COMP0004 Object-Oriented Programming
Programming Exercises 3

Aim to complete as many questions as you can by the end of Reading Week

Purpose: Writing Java classes.

Goal: Complete as many of the exercise questions as you can. If you are keeping up, you need to do at least the core questions. The additional questions are more challenging and are designed to stretch the more confident programmers. If you can't do them now, be prepared to come back and try them later on. You must complete the core questions and get them reviewed in a lab session.

Feedback: It is important that you get feedback on your exercise answers so that you know they are correct, that you are not making common mistakes, that the program code is properly presented and that you are confident you have solved the problem properly. To do this, get your answers reviewed by a TA during a lab session.

See the additional notes on the COMP0004 Moodle site about using Java, and classes Input, FileInput and FileOutput.

Q1.

a) Write a `class AddressBookEntry` to represent an entry in an address book. An object of the class should store the name, phone number and email address for one person.

The class should have a constructor to initialise a new object to represent a specific person, and getter methods to access each piece of information:

```
public AddressBookEntry(String name, String phone, String email);
public String getName();
public String getPhone();
public String getEmail();
```

The class must not do any input or output. The information needed to represent a person entry is passed via the constructor. The instance variables storing the data must be private.

b) Next write a `class AddressBook`, that uses a
`private ArrayList<AddressBookEntry>`
instance variable to store entries in an address book. The class should include methods to add, remove and search for entries. Note again, this class should not do any input or output.

c) Write a class with a main method that creates an address book and allows it to be used by adding, removing and searching for entries. This class does the input and output, as well as the main method.

Think carefully about the public methods and their parameters needed by all three classes. What is are the correct methods for the abstractions that the classes represent? All instance variables should be private, so that data is accessed only by the object it belongs to.

Q2. A Range represents a sequence, or range, of integers, for example 1 to 10, 3 to 15, or -6 to 3. Write a `class Range` in Java with the following methods:

- a constructor
  `public Range(int lower, int upper)`
  The range includes all the integers from the lower limit to the upper limit inclusive. For example, the range 2 to 4 includes 2, 3 and 4. The lower limit must be lower than the upper limit otherwise the constructor will throw an exception.
- A method `int getLower()` to return the lower limit.
- A method `int getUpper()` to return the upper limit.
- A method `boolean contains(int n)` that returns true if the parameter value n is in the range, false otherwise.
- A method getValues() that returns an `ArrayList<Integer>` containing all the values in the range in order.

Write a second class, with a main method, to use `Range` objects and confirm they work correctly. Note that your Range class should not do any input or output, or have a main method.

Q3. Consider a table that shows temperature conversions from Celsius to Fahrenheit, like this:

```
                Temperature Conversion
                ----------------------
C   F       C   F       C   F       C   F       C   F
0   32      1   33      2   35      3   37      4   39
5   41      6   42      7   44      8   46      9   48
10  50      11  51      12  53      13  55      14  57
```

a) Write a *class* for generating temperature conversion tables for specified ranges of temperatures. An object of the class is created to represent a particular table, and then its methods called to generate the table contents. The object should store the generated table in a 2D `ArrayList<String>` data structure. Think of the class as a factory for creating temperature conversion table objects.

The class should have the following methods:

- a constructor that allows the Celsius temperature range to be specified (min and max temperature to be shown in the table).
- a public method to set the centred table header, e.g., the label “Temperature Conversion”
- a public method to specify the column headers, e.g., ‘C’ and ‘F’.
- a public method to specify the number of columns.
- a public method to specify the number of spaces between each column.
- a private method to convert from Celsius to Fahrenheit (F=C * 9/5 -32).
- a public method to generate and format the table in the 2D `ArrayList<String>` representation. This ArrayList should be *private* to the class.
- a public method to return a *copy* of the 2D `ArrayList<String>` representation of the table.

Note the the object *does not do any input or output*. It does not print the table on the screen, it only generates the ArrayList representation.

