Project: Vehicles – Part 3

Deliverables

Your project files should be submitted to Web-CAT by the due date and time specified above (see the Lab Guidelines for information on submitting project files). You may submit your skeleton code files until the project due date but should try to do this by Friday (there is no late penalty since this is ungraded for this project). You must submit your completed code files to Web-CAT before 11:59 PM on the due date for the completed code to avoid a late penalty for the project. You may submit your completed code up to 24 hours after the due date, but there is a late penalty of 15 points. No projects will be accepted after the one day late period. If you are unable to submit via Web-CAT, you should e-mail your project Java files in a zip file to your lab instructor before the deadline. The Completed Code will be tested against your test methods in your JUnit test files and against the usual correctness tests. The grade will be determined, in part, by the tests that you pass or fail and the level of coverage attained in your Java source files by your test methods.

Files to submit to Web-CAT - - test files are required (see note 3, p. 8):

From Project 9 (see note 4, p. 8) regarding static *count* in Vehicle)

- Vehicle.java [modify constructor to throw NegativeValueException]
- Car.java, CarTest.java
- Truck.java, TruckTest.java [modify constructor to throw NegativeValueException]
- SemiTractorTrailer.java, SemiTractorTrailerTest.java [modify constructor to throw NegativeValueException]
- Motorcycle.java, MotorcycleTest.java [modify constructor to throw NegativeValueException]

From Project 10

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

New in Project 11

- NegativeValueException.java, NegativeValueExceptionTest.java
- VehiclesPart3.java, VehiclesPart3Test.java

Recommendations

You should create new folder for Project 11 and copy your relevant Project 9 and 10 source files to it (i.e., do not include VehiclesPart1.java and VehiclesPart2.java). You should create a jGRASP project and add these source files as well as those created in Project 11. You may find it helpful to use the "viewer canvas" feature in jGRASP as you develop and debug your program.

Specifications

Overview: This project is Part 3 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 implemented 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 that creates a Vehicle object, reads the data file using the readVehicleFile method, prints a summary, a vehicles list by owner and by use tax, and the list of excluded records. In Part 3 (Project 11), you are to add exception handing and invalid input reporting. You will need to do the following: (1) create a new class named NegativeValueException which extends the Exception class, (2) add try-catch statements to catch FileNotFoundException in the main method of the VehiclesPart3 class, and (3) modify the readVehicleFile in the UseTaxList class to catch/handle NegativeValueException, NumberFormatException, and NoSuchElementException in the event that these type exceptions are thrown while reading the input file.

Note that the main method inVehiclesPart3 should create a UseTaxList object and then invoke the readVehicleFile method on the UseTaxList object to read data from a file and add vehicles to the vehicleList array in the UseTaxList object. You can use inVehiclesPart3 in conjunction with interactions by running the program in the canvas (or debugger with a breakpoint) and single stepping until the variables of interest are created. You can then enter interactions in the usual way. You should create a jGRASP project upfront and then add the source and test files as they are created. All of your files should be in a single folder.

- **Vehicle** – Modifications
  (1) Change class variable vehicleCount from private to protected (see note #4, last page).
  (2) Add `equals` method (see note #6, last page).

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

  Requirements and Design: The constructors for these classes must be modified to check numeric parameters specific to their respective classes for negative values and throw a NegativeValueException as appropriate. For example, if -20000 is passed into the Vehicle constructor as the vehicle's *value*, the constructor should throw a NegativeValueException. Or if -2.5 is passed into the Truck constructor as *tons*, the constructor should throw a NegativeValueException. Since these constructors are not catching this exception, they must include NegativeValueException in their respective throws clauses.

  Testing: Since the constructors in Car, Truck, SemiTractorTrailer, and Motorcycle may throw a NegativeValueException, any method that calls one of these constructors must either catch the exception or it must throw the exception (i.e., include NegativeValueException in a throws clause for the method).

- **NegativeValueException.java**

  Requirements and Design: NegativeValueException is a user defined exception created by extending the Exception class with an empty body. The constructor for NegativeValueException should be parameterless, but it should invoke the super constructor with the String message `"Numeric values must be nonnegative"`. The inherited toString() value of a NegativeValueException will be the name of the exception and the message. This exception is to be caught in the readVehicleFile method in the UseTaxList class when a line of input data

contains a negative value for one of the numeric input values: value, tons, axles, engineSize. The NegativeValueException is to be thrown in the "vehicle" constructor that is responsible for setting the field in question. The following shows how the constructor would be called: `new NegativeValueException()` For a similar constructor, see InvalidLengthException.java in Examples\Polygons from this week's lecture notes on Exceptions.

