Week 3 Lab

Constructors and mutators

Objectives

In this week's lab, you will:

- refactor code to improve its design and readability.
- use the built-in `ArrayList` container class
- use a generic class
- use a `BufferedReader` for console I/O.
- implement a constructor
- implement setter methods

Task 1. Preparation: UML Class diagram

You must draw a UML class diagram showing your planned system at the end of this lab after implementing all tasks.

Again, you are strongly encouraged to draw this diagram by hand and scan it in (or take a photo of it) for submission. You don't have to show attributes or methods, only classes and their associations. No additional marks will be granted for using a UML diagramming tool.

This must be submitted via the link on the subject Moodle page *24 hours before the scheduled commencement of your lab class* to receive any marks.

Task 2. Refactor `University` to use `ArrayList`

Make a copy of the JavaUniversity project you created last week. Make all changes to the copy.

Native Java arrays have some significant limitations - not least, they are *statically allocated*. That means that their maximum size is fixed at the time of their creation – you can't easily resize an array. We would like to change `University` to make it easier to add new `Units`. Fortunately, the Java class libraries provide a wide variety of *collection* classes in the Java Collections Framework. We will use one called the `ArrayList` class.

Modify the University class to use an `ArrayList` rather than an array to store the collection of `Units`.

Task 3. Refactor `Unit`'s method names using Eclipse refactoring tools

In Week 2, you created a `Unit` class with a method called `getUnitDescription()` that returns a `String` consisting of the unit code and the unit name. This is not really a good name. By convention in Java, methods starting with `get` return the value of an attribute, and that's not

what this method does. Also, since it is a part of the `Unit` class, we really don't need the "Unit" part of the method name. We can just call it `description()`.

Making this sort of change using a simple text editor can be a lot of work. We need to change the method name not only in the `Unit` class, but also in every client of `Unit` that uses that method. We dont want to change any method that has the same name, but belongs to a different class. Fortunately, Most modern IDEs, such as Eclipse, provide refactoring tools to help.

Go to the Unit class and find the method `getUnitDescription()` . Select the text `getUnitDescription()` and then right-click on it to bring up a menu. Choose `Refactor` → `Rename...`. Edit the method name to be just `description()`. The IDE will analyse the code for the system and change the method name everywhere it occurs. Now open the `University` class and check that the method name has changed where it is used in the method `displayUnits()`.

If you are using an IDE other than Eclipse, work out how to use the refactoring tools in your IDE to make the same changes.

Once you have made the changes, informally test your program to ensure that the functionality is unchanged. Refactoring should not change *what* your program does, only *how* it does it. In a larger project, you would use automated *unit tests* and possibly *system tests* to ensure that your refactoring has not affected any functionality.

Task 4. Create a `Student` class

A university is nothing without students. We need to add a `Student` class to our system. You will need to use some of the new Java language features we have seen in lectures to implement `Student`, including private attributes and constructors.

The `Student` class needs to have private attributes to store:

- the student's ID (`studentId`)
- the student's given name (`givenName`)
- family name (`familyName`)

Note that we have not specified the type of `studentId`. You can choose to use a numeric integer type, or a Java string [1]. This kind of decision is actually a design decision. In a comment, briefly list the advantages and disadvantages of using both, and justify your final decision. "I don't know how to use Java integers very well" is not in itself sufficient justification.

Additionally, add a comment explaining why you should use "givenName" and "familyName" rather than "firstName" and "lastName".

Implement two constructors for `Student`, with these signatures:

- `Student(*type* newStudentId)`
- `Student(*type* newStudentId, String newGivenName, String newFamilyName)`

Replace `*type*` with the name of the type you have used as the attribute.

These features provide two different ways of creating and initialising a Student object.

Next, implement two mutators (methods that change attribute values):

- `setGivenName(...)`
- `setFamilyName(...)`

[1] actually, there are several other possible representations, but you need only consider these two

These should both take String arguments.

Finally, implement an accessor method called `description()` that returns a `String` containing the Student ID, given name, and family name concatenated together, in that order and separated by spaces.

Task 5. Enrolling Students in Units

Students need to be able to enrol in units. Extend the `Unit` class to support this.

