Java & OOSE 2

Laboratory Sheet 8

This Lab Sheet contains material primarily based on Lectures 15 – 16.

The deadline for Moodle submission is 17:00 on Thursday 17 November.

Aims and objectives

- Overriding equals() and hashCode() methods
- Making use of enum types
- Writing methods using the Java Streams API
- Using StringBuilder to write a toString() method

Set up

1. Download Laboratory8.zip from Moodle and launch Eclipse.
2. In Eclipse, select File → Import … to launch the import wizard – this will be used to import the starter projects into your workspace. Then select General → Existing Projects into Workspace and click Next.
3. Choose Select archive file and then browse to the location where you downloaded Laboratory8.zip. Select Laboratory8.zip and click Open.
4. You should now see a new project in the Projects window: Submission8_1. Ensure that the checkbox beside this project is selected, and then press Finish.
5. In the *Package Explorer view* (on the left of the Eclipse window), you should now see an icon representing the new project (in addition to any projects that were there before).

Please refer to the lab sheet from Laboratory 1 for additional details on how to open and run the code in the new project. Note that, as in previous labs, you will be creating some new classes as part of the assignment; refer to the lab sheet for Laboratory 4 for details on creating classes in Eclipse if necessary.

Submission material

Preparatory work for these programming exercises, prior to your scheduled lab session, is expected and essential to enable you to submit satisfactory solutions.

Unit Tests: I have supplied some simple test cases to check that your code is working properly, in the file TestStudentRecord and. **Note that the tests may not test all aspects of the program's behaviour – passing all of the tests does not mean that your code is perfect (although failing a test will definitely indicate a problem to be investigated).** You should also read through the specification and make sure that you have implemented everything correctly before submitting.

Submission

For this class, we return to the StudentRecord class that we used in labs 3 and 4. However, we will add an additional feature: the StudentRecord will now keep track of all courses that the student is currently taking or has taken in the past, and will provide methods to compute both an overall GPA and a GPA in the student's primary degree subject.

As a starting point, I have provided a StudentRecord class (in StudentRecord.java) that includes fields for the student name, ID, degree programme, and year of study, along with a constructor that initialises these four fields and getters that access them. Over the course of this assignment, you will first create two new classes, and then add fields and methods to the StudentRecord class making use of the new classes.

Representing a course specification

The first task is to represent a (simplified) course specification. You must create a class CourseSpec with the following fields (the bold word in each line represents the suggested field name):

- A string representing the **department** (e.g., "COMPSCI")
- A string representing the course **code** (e.g., "2003")
- A string representing the full course **title** (e.g., "Computing Science 2R: Algorithmic Foundations 2")
- An int representing the number of **credits** associated with the course (e.g., 10)

You should also create a constructor that initialises those four fields **in the above order** – that is, the fields in the constructor should represent the department, the code, the title, and the credits in order. **If you do not get the parameter order right in the constructor, you will encounter hard-to-track-down bugs in later parts of the assignment.**

You should also write getter methods for all of the above fields; you do not need to include any setters.

You should also override the equals() and hashCode() methods. For equals() and hashCode(), the only fields that you should consider are **department** and **code** (we are assuming that administrative changes may modify course titles and/or credit values, but that the codes like COMPSCI 2003 will remain unchanged).

Representing a course result

Once you have implemented CourseSpec, you should next implement a class CourseResult that represents a single attempt by a student to take a course. As part of this class, you should use the provided enum type Grade (in the file Grade.java) which represents the University of Glasgow 22-point scale as a Java enumeration. The CourseResult class should have the following fields (here, the field names do not matter as much, as they will not be used in the test cases directly):

- A CourseSpec giving the specification of the course that was taken
- A string representing the academic year when the course was taken (e.g., "2015-2016")
- A Grade representing the grade obtained by the student

For example, an instance of this class could represent the fact that a student took AF2 in 2015-2016 and obtained a grade of B2.

You should also implement a constructor and getter methods for all of the fields of this class; again, no setter methods are necessary.

Adding course information to the student record

Next, you must add fields to the StudentRecord class to represent the classes currently being taken by a student as well as the results of classes taken in the past, as well as methods that update and access those lists.

For the current courses, you should add a field storing a list of CourseSpecs (which should be initially empty), as well as a method

```
public void addCurrentCourse (CourseSpec course)
```

which adds the indicated course to the list of current courses. You should also add a method

```
public List<CourseSpec> getCurrentCourses()
```

which returns the list of courses. **Make sure that this method returns a copy of the list so that external classes cannot modify the list except through the addCurrentCourse() method** (use techniques similar to those used in the immutable Playlist class on the last lab).

Then, you should add a field storing a list of CourseResults (also initially empty), as well as a method

```
public void addCourseResult(CourseSpec spec, String year,
        Grade grade)
```

which creates a CourseResult from the given parameters and adds it to the list.

Finally, add a method with the following signature:

```
public List<CourseSpec> getPastCourses()
```

This method should return a list of the **distinct** courses taken by a student in the past – note, it is returning a list of CourseSpec, not of CourseResult. So if a student took AF2 two times, it should still only appear once in the resulting list.