Write a second class that uses the table class to generate tables and display them. This class does the input and output, so that the user can specify the table to generate and then see it displayed on the screen. The class also has the main method.

Hint: Wouldn’t it be convenient if you could use something like printf to help format the strings making up each line of the table? Maybe if you look at the JavaDoc for class String?

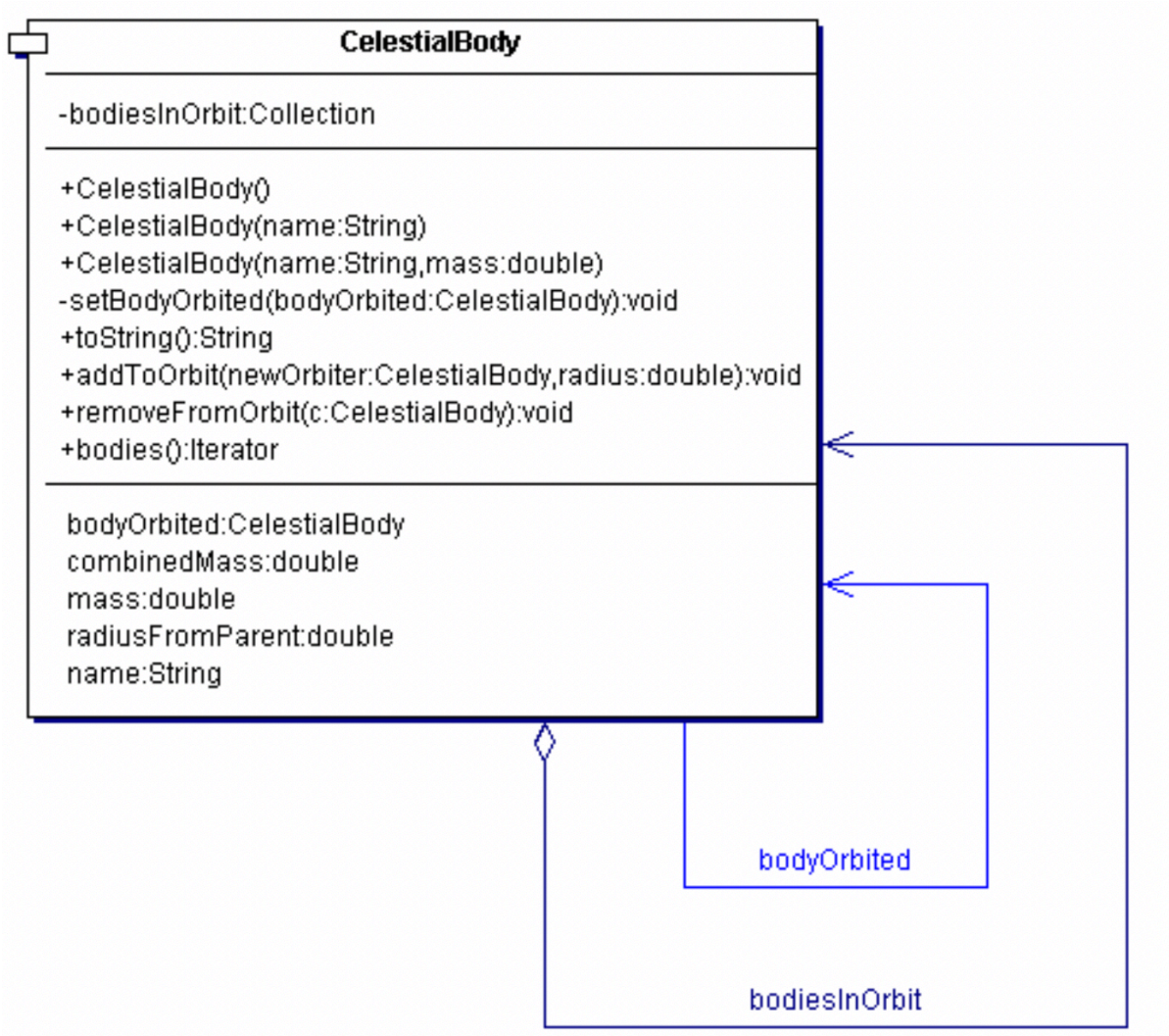
Strategy: The table class is quite complex, so don’t attempt to write the complete version all in one go. Start with a basic version that maybe just prints out the Celsius and Fahrenheit values without

