Project: Vehicles – Part 2

Deliverables

Your project files should be submitted for grading by the due date and time specified. Note that there is also an optional Skeleton Code assignment (ungraded) which will indicate level of coverage your tests have achieved. The files you submit to skeleton code assignment may be incomplete in the sense that method bodies have at least a return statement if applicable or they may be essentially completed files. In order to avoid a late penalty for the project, you must submit your files to the Completed Code assignment no later than 11:59 PM on the due date. If you are unable to submit to the grading system, you should e-mail your project Java files in a zip file to your TA before the deadline. Your grade will be determined, in part, by the tests that you pass or fail in your test files and by the level of coverage attained in your source files, as well as our usual correctness tests.

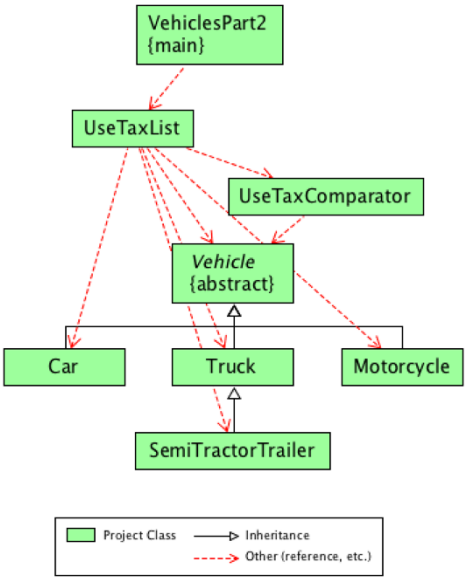
Files to submit for grading:

From Vehicles – Part 1

- Vehicle.java, VehicleTest.java
- Car.java, CarTest.java
- Truck.java, TruckTest.java
- SemiTractorTrailer.java, SemiTractorTrailerTest.java
- Motorcycle.java, MotorcycleTest.java

New in Vehicles – Part 2

- UseTaxList.java, UseTaxListTest.java
- UseTaxComparator.java, UseTaxComparatorTest.java
- VehiclesPart2.java, VehiclesPart2Test.java



Recommendations

You should create new folder for Part 2 and copy your relevant Part 1 source and test files (listed above) to it (i.e., do not include VehiclesPart1.java, VehiclesPart1Test.java). You should create a new jGRASP project and add these source and test files as well as new ones as they are created. You may find it helpful to use the "viewer canvas" feature as you develop and debug your program.

Specifications

Overview: This project is Part 2 of three that will involve calculating the annual use tax for vehicles where the amount is based on the type of vehicle, its value, and various tax rates. In Part 1, you developed Java classes that represent categories of vehicles: car, truck, semi-tractor trailer (a subclass of truck), and motorcycle. In Part 2, you will implement three additional classes: (1) UseTaxComparator that implements the Comparator interface, (2) UseTaxList that represents a list of vehicles and includes several specialized methods, and (3) VehiclesPart2 which contains the main method for the program. Note that the main method in VehiclesPart2 should create a UseTaxList object, read the data file using the readVehicleFile method. VehiclesPart2 then prints the summary, the vehicles listed by owner and the vehicles listed by use tax amount, and the list of excluded records. You can use VehiclesPart2 in conjunction with interactions by running the program in the canvas (or debugger with breakpoints) until the UseTaxList object has been created and the data has

been read in. You can then enter interactions in the usual way. You can also step into the methods of interest when you run "Debug". In addition to the source files, you will create a JUnit test file for each class and write one or more test methods to ensure the classes and methods meet the specifications.

- **Vehicle.java**

  Requirements and Design: In addition to the specifications in Part 1, the Vehicle class should implement the Comparable interface for Vehicle objects.
  - `compareTo`: Takes a Vehicle as a parameter and returns an int indicating the results of comparing Vehicle objects based on their respective owners. This method is required since the Vehicle class implements the Comparable interface for Vehicle.

- **Car, Truck, SemiTractorTrailer, and Motorcycle**

  Requirements and Design: No changes from the specifications in Vehicles – Part 1.

- **UseTaxList.java**

  Requirements: The UseTaxList class provides methods for reading in data and generating reports (summary and list), adding a Vehicle, and sorting the vehicles by owner and by use tax amount.

  Design: The UseTaxList class has fields, a constructor, and methods as outlined below.

