Lab 9 Collections/Generics

CST8132

Due:

See Blackboard for due date.

Marks:

Demonstration is mandatory in order to receive a grade for this Assignment as a whole.

Total out of 20 marks (worth 11% of term mark):

- Demonstration of correct program behavior: 5 marks
- Correct implementation: 5 marks
- Javadoc: 5 marks
- Junit4 Test: 5 marks

Recommended Reading:

ArrayLists Section 7.16 of Deitel and Deitel, Java How to Program (10th Edition)

Generics Chapter 20 of Deitel and Deitel, Java How to Program (10th Edition)

Generics Java Tutorial Online:
http://docs.oracle.com/javase/tutorial/java/generics/index.html

Purpose:

To use a Java Collections class (ArrayList) to gain experience with Java Collections, including Iterators and Comparators.

Demonstration:

Run your program for your lab instructor, and be prepared to point out various elements of your solution when requested.

Description:

Using your solution to Lab 6 as a starting point, enhance the application according to the following points:

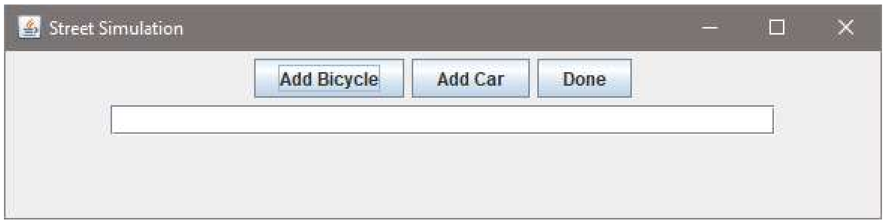
1. Street class: Use an ArrayList instead of an array to store the collection of vehicles.

Details: See Section 7.16 of the Deitel textbook to see how ArrayLists are created and used.

Your Vehicle, Car, Bicycle classes will not need any change.

2. Street class: Instead of hard-coding the creation of vehicles, the program will use a Swing GUI component of your choosing to prompt the user for and receive the type of the next vehicle to add to the collection. This means your program will need to support an indefinite number of vehicles (and ArrayLists are good at that). As each vehicle is added, the program will use its GUI to display to the user the name of the vehicle that was added to the collection.

   Details: use the provided SampleGui as a template for this GUI (the SampleGui is a stand alone program, run-it and see how it works before using as a template). Once you are familiar with SampleGUI, rename as the Street class and add your Street functions from Lab 6. A reasonable GUI for this program would have three buttons and a JTextField as pictured below:

   For example, when the user clicks the "Add Bicycle" button, your program would create a new Bicycle, add it to the collection, and display something like "Bicycle0 was added to the Street" in the JTextField. When the user is finished adding cars and bicycles to the Street, the user should click the "Done" button, which will cause your program to advance to the next stage, which is simulation (see next step).

3. Sample output for the simulation stage appears at the end of this document. The simulation stage prints the output of the simulation on the console, as was done in the previous version (Lab 6). In each iteration of the simulation loop, the program will do all of the following:
   a. Print the name, speed and sound of each vehicle, in order of decreasing speed (faster vehicles will be first in the ordering). If a bicycle and a car have the same speed, then the bicycle should come first. Your program should do this by sorting the ArrayList using the `Collections.sort(List<T> list, Comparator<? super T> c)` method, and then using an Iterator to go through the Vehicles one by one, printing its name, speed and sound.

Details for Step 3a: The Java Collections framework will do most of the work here, but we need to figure out how to use it.

Let's start with sorting our ArrayList of Vehicles, and let's suppose the variable name we chose for it is `vehicles`. This variable is of type `ArrayList<Vehicle>` but the method we want to use to sort that ArrayList takes a thing of type `List<T>` as its first parameter (see above). We can check the Java API docs and verify that `ArrayList<T> implements List<T>`. So that means every ArrayList<T> is a List<T>, so we'll pass our `vehicles` ArrayList as the first parameter to the Collections.sort method.

Now, what about the type of the second parameter to the Collections.sort method: `Comparator<? super T>`. The `? super T` type parameter is called a lower bounded wildcard, and it's like a plain type parameter `T` except it states the type we actually supply must be T or a super type of T. That is, if we call the Collections.sort method as we intend to, with a first argument of type `List<Vechicle>`, then the second argument must be of type `Comparator<? super Vehicle>`, where `? super Vehicle` means class Vehicle or any superclass of Vehicle. After all that, for our purposes we will use Vehicle as the type parameter, which implies we will be using

```
Collections.sort(List<Vehicle> list, Comparator<Vehicle> c)
```

Ok, so what will we pass as the second parameter, c? We need an object of type `Comparator<Vehicle>`. The Java API docs tell us that Comparator<T> is an interface, and the two methods in that interface are

```
int compare(T o1, T o2)

boolean equals(Object obj)
```

We need a class that implements Comparator<Vehicle>, so we can pass an instance of that class as the second argument when we call Collections.sort. We don't need to implement the equals method, because it's provided by the Object class. A new class, VehicleComparator, could look like this

```
public class VechicleComparator implements
Comparator<Vehicle> {

    int compare(Vehicle v1, Vehicle v2) {
            // insert code that will return
            // negative int if v1 is before v2,
            // 0 if v1 is not before or after v2,
            // positive int if v1 is after v2
            // see above for what it means for a vehicle
            // to be before another vehicle and don't
            // forget there is an instanceof operator
        }
    }
```

Alternatively, we could modify our existing Street class to implement Comparator<Vehicle>, and then the Street Object could pass a reference to itself as the second parameter to Comparator.sort.

