COIT20245
Assignment 2

1. Introduction

In this assignment, you are to implement a console application that supports simple BMI (Body Mass Index) analysis functionality through a class called BMIAnalyser. A phased implementation approach is recommended; refer to Appendix 2 for further details.

2. The Application

The data to be analysed is to be loaded from 3 arrays containing person identifiers, heights (in metres) and weight (in kgs) using the loadFromTables() method provided in Appendix 1. Loading involves the creation of objects of type Record from each data triple (id[i], height[i], weight[i]). These are to be stored in an array of type Record called data. The size of this array is to be the same size as the data arrays. During loading, the BMI for each individual is calculated, using the following formula:

BMI = weight / (height*height)

where weight is in kilograms and height is in metres. The corresponding category is then determined as follows:

Category	BMI (kg/m2)	
	from	**to**
Very severely underweight		15
Severely underweight	15	16
Underweight	16	18.5
Normal (healthy weight)	18.5	25
Overweight	25	30
Obese Class I (Moderately obese)	30	35
Obese Class II (Severely obese)	35	40
Obese Class III (Very severely obese)	40	

This allocation is to be provided as a separate (private) method called classify() that is not provided in Appendix 1. The code for loading is provided in Appendix 1 as the method loadFromTables().

In order to facilitate search, data is to be maintained in ascending order of person id. Sorting is to be done after loading, using a method called sortById(). This method must implement the insertion sort algorithm. Note that loadFromTables() and sortById() are to be invoked from the constructor for the BMIAnalyser class

The application's View class is to execute (using a switch statement) the following command options:

1. Display the record for a specified person
2. Display records for all persons with BMI values within a specified range.
3. Display the minimum, maximum and average BMIs
4. Exit the application

As it is a console application, the user will need to be prompted to enter a command and its arguments (if any). My personal preference is for a minimal interaction scheme, as shown below:

```
run:
The following commands are recognised
       Display this message                                    > 0
       Display a specific subject record:                      > 1 subjectID
       Display records for all subject records within a range > 2 bmi1 bmi2
       Display statistics (minimum, maximum and average marks)> 3
       Exit the application                                    > 9
> 1 S01
<S01,1.8,80.0,24.7, Normal (healthy weight)>
>
```

Note that

1. Each command is designated a number
2. Command 1 requires a single argument – the subject id
3. Command 2 requires 2 arguments – the lower and upper bounds of a range of BMI values
4. I have added a "help" command (command 0)
5. The command options are displayed at the start of the application and whenever a "help" command is entered, rather than than after each command.
6. Records are displayed with no explanation of the fields

Feel free to adopt the above scheme or if you prefer, implement a more verbose interaction scheme.

For the commands that require arguments (i.e. command 1 and command 2), note that

a. For commands 1 and 2, basic error checking is expected. For command 1, an error message is to be displayed if an id does not exist. For command 2, an error message is to be displayed if a member of the range is < 0 or > 100 or if the second member of a range is less than the first member.
b. For command 2, the range is exclusive of the values specified. Consequently, having both members of a range equal is to be flagged as an error.
c. For command 1, binary search is to be used.
d. For command 2, the results are to be stored in an ArrayList for display

The application **must** conform to the class diagram of Figure 1, although additional **private** members and methods are permitted.