First add either an array, or an `ArrayList`, attribute called `students` to `Unit` to store a collection of `Student` objects. Again, add a comment to justify your choice.

Add a method `enrolStudent(Student newStudent)` to `Unit` to enrol students in a unit.

n the University class, modify `createUnits()` so that it also creates some `Student` objects and enrols them in some units. Make sure you test both `Student` constructors and the mutators. Modify `displayUnits()` so that, as well as displaying the description for each unit, it also displays a list of the students enrolled in each unit.

You will need to decide how to access the enrolled students in `Unit` from the `University` class. There should be no I/O in any class other than `University`, and the mechanism should not allow the students array stored in `Unit` to be modified by any other class.

The output should look like this:

```
Welcome to Java University.

FIT1234 Advanced Bogosorts
Enrolled Students:
12345678 Victor Chang
12345679 Fred Nurke

FIT2027 Introduction to Spaghetti Coding
Enrolled Students:
12345680 Indira Naidu

FIT3456 Enterprise Fizzbuzz
Enrolled Students:
12345680 Indira Naidu
Thanks for using Java University.
```

Test your program, and show it to your demonstrator, before proceeding with the next task.

Task 6. Add console I/O

Many programs that you will write, including the assignments in this subject, will have a user interface of some kind, to allow humans to interact directly with them. These days, most such programs use some kind of graphical user interface, which is often web-based.

However, in certain circumstances a *console-based* user interface are used. You'll see a console-based user interface if you open the command prompt (or PowerShell) in Windows) or a Terminal on a Mac or a Linux machine.

A console user interface is a bit like having a two-way conversation with an old typewriter (which in fact how they originally worked). You'd type something at the keyboard, hit enter, and then the computer would respond with some text output of its own.

Console-based user interfaces are usually less attractive and harder to learn, but they:

- can be as fast or faster to use than a GUI for experienced users,
- can be used over a slow or heavily loaded network connection,
- can be easily *scripted* with another computer program taking the place of the human typing the input, and importantly,
- are much easier to write than a GUI or web interface!

There are several ways to implement a console user interface in Java. Unfortunately, the simplest method, is the `Console` class[2], is incompatible with the Eclipse console where the output displays when you run from within the IDE.

You have seen how to use `println` to produce console output. Below is a Java method you can use to read console input. The argument, `prompt`, is displayed on the line to the left of where you type any input. If you don't want a prompt, pass in an empty string or the `null` reference.

```
private String readString(String prompt) {
    System.out.print(prompt);
    BufferedReader in = new BufferedReader(new InputStreamReader(
        System.in));
    String s = null;
    try {
        s = in.readLine();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return s;
}
```

Modify `University` so that when each `Student` object is created, the user is prompted to enter the students first name, last name, and student ID. Read these data from the command line and use them when creating the `Student` object.

Getting marked

Once you have completed this task, show your lab demonstrator to be marked. If you do not get this done by the end of this week's lab, you may demonstrate it at the beginning of next week's – but make sure that your demonstrator has at least given you feedback on your design before you leave, so that you don't waste time implementing a poor design.

Note that to receive full marks, it is not sufficient merely to produce correct output. Your solution must implement the structure described above.

Task 7. Design a student list printer

Once you've finished implementing the current functionality, let's consider how the current design affects how easy it is to extend the system.

Write down a description of a method to print a list of all students, without any duplicates. Your approach cannot rely on removing any existing code, changing any classes, or changing attributes in the existing system.

You may use any combination of pseudocode, plain English text, and UML sequence diagrams to describe your method. You do not have to implement your method; you need enough detail that you could give it to a classmate and they could implement it without having to consider anything but the details of Java.

[2] `https://docs.oracle.com/javase/tutorial/essential/io/cl.html`

Take your time; you may submit this with *next week's lab preparation*. Feel free to discuss your ideas with your demonstrator, however.

Marking
This lab is worth 2% of your final mark for FIT2099. Marks will be granted as follows:

- 0 marks if you do not complete all tasks, or if your demonstrator has not seen and approved your design
- 1 mark if all tasks are completed but the design is poor or the implementation is flawed
- 2 marks if all tasks are complete, the design is good, and the implementation is correct