MORE DETAIL: Now we need an Iterator to iterate through our vehicles, as specified in the requirements. This one is easy. Every object of type List (like our vehicles arraylist) will return an Iterator when we invoke its iterator() method. So we can get an iterator this way:

```
Iterator iterator = vehicles.iterator();
```

This iterator will have methods

```
boolean hasNext()

Vehicle next()    // E next() in the generic sense
```

The requirements of this lab specify that you need to use these methods to control the loop that prints the vehicles names and speeds (so that you will learn about iterators and their methods).

b. The second thing in each iteration of the simulation loop is, in random order, each vehicle will make its noise (the name of the Vehicle will be printed, followed by its noise). Your program should do this by making use of the Collections.shuffle(List<T> list) method, and then making use of an Iterator to go through the Vehicles one by one.

Details: This is similar to Step 3a. above, except instead of putting the vehicles in proper order, we randomize their order with the Collections.shuffle method.

   c. The last thing in each iteration of the simulation loop is a single random vehicle has its pedal pushed.

      Details: This is very similar to the same step from Lab6

4. For step 5, you need to implement the provided SimInterface interface in your Street class.

5. Create a JUNIT4 test class called SimulationTest, and within this class:
   a. Create a test method called ”test”.
   b. Create an instance of the Street class.
   c. Pass the following String array (using the “parse” function of the SimInterface):

      {“car”, “car”, “car”, “bicycle”, “bicycle”}

   d. Invoke simulate() function of the Street object.
   e. Obtain the total of car speeds, divide by 10.
   f. Obtain the total of bicycle speeds, divide by 4;
   g. Add the result in step e and f, pass if it is 6, fail if it is not.

Sample Output

```
Update on the street 0:
--- sorted:
Car0 speed: 0, purr
Car1 speed: 0, purr
Bicycle2 speed: 0, sigh
Bicycle3 speed: 0, sigh
Bicycle4 speed: 0, sigh
--- shuffled:
Car1 speed: 0, purr
Bicycle3 speed: 0, sigh
Bicycle2 speed: 0, sigh
Bicycle4 speed: 0, sigh
Car0 speed: 0, purr
The pedal of Bicycle2 was pushed


Update on the street 1:
--- sorted:
Bicycle2 speed: 4, grunt
```

```
Car1 speed: 0, purr
Car0 speed: 0, purr
Bicycle3 speed: 0, sigh
Bicycle4 speed: 0, sigh
--- shuffled:
Car1 speed: 0, purr
Bicycle2 speed: 4, grunt
Bicycle4 speed: 0, sigh
Car0 speed: 0, purr
Bicycle3 speed: 0, sigh
The pedal of Car1 was pushed


Update on the street 2:
--- sorted:
Car1 speed: 10, vroom
Bicycle2 speed: 4, grunt
Car0 speed: 0, purr
Bicycle4 speed: 0, sigh
Bicycle3 speed: 0, sigh
--- shuffled:
Bicycle3 speed: 0, sigh
Car0 speed: 0, purr
Car1 speed: 10, vroom
Bicycle4 speed: 0, sigh
Bicycle2 speed: 4, grunt
The pedal of Bicycle2 was pushed


Update on the street 3:
--- sorted:
Car1 speed: 10, vroom
Bicycle2 speed: 8, grunt
Car0 speed: 0, purr
Bicycle3 speed: 0, sigh
Bicycle4 speed: 0, sigh
--- shuffled:
Bicycle4 speed: 0, sigh
Bicycle2 speed: 8, grunt
Bicycle3 speed: 0, sigh
Car1 speed: 10, vroom
Car0 speed: 0, purr
The pedal of Car1 was pushed


Update on the street 4:
--- sorted:
Car1 speed: 20, vroom
Bicycle2 speed: 8, grunt
Car0 speed: 0, purr
Bicycle4 speed: 0, sigh
Bicycle3 speed: 0, sigh
--- shuffled:
Bicycle2 speed: 8, grunt
Bicycle3 speed: 0, sigh
```

```
Bicycle4 speed: 0, sigh
Car1 speed: 20, vroom
Car0 speed: 0, purr
The pedal of Car0 was pushed


Update on the street 5:
--- sorted:
Car1 speed: 20, vroom
Car0 speed: 10, vroom
Bicycle2 speed: 8, grunt
Bicycle3 speed: 0, sigh
Bicycle4 speed: 0, sigh
--- shuffled:
Bicycle3 speed: 0, sigh
Car1 speed: 20, vroom
Car0 speed: 10, vroom
Bicycle2 speed: 8, grunt
Bicycle4 speed: 0, sigh
The pedal of Bicycle3 was pushed
```

Or… (if there are no vehicles)

```
No vehicles
```