CSC1016S Assignment 7: Interfaces, Inheritance

Assignment Instructions

This set of exercises reinforces material on the Object class and introduces useful interfaces in the API that can enhance the classes that you build for yourself.

The exercise involves:

- judicious overriding of methods inherited from the object class (toString, equals, hashcode),
- exploiting and implementing the Comparable and Iterable interfaces,
- exploiting features of the java.util.Collections class such as the sort and binary search methods.

We have a project that consists of (i) a number of classes intended to be part of a larger project/system, and (ii) a class that has been written to test that they work correctly. At the moment they don't because there are interfaces that have not been implemented and methods that are missing. The challenge is to sort out the problems.

The Scenario

The scenario concerns monitoring road usage. The idea is that software is under construction that may be used to record, for given periods on given days, the vehicles that pass on a given section of road. For instance, assuming such software, we might wish to record the vehicles that travel along University Avenue on Monday 23rd August between eight and nine am.

The reason for recording such information is of course that it may be useful in planning future road construction or in assessing taxation.

If we want to monitor road usage, we record a series of observations of vehicles. Ideally for each observation we'd record the time and relevant vehicle details such as registration, number of occupants and so on. As it stands however, the current software is quite limited. The only things recorded are the registrations of the observed vehicles.

Limited progress has been made. The software currently consists of three classes (available on the Vula page):

- A Registration class, objects of which represent car registrations.
- An Observations class, objects of which serve to represent a series of observations of vehicles.
- An enumerated type called "Province", values of which represent South African provinces from the point of view of registration identifiers.

The Province type provides a named value for each province, and a method for obtaining the correct Province object for a given registration identifier. The following fragment demonstrates the use of these.

```
Province p = Province.WESTERN_CAPE;
System.out.println(p);
final String plate = "CHZ897 L";
p = Province.identifyProvince(plate);
System.out.println(p);
```

CONTINUED

```
System.out.println(p==Province.GAUTENG);
System.out.println(p==Province.LIMPOPO);
```

When executed, the fragment outputs "`Western Cape`", then "`Limpopo`", then "`false`", and then "`true`".

Essentially, an Observations object stores a sequence of vehicle registrations in the order in which the corresponding vehicles were observed. e.g. say an observation of vehicle CA 976543 is followed by an observation of CDS791 MP, followed by vehicle CHZ897 L, followed by another observation of vehicle CA 976543. Then the sequence stored would be <<CA 976543, CDS791 MP, CHZ897 L, CA 976543>>. A vehicle may be observed more than once.

There's actually a fourth class available at the moment. It's a test harness that tests that the Observations and Registration classes work correctly. Using observation data stored in a file, it runs five tests:

1. GetTotalTest : check the Observations class "getTotal" method works.
2. IteratorTest: check the Observations "iterator" method works.
3. ObservedTest : check the Observations "observed" method works.
4. NumberObservedTest : check the Observations "numberOfObservations" method works.
5. GetVehiclesTest : check the Observations "getVehicles" method works.

As the main method in the test suite runs each test, each test reports on what the expected output should be.

Exercise One [15 marks]

Currently the tests do not produce the correct results. Can you fix the problem by judiciously adding methods to the Registration class?

Hint: The issue concerns overriding methods from the Object class to ensure correct printing and correctly testing for object equivalence.

Exercise Two [10 marks]

When we wrote test 2 we tried to use the following code but it wouldn't compile.

```
for(Registration reg : observations) {
      System.out.println(reg);
}
```

It would be convenient for programmers using the Observations class to be able to use a for-each look like this. It turns out that the problem can be solved by having the Observations class implement the java.lang.Iterable interface.

Make the necessary changes to the Observations class and modify the TestHarness class so that it uses a for-each for test 2.

NOTE: This question is part manually marked.