(1) **Fields:** All fields below should be private and initialized as indicated in the constructor.
(a) *taxDistrict* of type String is the entity in charge of the use tax list that is initialized to "not yet assigned".
(b) *vehicleList* is an array of type Vehicle that is initialized to length zero.
(c) *excludedRecords* is a String array that is initialized to length zero.
Note that the vehicles array and excluded records array should grow as items are added. Hence, the length of these arrays should be the same as the number of objects in the arrays. This eliminates the need for separate variables to track the number of objects contained in the arrays, and it also allows the use of for-each loops with the arrays.

(2) **Constructor:** The constructor has no parameters. The fields for taxDistrict, vehicleList, and excludedRecords should be initialized in the constructor as described in (1) above.

(3) **Methods:** Usually a class provides methods to access and modify each of its instance variables (i.e., getters and setters) along with any other required methods. The methods for UseTaxList are described below.
  - `readVehicleFile` has no return value, accepts the data file name as a String, and has a throws clause for FileNotFoundException . This method creates a Scanner object to read in the file and then reads it in line by line. The first line of the file contains the use tax list taxDistrict and each of the remaining lines contains the data for a vehicle. After reading in the taxDistrict name, the "vehicle" lines should be processed as follows. A

vehicle line (or record) is read in, a second scanner is created on the line, and the individual values for the vehicle are read in. Be sure to "trim" each value read in. All values should be read as strings and then non-String values should then "parsed" into their respective values using the appropriate wrapper class (e.g., Double.parseDouble(..)). After the values on the line have been read in, an "appropriate" vehicle object is created and added to the vehicles array using the addVehicle method. If the vehicle type is not recognized, the record/line should be added to the excluded records array using the addExcludedRecord method. The data file is a "semi-colon separated values" file; i.e., if a line contains multiple values, the values are delimited by semi-colons. So after you set up the scanner for the vehicle lines, you need to change the delimiter to a ";" by invoking `useDelimiter(";")` on the Scanner object. Each vehicle line in the file begins with a category for the vehicle. Your switch statement should determine which type of Vehicle to create based on the first character of the category (i.e., C, T, S, and M for Car, Truck, SemiTractorTrailer, and Motorcycle respectively). The second field in the record is the owner, followed by yearMakeModel, value, alternative fuel, as well the values appropriate for the category of vehicle represented by the line of data. That is, the items that follow alternative fuel correspond to the data needed for the particular category (or subclass) of Vehicle. An example file, *vehicle_1.txt*, is available for download from the course web site. Below are example data records (the first line/record containing the use tax list taxDistrict name is followed by vehicles lines/records). Note that two of the records below have invalid categories.

```
Tax District 52
Car; Jones, Sam; 2017 Honda Accord; 22000; false
car; Jones, Jo; 2017 Honda Accord; 22000; true
race car; Zinc, Zed; 2013 Custom Hot Rod; 61000; false
Car; Smith, Pat; 2015 Mercedes-Benz Coupe; 110000; false
Car; Smith, Jack; 2015 Mercedes-Benz Coupe; 110000; true
Truck; Williams, Jo; 2012 Chevy Silverado; 30000; false; 1.5
Firetruck; Body, Abel; 2015 GMC FE250; 55000; false; 2.5
truck; Williams, Sam; 2010 Chevy Mack; 40000; true; 2.5
Semi; Williams, Pat; 2012 International Big Boy; 45000; false; 5.0; 4
Motorcycle; Brando, Marlon; 1964 Harley-Davidson Sportster; 14000; false; 750
```

- `getTaxDistrict` returns the String representing the name of the tax district.
- `setTaxDistrict` returns nothing, accepts a String and assigns it to the tax district.
- `getVehicleList` returns the array containing the vehicles.
- `getExcludedRecords` returns the String array representing the excluded records.
- `addVehicle` has no return value, accepts a Vehicle object, increases the capacity of the vehicles array by one, and adds the vehicle in the last position of the vehicles array. The following two lines accomplish this (assuming `vehicleList` is the vehicles array and that `java.util.Arrays` has been imported).
  ```
  vehicleList = Arrays.copyOf(vehicleList, vehicleList.length + 1);
  vehicleList[vehicleList.length - 1] = vehicleIn;
  ```
- `addExcludedRecord` has no return value, accepts a String, increases the capacity of the excludedRecords array by one, and adds the String in the last position of the excludedRecords array. (hint: see `addVehicle` above)
- `toString` returns a String representing a list of vehicles in the vehicles array (does not include a list title); accepts no parameters. A `\n` should be added before and after each Vehicle object.
- `calculateTotalUseTax` returns a double representing the total use tax for all of the vehicles in the vehicles array.