Testing: Here is an example of a test method that checks to make sure a negative value for *value* in the constructor for Vehicle throws a NegativeValueException. Note that creating a Car invokes the constructor in Vehicle. You should consider adding test methods to check for negative values for the other numeric fields.

```
@Test public void negativeValueExceptionTest() {
   boolean thrown = false;
   try {
      Car car = new Car("Jackson, Bo", "2012 Toyota Camry", -25000, false);
   }
   catch (NegativeValueException e) {
      thrown = true;
   }
   Assert.assertTrue("Expected NegativeValueException to be thrown.",
                     thrown);
   /* or alternatively: */
   Assert.assertEquals("Expected NegativeValueException to be thrown.",
         true, thrown);
}
```

- **UseTaxList.java**

Requirements and Design: The UseTaxList class provides methods for reading in the data file and generating reports.

Design: In addition to the specifications in Vehicles – Part 2, the existing readVehicleFile method must be modified to catch following: NegativeValueException, NumberFormatException, and NoSuchElementException. When these exceptions occur, an appropriate message along with the offending line/record should be added the excludedRecords array.

- `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 message `"Invalid Vehicle Category in:\n"` and 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. For each *incorrect* line scanned (i.e., a line of data that contain missing data or invalid numeric data), your method will need to handle the invalid items properly. If a line includes invalid numeric data (e.g., the value for *value*, a double, contains an alphabetic character), a NumberFormatException (see note 1, p. 8) will be thrown automatically by the Java Runtime Environment (JRE). Your readVehicleFile method should catch and handle NumberFormatException, NoSuchElementException (for missing values), and NegativeValueException (thrown in vehicle constructors when a negative value is passed in) as follows. In each catch clause, a String consisting of the exception, the comment `" :\n"`, and the *line* with the invalid data should be added to the excludedRecords array. For example, in the catch clause for NumberFormatException e, the String resulting from the following expression should be added to the excludedRecords array.

```
e + " in:\n" + line
```

The file *vehicles2.txt* is available for download from the course web site. Below are example data records (the first line/record containing the tax district name is followed by vehicle lines/records):

```
Tax District 52
Car; Jones, Sam; 2014 Honda Accord; 22000; false
car; Jones, Jo; 2014 Honda Accord; 22000; true
car; Jones, Pat; 2014 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, Pet; 2015 Mercedes-Benz Coupe; 110000
Car; Smith, Pop; 2015 Mercedes-Benz Coupe; 110000; yes
Car; Smith, Jack; 2015 Mercedes-Benz Coupe; 110000; true
Truck; Williams, Jo; 2012 Chevy Silverado; 30000; false; 1.5
Truck; Williams, Alex; 2012 Chevy Silverado; 30000; false; -3.0
Firetruck; Body, Abel; 2015 GMC FE250; 55000; false; 2.5
truck; Williams, Sam; 2010 Chevy Mack; 40000; true; 2.5
truck; Williams, Bam; 2010 Chevy Mack; 40000; true; five
Semi; Williams, Pat; 2010 International Big Boy; 45000; false; 5.0; 4
Semi; Williams, Mat; 2012 Volvo Big Mack; 35000; false; 5.0; -4
Motorcycle; Brando, Marlon; 1964 Harley-Davidson Sportster; 14000; false; 750
Motorcycle; Rider, Easy; 1967 Harley-Davidson Electra; 10000; false; -1200
```

- **VehiclesPart3.java**

  Requirements and Design: The VehiclesPart3 class has only a main method as described below. In addition to the specifications in Project 10, the main method should be modified to catch and handle an FileNotFoundException if one is thrown in the readVehicleFile method of the UseTaxList class.

  In Part 3, `main` reads in the file name from the command line as was done in Vehicles – Part 2, creates an instance of UseTaxList, and then calls the readVehicleFile method in the UseTaxList class to read in the data file. After successfully reading in the file, the main method then prints the summary, vehicle list by owner, vehicle list by use tax, and the list of excluded records. The main method should not include the *throws FileNotFoundException* in the declaration. Instead, the main method should include a try-catch statement to catch FileNotFoundException when/if it is thrown in the readVehicleFile method in the UseTaxList class. This exception will occur when an incorrect file name is passed to the readVehicleFile method. After this FileNotFoundException is caught, print the messages below and end.

