CSC1016S Assignment 3: More on Objects and Classes

Assignment Instructions

This assignment involves constructing programs in Java using object composition i.e. write class declarations that define types of object that contain other objects.

Exercise One [25 marks]

This exercise concerns constructing the display component of a digital clock that uses the 24 hour system e.g.

The specification for this component is as follows:

Class ClockDisplay

A ClockDisplay object is a holder for the current time in hours and minutes. It provides methods for advancing the current time and for obtaining the value as a formatted string.

Instance variables

private CounterDisplay hours;
private CounterDisplay minutes;

Constructors

public ClockDisplay(int pHours, int pMinutes)
    // Create a new ClockDisplay and set the display value to pHours and pMinutes.

Methods

public void setTime(int pHours, int pMinutes)
    // Set the display value to pHours and pMinutes.

public void tick()
    // Advance the display time by one minute.

public String getDisplayValue()
    // Obtain the display value; a String composed of 2 digits for hours then a colon, then 2 digits for minutes.

Given the following code fragment:

```
ClockDisplay clockDisplay = new ClockDisplay(5, 30);
clockDisplay.tick();
System.out.println(clockDisplay.getDisplayValue());
```

We should expect to see the following output:

```
05:31
```

As indicated in the specification, a ClockDisplay object actually contains two CounterDisplay objects – one for managing hours, the other for managing minutes. Here is the specification for CounterDisplay objects:

CONTINUED

Class CounterDisplay

A CounterDisplay object is a counter that cycles over a fixed range. It contains a current value and an upper limit, and it provides methods for setting the current value, advancing it, examining it, and obtaining it in the form of a 2 digit String.

Instance variables

private int value;
private int limit;

Constructors

public CounterDisplay(int pLimit)
    // Create a new CounterDisplay, set counter value to 0, and set the limit to pLimit.

Methods

public void setValue(int pValue)
    // Set the current counter value to pValue.

public int getValue()
    // Get the current value.

public void increment()
    // Advance the counter by 1, rolling over to 0 if the limit is reached.

public String getDisplayValue()
    // Obtain the display value i.e. the current counter value in the form of a 2 digit string.

Given the following code fragment, for example:

```
CounterDisplay counterDisplay = new counterDisplay(3);
counterDisplay.increment();
counterDisplay.increment();
System.out.println(counterDisplay.getDisplayValue());
counterDisplay.increment();
System.out.println(counterDisplay.getDisplayValue());
```

We should expect to see the following output:

```
02
00
```

To shed light on how a ClockDisplay object utilises its CounterDisplay objects, here's a partially completed implementation of the ClockDisplay constructor:

```
public ClockDisplay(int pHours, int pMinutes) {
    hours = new CounterDisplay(24);
    hours.setValue(pHours);
    //...
}
```

The task

Write complete class declarations for CounterDisplay and ClockDisplay. On the Vula assignment page you will find test harness classes (*TestClockDisplay.java* and *TestCounterDisplay.java*) that you may wish to use.

CONTINUED

Exercise Two [25 marks]

Given the following English language description of a prepaid mobile phone account, develop an Account class (*Account.java*) that uses a Plan class (*Plan.java*), and construct a test harness class that may be used to create an account and test its functions.

- An account has a balance from which the cost of calls is deducted.
    - Initially the balance is zero cents.
    - An account can be 'topped up'.
- All monetary transactions are conducted in ZAR cents.
- When the customer dials a number, their account is queried to determine the available airtime. (You may assume that calls are never allowed to exceed it.)
- On completion of a call, given the duration, the account is updated.
- The cost of calls/available air time is dictated by the account plan.
    - A plan has a name.
    - A plan has a cents per second call rate.
- The customer can query their account.

Tips:

- Identify the interface between Account and Plan objects and the rest of the mobile phone system.
- Think about the best place for calculations and alterations of data to take place.
    - For instance, if an Account object can delegate calculation of the cost of a call to its Plan object then it doesn't need to know about call rates when updating the balance.
- Think about appropriate operations on an account. For instance, given the description, an account balance can be topped up, it cannot be 'set'.
- If some aspect of behaviour is not described, then we don't need it.
    - For instance, you may assume that call rates are fixed for all time!
- Your test harness does not need to be an 'all singing and dancing' affair. Hard code just enough to prove to yourself and a tutor that everything works.

Marking criteria

You must submit your answer to the automarker, however, the submission will be manually marked by a tutor. (Ask a tutor to do this for you in your lab.)