- `calculateTotalValue` returns a double representing the total value for all of the vehicles in the vehicles array.
- `summary` returns a String representing summary information for the tax district. It includes the tax district name, the total number of the vehicles, the total value for the vehicles and total use tax for the vehicles. Note that this method should call the "calculate" methods described above to get the total value and total use tax, and it should end with a `\n` character. See example output below.
- `listByOwner` returns a String representing the vehicles list by owner (the natural sorting order). The vehicles array should sorted by owner before building the String to be returned. The resulting String should include the title and list of vehicles as shown in the example output below. The title should not be preceded by `\n`. Recall, the `toString` method returns the list of vehicles.
- `listByUseTax` returns a String representing the vehicles list by the name. The vehicles array should be sorted by use tax (see the UseTaxComparator class below) before building the String to be returned. The resulting String should include the title and list of vehicles as shown in the example output below. The title should not be preceded by `\n`.
- `excludedRecordsList` returns a String representing the list vehicles records/lines that were read from the file but excluded from the vehicles array in UseTaxList. The resulting String should include the title and list of excluded records/lines as shown in the example output below.

Code and Test: See examples of file reading and sorting in the lecture notes. The Arrays.sort method in the java.util package sorts the array in place. The natural sorting order for Vehicle objects is determined by the compareTo method from the Comparable interface. If *vehicleList* is the variable for the array of Vehicle objects, it can be sorted in natural order with the following statement.

```
Arrays.sort(vehicleList);
```

The sorting order based on use tax is determined by the UseTaxComparator class which implements the Comparator interface (described below). It can be sorted with the following statement.

```
Arrays.sort(vehicleList, new UseTaxComparator());
```

After the vehicles array is sorted, the array returned by the getVehicleList method should be in the order resulting from the most recent sort.

- **UseTaxComparator.java**

Requirements and Design: The UseTaxComparator class implements the Comparator interface for Vehicle objects. Hence, it implements the following method.

- `compare(Vehicle v1, Vehicle v2)` that defines the ordering from lowest to highest based on the use tax for v1 and v2.

  Note that the *compare* method is the only method in the UseTaxComparator class. An instance of this class should be used as one of the parameters when the Arrays.sort method is used to sort by "use tax" (see above). For an example of a class implementing Comparator, see lecture notes on Comparing Objects.

- **VehiclesPart2.java**

  Requirements and Design: The VehiclesPart2 class has only a main method as described below.
  - `main` gets the file name from the command line (i.e., args[0]), creates an instance of UseTaxList, and then calls its readVehicleFile method to read in the data file and populate the vehicles array in UseTaxList object. The main method then prints the summary, the vehicle list by owner, the vehicle list by use tax, and the list of excluded records. After the summary is printed, be sure to print `\n` characters as needed before each of the three lists is printed. An example data file, *vehicles1.txt*, can be downloaded from the Lab web page. The output from main for this file is on the following page. Note that main should have a throws clause for FileNotFoundException.

  Code and Test: The example data file, vehicles1.txt, has been uploaded into the grading system and is available for your test methods to call as needed. If you want to use additional data files, you will need to use .txt as the extension and then upload the data files along with your source files. After you have implemented the VehiclesPart2 class, you should create the test file VehiclesPart2Test.java in the usual way. The only test method you need is one that creates an instance of VehiclesPart2 (to cover its default constructor), and then checks the class variable v*ehicleCount* that was declared in Vehicle and inherited by each subclass. In the test method, you should declare and create an instance of VehiclesPart2, reset v*ehicleCount*, create an args array containing the file name vehicles1.txt, call your main method in VehiclesPart2, which should result in vehicles1.txt being read in, then assert that v*ehicleCount* is eight (assuming that eight objects from the Vehicle hierarchy were created and stored in the UseTaxList object created when main is called). The following statements accomplish the test.

```
VehiclesPart2 vPart2Obj = new VehiclesPart2(); // test constructor

Vehicle.resetVehicleCount();

String[] args = {"vehicles1.txt"};
VehiclesPart2.main(args);
Assert.assertEquals(8, Vehicle.getVehicleCount());
```

UML Class Diagram

As you add your classes to the jGRASP project, you should generate the UML class diagram for the project. To layout the UML class diagram, right-click in the UML window and then click Layout > Tree Down. Click in the background to unselect the classes. You can then select the VehiclesPart1 class and move it around as appropriate, then do the same for the UseTaxList and UseTaxComparator. An example of the generated UML class diagram is shown on page 1. Note that the dependencies represented by the red dashed arrows indicate that VehiclesPart1depends on UseTaxList which in turn depends on UseTaxComparator, Vehicle, and Vehicle's subclasses. Note that UseTaxComparator only depends on Vehicle.