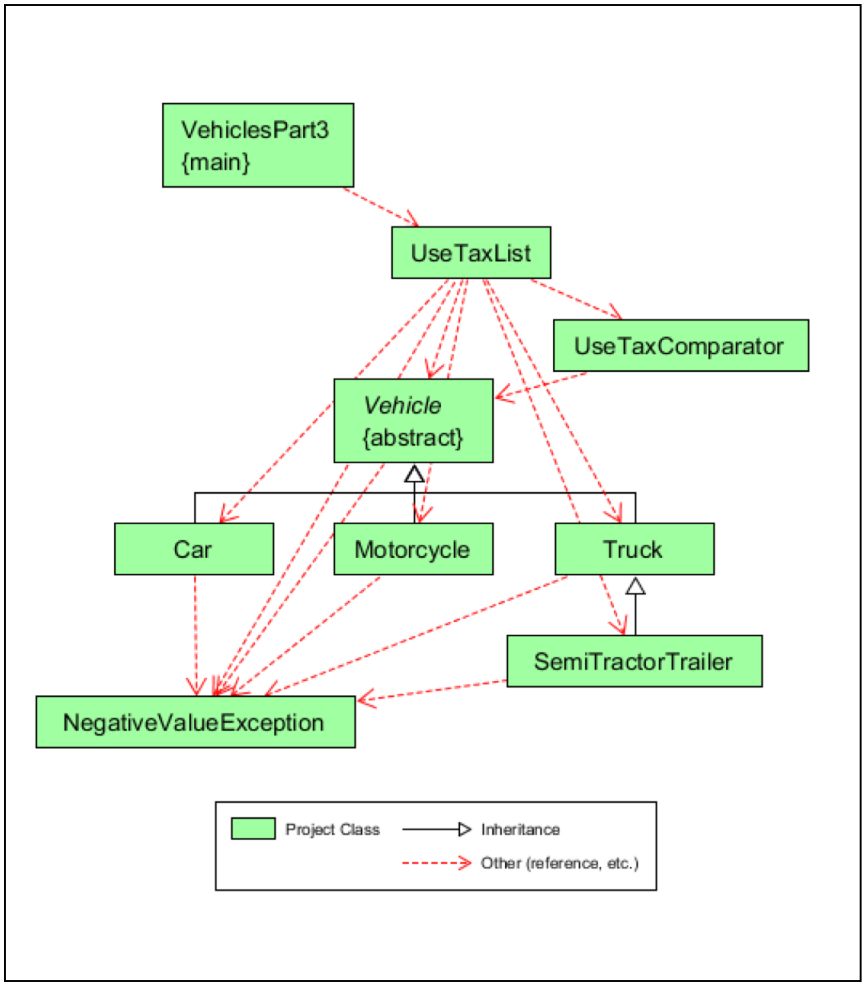
  ```
  *** File not found.
  Program ending.
  ```

  Also, if the user runs the program without a command line argument (e.g., `args.length == 0`), main should print the following message and end.

  ```
  *** File name not provided by command line argument.
  Program ending.
  ```

  An example data file can be downloaded from the Lab web page. The program output for *vehicles2.txt* begins on the next page after the UML class diagram. See note 2 on p. 8 for testing your main method.