For full marks, you must implement getPastCourses using the Stream API. The following stream methods will likely be useful for you in this method:

- map(), distinct(), and collect()

The documentation at https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html has details of all of these methods.

Computing a GPA

A student's GPA is the weighted average of their grades in all of their past courses. For simplicity, however, we will assume that all courses have the same number of credits – under this assumption, a student's GPA can be computed simply by taking the average of their grade in all of their past courses.

Note that, while grades are represented internally using instances of the Grade enum, you can access the number of grade points by calling the `.ordinal()` method. For example, if you have a variable `grade` with the value `Grade.A3`, then calling `grade.ordinal()` will return the integer 20.

Add a method to StudentRecord with the following signature:

```
public double computeGPA()
```

This method should compute and return the student's (simplified) GPA by averaging their grades across all past courses. **For full marks, you must implement computeGPA() using the Stream API**. The following stream methods will likely be useful for you here:

- `mapToDouble()` (from java.util.stream.Stream)
- `average()` (from java.util.stream.IntStream)

Writing StudentRecord.toString()

Finally, you should override StudentRecord.toString() to return a full description of the current student record, including all previous and current courses as well as the student's GPA. Here is a sample of what the output could look like; note that this is only an example, and you are free to design your own output format.

```
Student #1000: Firstname Lastname -- BSKT (Year 3)
Current courses:
- BSKT200: "Advanced Basket Weaving" (10 credits)

Previous courses:
- BSKT100: "Basket Weaving" (10 credits): G2 (2014-2015)
- ASTRO100: "Introductory Stargazing" (10 credits): D3 (2014-
2015)
- BSKT100: "Introduction to Basket Weaving" (10 credits): A2
(2015-2016)
- ASTRO100: "Introductory Stargazing" (10 credits): B2 (2015-
2016)

GPA: 11.75
```

For full credit, **you must use a StringBuilder to create the toString() result** – see the notes for lecture 14 for details on StringBuilder.

Testing your code

In addition to the JUnit tests in StudentRecordTest, you can also use the StudentRecordMain class's main method to do more testing; this method creates a set of course specifications, uses them to create a student record, and then prints out the results. The above sample output was printed from StudentRecordMain. At the end of StudentRecordMain, it also calls getPastCourses() and prints out the results.

Submission

You should submit your work before the deadline no matter whether the programs are fully working or not.

When you are ready to submit, go to the JOOSE2 moodle site. Click on `Laboratory 8 Submission`. Click '`Add Submission`'. Open Windows Explorer and browse to the folder that contains your Java source code and drag *only* the three Java files `CourseResult.java`, `CourseSpec.java`, and `StudentRecord.java` into the drag-n-drop area on the moodle submission page. **Your markers only want to read your java files, not your class files.** Then click the blue `save changes` button. Check the .java files are uploaded to the system. Then click `submit assignment` and fill in the non-plagiarism declaration. Your tutor will inspect your file and return feedback to you via moodle.

Outline Mark Scheme

Here is a rough mark scheme that your tutor will use when assessing your work.

A: Excellent attempt. Code compiles and runs. Code passes unit tests. Correct output for main method. Uses streams and StringBuilders where indicated.
B: Very good attempt. Code compiles and runs, although may not give correct answers in all cases. Functionality implemented correctly but not using the correct techniques (streams/StringBuilders).
C: Good attempt at solution. Code may not compile, but you have attempted to implement all specified behaviour.
D: Minimal attempt at solution. Code may not compile, but you have attempted at least some of the required behaviour.

Within these general guidelines, the tutors are also checking to see whether:

- you use sensible variable names and method names
- you write appropriate comments (Javadoc or similar) for classes and 'interesting' methods
- you write clean and efficient control flow structures (e.g. for each loops) in your code.
- you include an `@author` tag in each source code file
- you are using packages correctly

Possible extension

The above is all that we expect you to do on this assignment, and is all that you are required to submit. This section describes possible extensions to this work that should be attempted only by those who are interested – this work is not required, and will not be assessed.

For the GPA calculation above, we made the simplifying assumption that all courses have the same number of credits. However, as you probably know, computing a real GPA involves taking a weighted average – that is, instead of taking the simple average of all grades, you should instead compute

$$\frac{\sum(credits(c_i) * grade(c_i))}{\sum credits(c_i)}$$

Where *credits(c_i)* is the number of credits for a given course (e.g., 10 for AF2, 20 for JOOSE2), and *grade(c_i)* is the grade in the course (A1=22, etc.).

Computing a weighted average like this is more complicated with streams – here is a recent StackOverflow thread that gives an idea of a starting point.

http://stackoverflow.com/questions/40420069/calculate-weighted-average-with-java-8-streams

See if you can generalise the GPA-computing method in StudentRecord to return a weighted average. I have provided an additional test case to test your implementation; just un-comment the @Ignore annotation if you want to use it.