CS100M Spring 2006 Project 6 due Thursday 5/4 at 6pm

Important! Turn off the file backup feature in DrJava—it causes problems on some system configurations! Go to menu item **Edit→Preferences**, choose the last category, "Miscellaneous," then *uncheck* the box for "Keep Emacs-style Backup Files."

Submit your files `Task.java` , `Car.java`, `Truck.java`, `ServiceBay.java` and `AutoRepairShop.java` on-line in CMS under Project 6 before the project deadline. For java code be careful to submit the **.java** file, not the **.class** file. Both correctness and good programming style contribute to your project score.

You must work either on your own or with one partner. You may discuss background issues and general solution strategies with others, but the project you submit must be the work of just you (and your partner). If you work with a partner, you and your partner must register as a group in CMS and submit your work as a group.

Objectives

In this project, you will learn to use one dimensional arrays, inheritance and polymorphism. You will solve a scheduling problem for a service garage (car repair).

Auto Repair Shop

An auto repair shop (garage) has several service bays (places where a car can be serviced). Assume that there are only two types of vehicles: cars and trucks. A service bay may serve cars only, trucks only, or both types. Since cars do not differ too much in size, the bays that can service cars can service any car. However, trucks vary greatly in size, so bays that service trucks have restrictions on truck sizes.

We will assume that each vehicle has a single problem that fits into one of the following "Task" categories (types): MAINTENANCE, REPAIR, DIAGNOSTIC, and STATE INSPECTION. Since some operations need specialized equipment, not all the bays can perform all of the operations. A task can have several states: "waiting" to be serviced, "in progress," and "finished."

Vehicles are brought into your parking lot all day long and, as expected, your general policy is first-come-first-served. However, if the next vehicle in line does not fit an unoccupied bay, you will look further down the queue to find the next vehicle that can be serviced in the unoccupied bay. Assume that only vehicles that are serviceable by your garage are brought to parking lot. After servicing, a vehicle is returned to its original parking spot in the parking lot. This means that your garage can serve only a fixed number of vehicles each day, based on the size of the lot.

You will simulate the running of this garage for a number of time units. Beyond the general description above, additional details and specifications are given in the comments of the class files. Below, we give an overview of how the classes relate to one another, suggest the order in which you complete the classes and how to go about testing, and give hints. Remember to test your code *incrementally*: one class, or even one method, at a time. It is tempting to skip the testing and rush through the code writing, but that will burn more time in the end because you will then have to deal with a large number of confounding bugs! *Incremental and systematic testing will end up saving you time and give a better final product!*

Take a look at the example output in the file `sampleOutput.txt` to get an idea of what the simulation result might look like. You don't have to use an identical output format but your code must provide the same information neatly.

You must follow the specifications and you must not change the provided code (except for any given dummy return statements in the methods that you need to implement). As much as possible, call available methods instead of rewriting the code (method body). You may add extra *private* methods in the given files but not public methods. As usual, we have provided skeleton code that compile correctly, so we expect that your submitted code will at least compile. Submitted code that does not compile will draw a severe penalty.

Overview of the classes

Cars and trucks are both vehicles, so the classes `Car` and `Truck` both extend class `Vehicle`. For our simulation problem, each `Vehicle` has a job that needs to be worked on; class `Task` represents the job. Therefore, one of the fields in a `Vehicle` object is of type `Task`.

A typical repair shop has a number of service bays in which repair jobs (`Tasks`) can be performed. Therefore, an `AutoRepairShop` object has an array of `ServiceBays`. The `AutoRepairShop` also has a parking lot for `Vechicles` that are either waiting for service or have been serviced. This parking lot is represented as an array of `Vehicles` in the `AutoRepairShop` object.

Download the set of skeleton files for Project 6 and compile them. *Read all the given code and comments carefully* **before** *you start to write code!* There are six classes and many relationships to deal with, so get organized! As you read, write on a sheet of paper just the field names and the method signatures and return types. For each class, draw a box to enclose all the members of that class. **The objective here is to have on one page all the operations that are available (and needed) for the entire project**, grouped by the classes. The five minutes it takes to make this info page will pay themselves back many times over when you write the methods (develop algorithms) and constantly need to know which class can do what.

So many classes... so little time

Where to begin? Pick the most independent class—the one that doesn't depend on other classes (but the other classes may depend on it)! The winner is class `Task`. Once you've implemented class `Task`, test it, and then move on to a relatively independent class, such as `Car`. Note that class `Vehicle` is complete. Then work on `Truck`, `ServiceBay`, and finally `AutoRepairShop`. Don't forget to test throughout the development process.

Notes and hints on individual classes

Class `Task`

This class has been implemented partially and you need to complete seven methods. Note the four public constants that you should make use of. All the methods except `toString` should be short. Note that a `Task` is always in one of three mutually exclusive states: waiting, in-progress, or finished. Two fields are used for two of these states so that third state is inferred from the previous two. This means that sometimes you need to set the values in *two* fields in order to change the state.

How do you test the methods in this class independently of the other classes? Create a `Task` object in the `main` method of a test class and then call the individual methods. For example:

```
public class Test{
  public static void main(String[] args) {

    Task t= new Task(3, Task.MAINTENANCE);  // Should compile/run without error if constructor correct
    System.out.println(t.isWaiting()); // Should print true
    System.out.println(t.isInProgress()); // Should print false
    System.out.println(t.isDone()); // Should print false
    System.out.println(t.getDuration()); // Should print 3
    System.out.println(t); // Should print correct values if toString correctly calls the above methods
    t.start();
    System.out.println(t); // Now the status should be in-progress
    for (int i=0; i<3; i++)
      t.perform();
    System.out.println(t); // Now the status should be finished
  }
}
```

Each statement above is a simple test of an individual method or behavior in the class. *Add the tests one at a time to simplify the debugging.* Below is example output from the toString method:

```
    Task: Type: MAINTENANCE
    Duration: 3 Status: in progress; Time remaining: 3
```

Where is a good place to find data for testing? Look at the given `main` method in class `AutoRepairShop`. The above example case `new Task(3, Task.MAINTENANCE)` was copied from `main` in `AutoRepairShop`.