UML Class Diagram



Example Output for vehicles2.txt

```
----jGRASP exec: java -ea VehiclesPart3 vehicles2.txt

------------------------------
Summary for Tax District 52
------------------------------
Number of Vehicles: 9
Total Value: $503,000.00
Total Use Tax: $15,310.00


------------------------------
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 2014 Honda Accord (Alternative Fuel)
Value: $22,000.00 Use Tax: $110.00
with Tax Rate: 0.005

Jones, Sam: Car 2014 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

Smith, Pop: 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 2010 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 2014 Honda Accord (Alternative Fuel)
Value: $22,000.00 Use Tax: $110.00
with Tax Rate: 0.005

Jones, Sam: Car 2014 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 2010 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

Smith, Pop: 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
------------------------------

NegativeValueException: Numeric values must be nonnegative in:
car; Jones, Pat; 2014 Honda Accord; -22000; true

Invalid Vehicle Category in:
race car; Zinc, Zed; 2013 Custom Hot Rod; 61000; false

java.util.NoSuchElementException in:
Car; Smith, Pet; 2015 Mercedes-Benz Coupe; 110000

NegativeValueException: Numeric values must be nonnegative in:
Truck; Williams, Alex; 2012 Chevy Silverado; 30000; false; -3.0
```

```
Invalid Vehicle Category in:
Firetruck; Body, Abel; 2015 GMC FE250; 55000; false; 2.5

java.lang.NumberFormatException: For input string: "five" in:
truck; Williams, Bam; 2010 Chevy Mack; 40000; true; five

NegativeValueException: Numeric values must be nonnegative in:
Semi; Williams, Mat; 2012 Volvo Big Mack; 35000; false; 5.0; -4

NegativeValueException: Numeric values must be nonnegative in:
Motorcycle; Rider, Easy; 1967 Harley-Davidson Electra; 10000; false; -1200


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

Notes:

1. **Exceptions for numeric items –** This project assumes that you are reading each double value as String using next() and then parsing it into a double withDouble.parseDouble(...) as shown in the following example.

   ```
   . . . Double.parseDouble(myInput.next());
   ```

   This form of input will throw a java.lang.NumberFormatException if the value is not an double.

   If you are reading in each double value as a double using nextDouble(), for example

   ```
   . . . myInput.nextDouble();
   ```

   then a java.util.InputMismatchException will be thrown if the value read is not a double. Since an InputMismatchException is a subclass of NoSuchElementException, this exception will be caught in your catch clause for NoSuchElementException but will be reported as an InputMismatchException.

   You can either change your input to use Double.parseDouble(...) or you can catch the java.util.InputMismatchException and handle it the same way you handled the NumberFormatException.
   If you have mixed the two forms of input in your program and you want to keep both, then you will need to catch and handle both of the exceptions.

2. **Testing your main method –** You will need three test methods for VehiclesPart3Test.java: (1) test with a good file name, (2) test with a bad file name, and (3) test with no file name (i.e., the user did not provide a command line argument). In the latter two cases, the vehicleCount should be zero after calling the main method in VehiclesPart3.

3. **General note on test files** – The data files for Part 2 (vehicles1.txt) and Part 3 (vehicles2.txt) have been uploaded in Web-CAT. If you have test methods that read vehicles1.txt, you can retain these and then add new test methods that read vehicles2.txt as needed.

4. **Static count in Vehicles** – If you are incrementing vehicleCount in the Vehicles constructor you may find it necessary to decrement vehicleCount when a NegativeValueException exception is thrown in one of the subclass constructors. The exception causes the constructor to end, and thus the instance is not created. However, since the call to the super constructor incremented vehicleCount before the exception in a subclass, vehicleCount should be decremented in the subclass where the NegativeValueException exception is thrown. To do this you'll need to change vehicleCount from *private* to *protected*.

5. **Skeleton Code (ungraded)** – You can submit to this Web-CAT assignment to check the coverage of your test methods. Just submit your project test files along with your project source files.

6. **Assert.assertArrayEquals** – When the Assert.assertArrayEquals is used in a JUnit test method, it does an element by element compare of the arrays. However, since if elements are objects rather than primitives, the addresses are compared instead of the objects' fields themselves. You are likely get a *false* since the addresses are only equal if the references are aliases. To make the Assert.assertArrayEquals work for arrays of Vehicle, the *equals* method inherited from Object should be overridden in the Vehicle class. The following *equals* method determines equality of two Vehicle objects by comparing the two vehicle owners, yearMakeModel, and value. The hashCode method is required by Checkstyle as a matter of completeness. You can test the *equals* method by creating three Car objects where the first two are equal and the third is not equal. Then assert that car1.equals(car2) is true and assert that car1.equals(car3) is false. To cover the hashCode method, you can assert that car1.hashCode() equals car2.hashCode().

```
/**
 * @param obj the other object
 * @return boolean
 */
public boolean equals(Object obj) {

   if (!(obj instanceof Vehicle)) {
      return false;
   }
   else {
      Vehicle other = (Vehicle) obj;
      return (owner + yearMakeModel + value).
           equals(other.owner + other.yearMakeModel + other.value);
   }
}

/** @return 0 */
public int hashCode() {
   return 0;
}
```