. The average should be weighted based on how many students took the courses, as explained above, again ignoring grades of “Other”.

This method will return the average GPA of all courses in the given **department**, which should be a **float**. If no students received a grade in the department, or if the only grades are “Other”, return -1.

- **List<Professor> getProfessorsTeachingCourse (Course course)**, which returns a list of **Professors** who taught the given **course** at least once. This list should not have duplicates.
- **List<Professor> getProfessorsTeachingDepartment (String department)**, which returns a list of **Professors** who taught at least one course in the given **department**. This list should not have duplicates.

3 Notes

It may be confusing to understand what you are being asked to implement for this project. We recommend reading this description, looking at the data to see how it is formatted, and looking at the public tests to see what should be returned for various method calls, before beginning to write code. Calculating average GPA for certain courses before writing code may also be helpful.

In this project you are provided with Fall 2017 grade data. The file is named “201708_edited.csv”. Not every professor is included in the data, but every course should be. If a professor was not listed, they are listed as “UNKNOWN”.

The public tests contain a couple of examples of expected output. If you would like to see the correct output for other courses, you can look on PlanetTerp, the website that provided the data. For example, to see CMSC131’s average for Fall 2017, you may visit <https://planetterp.com/course/CMSC131?semester=201708>. The `?semester=201708` at the end of the URL limits the data to Fall 2017 grade data. Note that while this is also supported for professor pages, your output may not match that of a professor page, because PlanetTerp’s data is more specific for professors than the grade data provided. You can also use the grades page (<https://planetterp.com/grades>) to search by course, and limit by semester and section, to verify your results are correct. Note that due to differences in rounding, your results may be different from PlanetTerp’s by up to 0.01 GPA (but should not differ by more than that).

Finally, we encourage you to experiment with this project. Try making other methods that you think would be useful. There are a lot of things you can do with this data — have fun with it!

4 Grading and Good Faith Attempt Policy

Your submission will be evaluated based on the public, release, and secret tests in the submit server. All public tests must be passed to meet the GFA requirements for this project.

Tests grading breakdown:

<i>Public</i>	40%
<i>Release</i>	40%
<i>Secret</i>	20%
Total	100%

5 Requirements on readability and coding style

Follow a good coding style (e.g., <http://www.cs.umd.edu/~nelson/classes/resources/javastyleguide/>). Your code should be properly indented (use the AutoFormat feature in Eclipse), and well documented with appropriate comments. It is mandatory to use meaningful variable names, rather than “magic numbers” (literal constants). There are no explicit requirements for how many and what type of variables. For readability including variable naming, we suggest you to use these rules:

- Format the code clearly with blank lines, spacing and indents
- Use self-documenting, meaningful variable names
- Use variable names that begin with lower case and then camel case (e.g., `roomTemp`) if two words
- Use capital letters (e.g., `ALL_CAPS`) for variables that encodes constant values that aren’t updated
- Use appropriate comments that indicate what a section, or a complicated line, does

As you have done for other projects, the first few lines of your code should be comments with the project name, date, and author (you), as well as an explanation of what the project does.

You are **required** to write your name, directory ID, university ID, and section number as a comment at the top of each file you submit from this project forward. Additionally, we expect you to write the honor pledge (***I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination.***) as a comment near the top of each file you submit.

6 Submission

Submit your P5 to the submit server by the due date/time above. It is recommended that you submit directly using the 'Submit Project' option in Eclipse. After submission, check in the submit server that your work is submitted correctly and also that your code passed all tests. Note that you will need to pass all public, release, and secret tests to get full credit.

7 Academic Integrity

Make sure you read the academic integrity section of the syllabus so you understand what you must not do. Note that we check your submission against other students' submissions, and that we are required to report academic dishonesty cases to the University's Office of Student Conduct. As stated in section 5, you are expected to write the honor pledge as a comment near the top of each file you submit.