any formatting. Then incrementally add one small feature after another moving forward step-by-step to the final version.

b) Add additional features to your temperature conversion table class to allow different ways to display the table and more formatting options.

Challenge Questions

Q4. Look at this UML class icon:



A CelestialBody is an object like a planet, star, moon. Objects of this class could be used to construct a model (data representation) of the solar system, using the associations shown outside the icon on the right. The icon shows various attributes (instance variables) and methods in UML style (looks like Java but not the same).

a) Write a Java version of the class CelestialBody. Make your own assumptions about the method implementations. You can add or remove attributes and methods as you see fit, but make changes when you *know* you need them, not just to make lots of changes for things that might (or might not) be needed!

b) Write the code (more classes) to implement a program to use your CelestialBody class. You will probably want to work on a) and b) in parallel. The program might read data from files to create the objects to represent the solar system, for example, and then display a representation of the solar system. If you are ambitious you might want to attempt a graphical display, otherwise using text.

Q5.

The Game of Greed

Greed is a dice game played between two or more players. The object of the game is to tally points from the dice rolls and be the first player to score 5,000 points. There are five dice in the game that are rolled from a cup.

To enter the game, a player must score at least 300 points on the first roll of his or her turn; otherwise, the player is considered “bust”. If the player goes bust, he or she must wait their turn to roll again.

If this first roll produces 300 or more points, then the player has the option of stopping and keeping the initial score or continuing. To continue, the player rolls only the dice that have not yet scored in this turn.

A player may continue rolling until all the dice have scored or until they go bust. With the exception of the entry roll, a bust is when an individual roll produces no points. The player may stop and keep their score after any roll as long as they have not gone bust.

If all the dice score on a single role, then the player can continue and roll all the dice again.

When a player goes bust or opts to stop, the dice are passed to the next player for their turn.

Each player keeps the running total of all the scores from their turns.

The score for each dice roll is determined as follows:

- Three of a kind (three dice all showing the same value) scores 100 times the face value (e.g, if three dice are showing 3, then 100 x 3).
    - But if the face value is one, then it is scored as 1,000 (e.g., there are three dice showing 1).
- Single dice showing 1 or 5, score 100 and 50 points respectively.

Finally, the winner is determined after a player collects a total score of 5,000 or more and all players have had an equal number of turns. If, for example, a player scores over 5,000 points, they may still lose if a subsequent player ends up with a greater final score.

Examples (for first roll):

i) Rolled 44446 = 400 points (three of a kind showing 4)

with the option to roll the 4 & 6 again.

ii) Rolled 11111 = 1000 + 100 + 100 = 1200 points (three of a kind, plus two 1s)

with the option to role all five dice again, as all dice scored.

iii) Rolled 12315 = 100 + 0 + 0 + 100 + 50 points

with the option to roll the 2 & 3 again.

Implement this game using classes and objects. It is up to you whether you want to include computer players or a user interface to allow human players (a terminal based interface is just fine). To better understand how the game is played, try playing it using real dice with some friends.

Below is a class diagram showing one set of classes that might be used to implement the game. This is not the only way — ask yourself if the Game or Cup classes are needed. Also which class should really keep score and apply the rules? The association lines below are shown as bi-

directional (no arrow heads), requiring decisions to be made on where the arrow heads should go. The open diamond is the aggregation annotation (collection of).