- Do the Account and Plan class support call accounting? [11 marks]
- Is there suitable delegation? [5 marks]
- Is the data encapsulated? [7 marks]

CONTINUED

Exercise Three [25 marks]

This exercise is an evolution of the Meteorology problem from assignment two. In that exercise we had a program that gathered statistics on phenomena such as temperature and humidity. In this exercise we now have the notion of a 'MonitoringStation' type of object that serves the same function. We can imagine some sort of network of such stations – as must be the case for weather forecasting – and thus programs that contain many MonitoringStation objects.

This exercise involves composition of objects and the use of arrays.

Consider the following specification:

Class MonitoringStation

A MonitoringStation represents an entity that collates measurements of phenomena such as temperature, pressure, thickness, height, depth, acidity. For each phenomenon it records the maximum reading, minimum reading, number of readings so far, and the average.

Instance variables

private String name;
private Collator[] phenomena;

Constructors

public MonitoringStation(String name, String[] phenomenaNames)
// Create a MonitoringStation with the given name that will record statistics for the given
// phenomena.

Methods

public void recordReading(String phenomenonName, double reading)
// Update the records for the given phenomenon.

public double average(String phenomeonName)
// Get the current value.

As indicated by the specification, a Monitoring station contains an array of Collator objects, one per phenomenon.

- The constructor accepts an array of phenomena names. It must create and store an array of Collator objects (one Collator per phenomenon).
- The methods accept the name of a phenomenon as a parameter. They must search the array for the corresponding Collator object.
- On the Vula page for this assignment you will find a Collator class that you should use. It's slightly different to the one you were asked to construct: a Collator **has a label**; it can be labelled with the name of the phenomenon it represents.

On the Vula page you will also find a test harness class called StationUI. You can use this to test your code.

Sample I/O:

```
Monitoring Station Test Harness

Enter the station name:
Zebra
```

CONTINUED

```
Enter a comma separated list of the phenomena to be recorded:
height, width

Make a selection and press return:
0.    Quit
1.    Record a phenomenon measurement.
2.    View the average reading for a phenomeon.
1
Enter the phenomenon name and value (e.g. 'temperature 7'):
height 13.6

Make a selection and press return:
0.    Quit
1.    Record a phenomenon measurement.
2.    View the average reading for a phenomeon.
1
Enter the phenomenon name and value (e.g. 'temperature 7'):
height 12

Make a selection and press return:
0.    Quit
1.    Record a phenomenon measurement.
2.    View the average reading for a phenomeon.
1
Enter the phenomenon name and value (e.g. 'temperature 7'):
width 100

Make a selection and press return:
0.    Quit
1.    Record a phenomenon measurement.
2.    View the average reading for a phenomeon.
2
Enter the phenomenon name:
height
The average value for phenomeon height is 12.80.

Make a selection and press return:
0.    Quit
1.    Record a phenomenon measurement.
2.    View the average reading for a phenomeon.
0
```

CONTINUED

Exercise Four [25 marks]

A rational number is one representable as:

$$\frac{numerator}{denominator}$$

Where numerator and denominator are integers.

Here is the specification for a rational number class:

Class Rational

Simple representation of a rational number.

Instance variables

private int numerator;
private int denominator;

Constructors

public Rational(int numerator, int denominator)
// Create a Rational object that represents the rational number with the given numerator and // denominator.

Methods

public Rational add(Rational other)
// Return the result of adding this rational number to the other provided as a parameter.

public Rational multiply(Rational other)
// Return the result of multiplying this rational number with the other provided as a parameter.

public String toString()
// Obtain a String representation of this rational number in the form 'numerator/denominator'.

With this class we can write code such as follows:

```
Rational r1 = new Rational(1, 4);
Rational r2 = new Rational(1, 8);
Rational r3 = r1.add(r2);
Rational r4 = r1.multiply(r2);
System.out.println(r3);
System.out.println(r4);
```

This fragment represents adding 1/4 to 1/8, and multiplying 1/4 to 1/8. It produces the following output:

```
3/8
1/32
```

On the Vula page for this assignment you will find a partially completed Rational class. Your task is to implement the 'add' and 'multiply' methods. You will need to hardcode a simple test harness to check your work.

Note: The Rational class constructor simplifies i.e. new Rational(4, 8) is equivalent to new Rational(1, 2) so don't be surprised when this happens!

CONTINUED

Marking and Submission

Submit the *CounterDisplay.java* and *ClockDisplay.java* files; the *Plan.java, Account.java and test harness* files; the *MonitoringStation.java,* and *Rational.java* files contained within a single .ZIP folder to the automatic marker. The zipped folder should have the following naming convention:

yourstudentnumber.zip

END