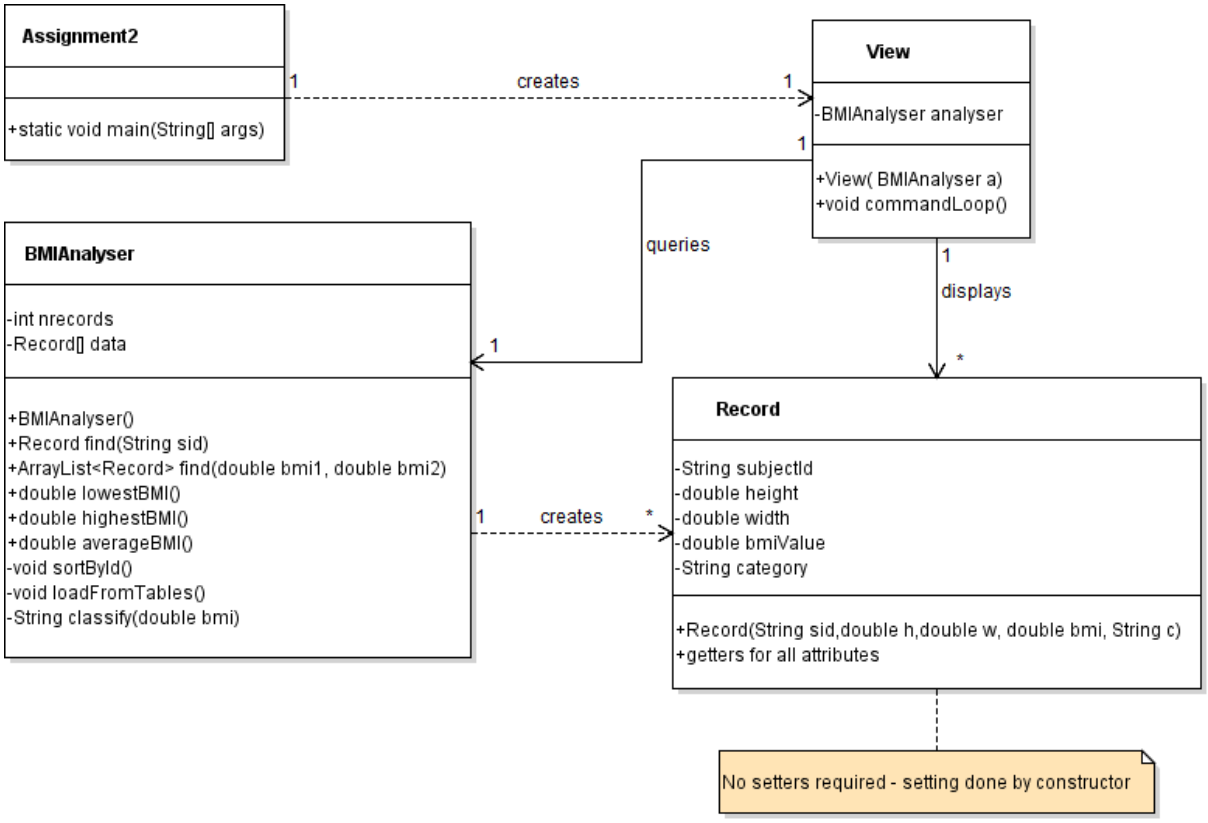

Figure 1. Class Diagram

Note that in Figure 1, the visibility of class methods and attributes are designated as + (public) and – (private). Associations are annotated with a label (for readability) and multiplicity. The multiplicity captures the number of object instances that can be involved in the association. In this case, only two values apply – 1 and *, which means many.

3. Submission

You are to submit a zipped folder containing

- The four .java files that comprise your application. Alternatively, if you have used NetBeans, this can be replaced with a zipped project folder. Details of how to do this are provided in the NetBeans FAQ on the unit website.
- Report.docx. This file contains a test report that includes student name, student ID number, unit name, unit code, a test plan and test results. The test plan is to contain a list of the input values that you have used to test the program, the expected output values and the actual output values generated by your program. The test results are screenshots to show that that the program generates the actual outputs shown in the test plan.

Note that because BMI is calculated as a floating point value, we can't (in general) test it for equality. What this means in terms of your test plan is that you do **not** need to test the category end points – all you need to do is to generate one set of inputs (and the corresponding output) per category. In generating these inputs, you may find a BMI calculator, such as https://www.heartfoundation.org.au/your-heart/know-your-risks/healthy-weight/bmi-calculator

useful. Note that markers will use their own test data when testing your assignment.

4. Marking Criteria

	Criteria	**Marks Allocated**
1	**Functionality: Assignment2 class (2 marks)**	
	Construction of objects as per spec	/2
2	**Functionality: View class (3 marks)**	
	Command loop implementation	/2
	Constructor implementation	/1
3	**Functionality: Record class (2 marks)**	
	Constructor, getters implementation	/2
4	**Functionality: BMIAnalyser class (13 marks)**	
	Find record by id (0 if binary search not used)	/3
	Find records by range of BMI	/3
	Find lowest BMI	/1
	Find highest BMI	/1
	Find average BMI	/1
	Sort records (0 if insertion sort not used)	/3
	Calculate category	/1
5	**Coding Style (5 marks)**	
	Program logic	/1
	Spacing and indentation conventions	/1
	Naming conventions	/1
	Comments	/2
6	**Report (5 marks)**	
	Test plan	/3
	Test results	/2

	Sub-Total	/30
	Penalties	
	Does not compile/run: 25-30 marks	
	Late submission : 5% (1.5 marks) / day or part of a day	
	Total	/30

Appendix 1

The code for the `BMIAnalyser. loadFromTables()` method (which is to be called from the BMI Analyser constructor) is as follows:

```
private void loadFromTables() {
    String[] subjects = {
        // your values go here
    };
    double[] height = {
        // your values go here
    };
    double[] weight = {
        // your values go here
    };

    nrecords = subjects.length;
    data = new Record[nrecords];
    for (int i = 0; i < nrecords; i++ ) {
        double v = weight[i]/(height[i]*height[i]);
        String c = classify( v );
        Record r = new Record(
              subjects[i], height[i], weight[i], v, c );
        data[i] = r;
    }
}
```

