CSC/CPE 203

Lab 1- Java Collections, Control Structure, Classes, Testing, and UML

Due: 10/06

This lab is adapted from this assignment in Evan Peck's list of Ethical CS modules for CS 1.

Objectives

- To be able to iterate through different Java collections with various control loops (part 1)
- To be able to write your own class with specified methods (part 2).
- To be able to use unit testing, including writing your own tests for your code (part 1 & 2)
- To be able to write a very simple UML diagram of a class using yEd (part 3)

Resources

- Your peers
- Your instructor
- The Java Standard Library (Java 18) — Bookmark this web page, you will be using the Java Standard Library documentation regularly this quarter.

Setup

You can use IntelliJ IDEA to write and run your code.

Important: For this lab, we will be using JUnit to help us test the correctness of our code. You can either let IntelliJ add it to your project automatically, or you can do it manually yourself. To do it manually, you'll need to download the JUnit library (available in the Modules page) and add it to your project in IntelliJ IDEA. Instructions for how to do it either way can be found **here**.

Aside: What's JUnit? JUnit lets us make assertions about what our code *should* do. It is similar to Unit test in Python. You could think of them as statements of *expectations*—for example, if we wrote a program to do addition, then we can *expect* that `2 + 2 = 4` and that `-1 + 3 = 2`. As a project grows over time, these assertions ("tests") grow along with it, giving us a "safety net" of checks that can assure us that our program is doing what we expect.

Retrieve the files provided for this lab from Canvas. This lab allows you to practice Java syntax related to collections and control (Part 1), basic class structure and methods (Part 2), and

includes an introduction to UML (Part 3). Those already familiar with Java should finish the lab and then offer to help others.

Algorithms as Future-Makers

Imagine you are working for *Moogle*, a well-known tech company that receives tens of thousands of job applications from graduating seniors every year. Since the company receives too many job applications for the Human Resources Department (HR) to individually assess in a reasonable amount of time, you are asked to create a program that algorithmically analyzes applications and selects the ones most worth passing onto HR. Your job is to build an algorithm that helps determine the order which job applicants will be called in for an interview. This is a real scenario playing out every day and will likely impact you as an applicant someday.

To make it easier for their algorithms to process, *Moogle* designs their application forms to get some numerical data about their applicants' education. Applicants must enter the grades they receive from their 5 core CS courses, as well as their overall non-CS GPA at the university. For your convenience, this will be stored in a Java array or list that you can access. For example, a student who received the following scores…

- **Intro to CS**: 100
- **Data Structures**: 95
- **Algorithms**: 89
- **Computer Organization**: 91
- **Operating Systems**: 75
- **Non-CS GPA**: 83

…would result in the following list:

```
[100, 95, 89, 91, 75, 83]
```

That is, you can assume that index 0 is *always* **Intro to CS**, 1 is *always* **Data Structures**, and so on.

Part 1: Java Control Constructs and Introduction to Collections

In the provided `part1` subdirectory, edit each of the given files to complete the implementations and tests. Comments in each file will direct the edits that you are expected to make. Look up any classes you don't know how to use (such as Map) in the Java Standard Library.

It is recommended that you use the order of the unit tests in `TestCases.java` as a guide for the order in which to complete the methods. This order corresponds to the increasing complexity of the methods. A good process to follow would be to read the first group of tests in `TestCases.java`.

Edit the corresponding class to make the code pass the tests. *Run* the tests to make sure they now pass. Add the required additional test case. *Run* the tests again. Then move on to the next group.

The output for a failed test is incredibly verbose (a stack trace is printed). You can see the specific error at the numbered line above each stack trace. *As you fix the code one file at a time, re-run the code to make sure that the number of errors is decreasing!*

(For Map: you are mapping applicant name with their course grade. You are creating list of applicants that all their course grade > threshold!)

Part 2: Classes and Objects

For this part, you will implement a simple class (named Applicant) representing a job applicant and their resume. Do this by creating a file named Applicant.java and put in all the code specified below. This class must support the following operations (including a single constructor).

- public Applicant(String name, List<CourseGrade> grades) — Constructor.
- public String getName() — Returns the applicant's name.
- public List<CourseGrade> getGrades() — Returns the applicant's list of scores.
- public CourseGrade getGradeFor(String course) — Returns the score that was asked for, which is stored in grades list. The course parameter can be one of the courses as defined here. That is, getGradeFor("Intro to CS") will return the CourseGrade at index 0, getGradeFor("DataStructures") will return the CourseGrade at index 1, and so on.

Once this class is created, look at the TestCases.java file in the part2 directory—complete the test cases or add new ones as appropriate.

Creating your own filter

You've now written and seen the effect of 3 different analyses. It's time to create your own.

You will decide what you think is the most fair way to filter applications. **You shouldn't just use what we've created so far. Come up with something better.**

So far, our Applicant only has a name and course grades. Do you want to take more information into account while deciding whether or not to accept the applicant? **Add the appropriate fields to the Applicant class.**

Then, in the SimpleIf class, create a new function: * public static boolean analyzeApplicant2(Applicant applicant)

And implement your new filtering mechanism. In a JavaDoc comment for analyzeApplicant2, *explain in plain English* your rationale for this new applicant filter.

Finally, write tests in part2/TestCases.java to ensure that your new filter is working correctly.

Part 3: UML

See the given pdf description of the UML portion of the lab and create UML diagram for Applicant class of part 2.

Demo

Show your finished UML diagram from Part 3 to your instructor *in lab* on the day it is due. After demo, your Parts 1 and 2 will be tested with extra testcases via submission to Canvas.

Submission

Submit your code by due date for Parts 1 and 2 in the Canvas. The extra tests will be applied to your submitted code. Code submitted after 10PM on the due date will not be graded.

Rubric:	**Points**
1) UML diagram	5
2) Test P1 (19 tests)	19
3) Test P2 (9 tests)	6 (2 tests is given add 4 more tests)
4) Extra filter	5
5) Extra test cases P1	20 (Using this test I will check the correction of your code.)
6) Extra test cases P2	15 (Using this test I will check the correction of your code.)
7) Submission	30