Class `Vehicle`

This class has been implemented completely. Do not change the code in this class but read it carefully.

Classes `Car` and `Truck`

Both of these classes are children of class `Vehicle` and are partially implemented. Complete and test these classes one at a time. All the methods should be quite short. Add more code to the `main` method of your test class: create a `Car` object and call its methods individually and make sure that the output reflects the correct behavior for the `Car` class. Then implement the `Truck` class and do more tests. Again, you can easily copy some data from `main` in `AutoRepairShop` for testing. The `acceptableAtBay` method can't be tested until you implement the `ServiceBay` class. Example output from the `toString` methods of `Car` and `Truck` are shown below. Notice that these `toString` methods make use of `toString` from the `Task` class.

```
Car
Subaru  Licence Plate: CRD-1587  Body style: 4D Sedan
Task: Type: MAINTENANCE
Duration: 3 Status: waiting; Time remaining: 3

Truck
VOLVO  Licence Plate: CD-56566  Height: 13 Weight: 10
Task: Type: REPAIR
Duration: 80 Status: waiting; Time remaining: 80
```

Class `ServiceBay`

All but two of the methods have been completed for this class. Read the entire class carefully. Notice that the allowed `Task` types for a `ServiceBay` is stored in an array of `int`s; the allowed `Vehicle` types for a `ServiceBay` is stored in an array of `String`s. The two methods you implement will work with these arrays.

Again, you can copy some code from `main` in `AutoRepairShop` to `main` in your test class. Create at least one `ServiceBay` object and test the methods you have written. You will need to use the `Task`, `Car`, and `Truck` objects created in the previous testing steps. Be sure to test the `acceptableAtBay` methods from the `Car` and `Truck` classes as well. Example output from the `toString` method of `ServiceBay` is shown below. Two cases are shown: idle bay and a `Truck` in the bay. Again, this `toString` makes use of `toString` from the other implemented classes.

```
Bay:
Allowed vehicle types: Car Truck
Allowed tasks: DIAGNOSTIC MAINTENANCE STATE INSPECTION
Max height: 15 Max weight: 20
Working on: Idle

Bay:
Allowed vehicle types: Car Truck
Allowed tasks: DIAGNOSTIC MAINTENANCE STATE INSPECTION
Max height: 15 Max weight: 20
Working on: Truck
VOLVO  Licence Plate: CD-56566  Height: 13 Weight: 10
Task: Type: REPAIR
Duration: 80 Status: waiting; Time remaining: 80
```

Class `AutoRepairShop`

This class has been implemented partially and you need to implement four methods and answer two questions. The code in the methods of this class is more complicated than that in the other classes since this class interacts with *all* the other classes directly. For example, one may need to "perform" a `Task` on a `Vehicle` in a `ServiceBay`. The corresponding statement may look like the following:

```
this.bays[i].getCurrentVehicle().getJob().perform();
```

where `i` is a valid index in the `bays` array. Notice how the method calls are chained up? `this.bays[i]` is a `ServiceBay`, which has a `getCurrentVehicle` method; the `getCurrentVehicle` method returns a `Vehicle`, which has a `getJob` method; the `getJob` method returns a `Task`, which has the `perform` method that we need to call ultimately. This is where that info page you prepared at the start of the project will keep on giving!

See `sampleOutput.txt` for the format of method `toString`. There are a few other considerations to keep in mind as you implement the `AutoRepairShop` class:

- The "parking lot" has both unserviced and serviced `Vehicles`. Make sure that you schedule only the unserviced ones in a `ServiceBay`.
- The component `originalPositions[i]`, corresponding to `bays[i]`, is initialized to the bogus position `-1` to match the case that no `Vehicle` is currently in `bays[i]`. It may be useful to keep this convention: set `originalPositions[i]` to `-1` when `bays[i]` becomes empty.
- You may want to write the `static nextArrival` method fairly early so that the method can be called to simplfy the testing of the class. The method specification stipulates some random behavior, but random behavior makes testing difficult, so what should you do? *Write a temporary, simple method body that is "deterministic,"* i.e., has a fixed (not random) behavior. For example, have the method always return the next `Vehicle` in the array. Then after you have implemented and tested the rest of the class, change the method body to reflect the actual specification.
- Somewhere in this class (in the parts that you are allowed to change) you need to increment the system time! What is the most logical place to do this?

Don't forget to answer the two questions in methods `answerQ1` and `answerQ2`. Be concise.

Final words... This "scheduling problem" is a well known and well studied problem in computer science and operations research. In computer science, the problem is often studied in the context of assigning jobs to a set of available processors in an "optimal" way. Many other industrial processes and some environmental systems can be modeled as a scheduling problem. Unfortunately, the optimal solution cannot be found easily or efficiently—the greedy algorithm we use in this project is easy to implement but is far from optimal. However, there are a number of established methods for achieving reasonable solutions. You can explore these ideas and more in other computer science courses, e.g., CS482 Introduction to Analysis of Algorithms and CS681 Analysis of Algorithms.