Appendix 2

It is good practice to develop an application incrementally. The idea is that the work is broken up into phases, with each phase producing a working system. Furthermore, if a phase involves the provision of multiple functionalities, as in Phase 3 below, these should also be tackled incrementally. The advantage of this approach is that you have a series of working systems that you can demonstrate to the customer. In your case, it means that you can always submit a working system, albeit one perhaps with limited functionality. For this to work properly, you need to save copies of your working versions as you go. This way of doing things is such a good idea (especially for team development) that software support is readily available in the form of version control systems such as SVN, GIT, CVS etc. However, we don't introduce you to version control until later in your course.

In terms of how you might phase this assignment, below is how I went about it, given that I was working from a class diagram. Arguably, it would be better to start with Phase 2 and have an initial application that produces output. I didn't do this as Phase 1 is straightforward and much of it is needed for Phase 2 anyway.

Phase 1.

1. Create all classes with members and dummy methods as per the class diagram. Where constructors set member variables via parameters, add code to do this. Leave void methods with an empty body. Where methods require a return value, just return a sensible type-consistent default value, as shown below for the BMIAnalyser public methods:

```
public Record find( String sid ) {
    return new Record ( sid, 0, 0, 0, "none" );
}

public double lowestBMI() {
    return 0.0;
}

public double highestBMI() {
    return 0.0;
}

public double averageBMI() {
    return 0.0;
}

public ArrayList<Record> find( double bmi1,double bmi2 ) {
    ArrayList<Record> alr = new ArrayList<>();
    alr.add( new Record ( "S01", 0, 0, 0, "none" ) );
     alr.add( new Record ( "S02", 0, 0, 0, "none" ) );
    return alr;
}
```

We could have returned null for the methods that return object references, but I chose to return something slightly more interesting. Bear in mind that if find(String sid) is unsuccessful, returning null is a good way to indicate failure and you will ultimately need to test for this. In the case of find(double bmi1, double bmi2), there are two sensible choices –

either return null or return an empty list on failure. Also note that you will have to include any import statements that are needed.

2. In the main() method, add the following code:

```
BMIAnalyser a = new BMIAnalyser();
View v = new View( a );
v.commandLoop();
```

Do not proceed to Phase 2 until this code compiles. You now have an application that compiles but when it runs, no output will be produced.

Phase 2.

Get the command loop working with the dummy methods. You will need to provide real getter methods for the Record class. You might find it helpful to write some private helper methods, e.g.

```
private String help() {
    return
        "The following commands are recognised\n"+
        "\tDisplay this message                                    > 0\n"+
        "\tDisplay a specific subject record:                      > 1 subjectID\n"+
        "\tDisplay records for all subject records within a range > 2 bmi1 bmi2\n"+
        "\tDisplay statistics (minimum, maximum and average marks)> 3\n"+
        "\tExit the application                                    > 9" ;
}
```

and perhaps display methods for commands 1,2 and 3.

When I did this with our dummy testing methods, the following output was generated:

```
run:
The following commands are recognised
       Display this message                                    > 0
       Display a specific subject record:                      > 1 subjectID
       Display records for all subject records within a range > 2 bmi1 bmi2
       Display statistics (minimum, maximum and average marks)> 3
       Exit the application                                    > 9
> 1 xxxx
<S01,0.0,0.0,0.0,none>
> 2 0 100
<S01,0.0,0.0,0.0,none>
<S02,0.0,0.0,0.0,none>
> 3
Lowest  BMI is 0.0
Highest BMI is 0.0
Average BMI is 0.0
> 9
BUILD SUCCESSFUL (total time: 58 seconds)
```

Note I have displayed records very tersely; you will need to at least add member names. Also, the interaction is minimalist. You may prefer to display the menu before every command and explicitly prompt for all inputs – either style is fine.

Phase 3

We now need to

1. populate the bmi analyser with data using the code provided

2. change the relevant bmi analyser methods to use this data rather than just returning dummy values as in Phase 2.

Note that

1. we can do the changes one at a time, retaining dummy methods for the commands that we are yet to work on.
2. generating statistics will be the same regardless of whether the records are in sorted order (they are to be sorted according to subject id) so that is a good (and easy) place to start.

Phase 4

Test the application. Make sure that error checking (as requested) works. Also change the data to introduce edge cases. E.g. what will happen with 0 records? One record? Etc.