CONTINUED

Exercise Three [15 marks]

Vehicle registrations have a natural alpha numeric ordering e.g. the registrations CA 976543, CDS791 MP, CHZ897 L are in order. It makes sense to be able to obtain this ordering when working with a set of registrations. For example, test 5 obtains the set of registrations for all vehicles observed. It would be easier to check the results if they were output in order. The java.util.Collections class provides a method that can be used on lists that is called "sort". We'd like test 5 to do something like this:

```
/*
 * Test Five: check the Observations "getVehicles" method works.
*/
System.out.println("Testing to see the registrations of all the
vehicles that have been observed.");
List<Registration> result = observations.getVehicles();
// Print them in order so can easily check result.
Collections.sort(list);
System.out.println("The result is "+list);
System.out.println("(should be "+VEHICLES_USED+")");
```

The sort method operates on objects that are Comparable. Solve this problem by ensuring the Registration class implements the java.lang.Comparable interface, and then replace the existing test 5 with the above code.

NOTE: This question is part manually marked.

Exercise Four [45]

Having ironed out the bugs in the basic system, it really would be better if the software were more sophisticated. Specifically, if when recording an observation, the software could store the Registration plate and the time at which it was observed.

Our plan for extending the software is to develop:

- A Time class, where objects of the class represent a time in twenty-four hour clock. More than one Time object may represent the same twenty four hour time. Time objects have a natural ordering.
- A class called "Observation", where an object of the class stores a Time and a Registration. It's possible that more than one Observation object may represent the same observation. Observation objects are also naturally ordered by time.
- A new version of the Observations class called ObservationsList that stores a sequence of Observation objects.

Fortunately, a suitable time class is readily available (the one from the race time exercise of Assignment five).

The specifications for the other classes follow:

CONTINUED

Class ObservationsList

An ObservationsList maintains a list of observations of vehicles, say for the purpose of monitoring road use.

Interfaces

Iterable

Constructors

public ObservationsList()
    // Create a new ObservationsList object.

Methods

public void record (Observation observation)
    // Add the given observation to this observations list.

public void record(Registration reg, Time time)
    // Create an Observations object and add it to this observations list.

public int getTotal()
    // Obtain the total number of observations.

public List<Registration> getVehicles()
    // Obtain a list of the registration identifiers of the vehicles observed.

public ObservationsList getObservations(final Registration identifer)
    // Obtain an observations list that only contains all observations of the vehicle
    // with the given registration identifier.

public ObservationsList getObservations(final Time s, final Time e)
    // Obtain an observations list that only contains observations made between time s and time e
    // inclusive. (Requires that s.comareTo(e)≤0)

public Iterator<Observation> iterator()
    // Obtain an iterator that can be used to view the observations one-by-one.

(We considered making this class a subclass of the existing Observations class but it wouldn't have worked. For instance, the recordObservation(Registration reg) method would break this classes functionality since no Time component is provided.)

CONTINUED

Class Observation

An Observation records the time at which a particular vehicle was observed. It comprises a time and a registration.

Interfaces

Comparable

Constructors

```
public Observation(final Registration registration, final Time time)
        // Create a new Observation recording the observation of the vehicle with the given registration at
        // the given time.
```

Methods

```
public Time getTime()
        // Obtain the time at which the observation was made.

public Registration getIdentifier()
        // Obtain the registration identifier of the vehicle observed.

public boolean isFor(final Registration identifier)
        // Returns true if this observation is of the vehicle with the given identifier.

public boolean inPeriod(final Time s, final Time e)
        // Returns true if this observation was made between times s and e inclusive.
        // (Requires that s.compareTo(e)≤0)

public boolean equals(Object o)
        // Returns true if o is an Observation and the registration and time match this observation,
        // otherwise returns false.

public int compareTo(Observation other)
        // Compares this observation to the other observation on the basis of time.
        // Returns -1 if this observation occurred before the other, 0 if the observations are coincident,
        // and +1 if this observation occurred after the other.

public String toString()
        // Return a string of the form [<time>, <registration identifier>]
```

Implement the classes and construct a simple test harness to check your work.

Challenge Five [15 marks]

As a test of the quality of your answer to challenge four, consider the possibility that it might be possible Observation objects are not added in the order in which they occur. Can you ensure that, if this is the case, that observations are inserted in the sequence in the right place?

HINT: you might consider looking at the java.util.Collections binarySearch method.

Marking and Submission

Submit the *Registration.java, Observations.java, TestSuite.java, Observation.java, and Observations.java* files contained within a single .ZIP folder to the automatic marker. The zipped folder should have the following naming convention:

CONTINUED

yourstudentnumber.zip

END

CONTINUED