Example Output

```
 ----jGRASP exec: java VehiclesPart2 vehicles1.txt
------------------------------
Summary for Tax District 52
------------------------------
Number of Vehicles: 8
Total Value: $393,000.00
Total Use Tax: $12,010.00


------------------------------
Vehicles by Owner
------------------------------

Brando, Marlon: Motorcycle 1964 Harley-Davidson Sportster
Value: $14,000.00 Use Tax: $280.00
with Tax Rate: 0.005 Large Bike Tax Rate: 0.015

Jones, Jo: Car 2017 Honda Accord (Alternative Fuel)
Value: $22,000.00 Use Tax: $110.00
with Tax Rate: 0.005

Jones, Sam: Car 2017 Honda Accord
Value: $22,000.00 Use Tax: $220.00
with Tax Rate: 0.01

Smith, Jack: Car 2015 Mercedes-Benz Coupe (Alternative Fuel)
Value: $110,000.00 Use Tax: $2,750.00
with Tax Rate: 0.005 Luxury Tax Rate: 0.02

Smith, Pat: Car 2015 Mercedes-Benz Coupe
Value: $110,000.00 Use Tax: $3,300.00
with Tax Rate: 0.01 Luxury Tax Rate: 0.02

Williams, Jo: Truck 2012 Chevy Silverado
Value: $30,000.00 Use Tax: $600.00
with Tax Rate: 0.02

Williams, Pat: SemiTractorTrailer 2012 International Big Boy
Value: $45,000.00 Use Tax: $3,150.00
with Tax Rate: 0.02 Large Truck Tax Rate: 0.03 Axle Tax Rate: 0.02

Williams, Sam: Truck 2010 Chevy Mack (Alternative Fuel)
Value: $40,000.00 Use Tax: $1,600.00
with Tax Rate: 0.01 Large Truck Tax Rate: 0.03


------------------------------
Vehicles by Use Tax
------------------------------

Jones, Jo: Car 2017 Honda Accord (Alternative Fuel)
Value: $22,000.00 Use Tax: $110.00
with Tax Rate: 0.005
```

```
Jones, Sam: Car 2017 Honda Accord
Value: $22,000.00 Use Tax: $220.00
with Tax Rate: 0.01

Brando, Marlon: Motorcycle 1964 Harley-Davidson Sportster
Value: $14,000.00 Use Tax: $280.00
with Tax Rate: 0.005 Large Bike Tax Rate: 0.015

Williams, Jo: Truck 2012 Chevy Silverado
Value: $30,000.00 Use Tax: $600.00
with Tax Rate: 0.02

Williams, Sam: Truck 2010 Chevy Mack (Alternative Fuel)
Value: $40,000.00 Use Tax: $1,600.00
with Tax Rate: 0.01 Large Truck Tax Rate: 0.03

Smith, Jack: Car 2015 Mercedes-Benz Coupe (Alternative Fuel)
Value: $110,000.00 Use Tax: $2,750.00
with Tax Rate: 0.005 Luxury Tax Rate: 0.02

Williams, Pat: SemiTractorTrailer 2012 International Big Boy
Value: $45,000.00 Use Tax: $3,150.00
with Tax Rate: 0.02 Large Truck Tax Rate: 0.03 Axle Tax Rate: 0.02

Smith, Pat: Car 2015 Mercedes-Benz Coupe
Value: $110,000.00 Use Tax: $3,300.00
with Tax Rate: 0.01 Luxury Tax Rate: 0.02


------------------------------
Excluded Records
------------------------------

race car; Zinc, Zed; 2013 Custom Hot Rod; 61000; false

Firetruck; Body, Abel; 2015 GMC FE250; 55000; false; 2.5


 ----jGRASP: operation complete.
```