Worksheet 9: Concrete Inheritance & Interfaces

Updated: 19th May, 2016

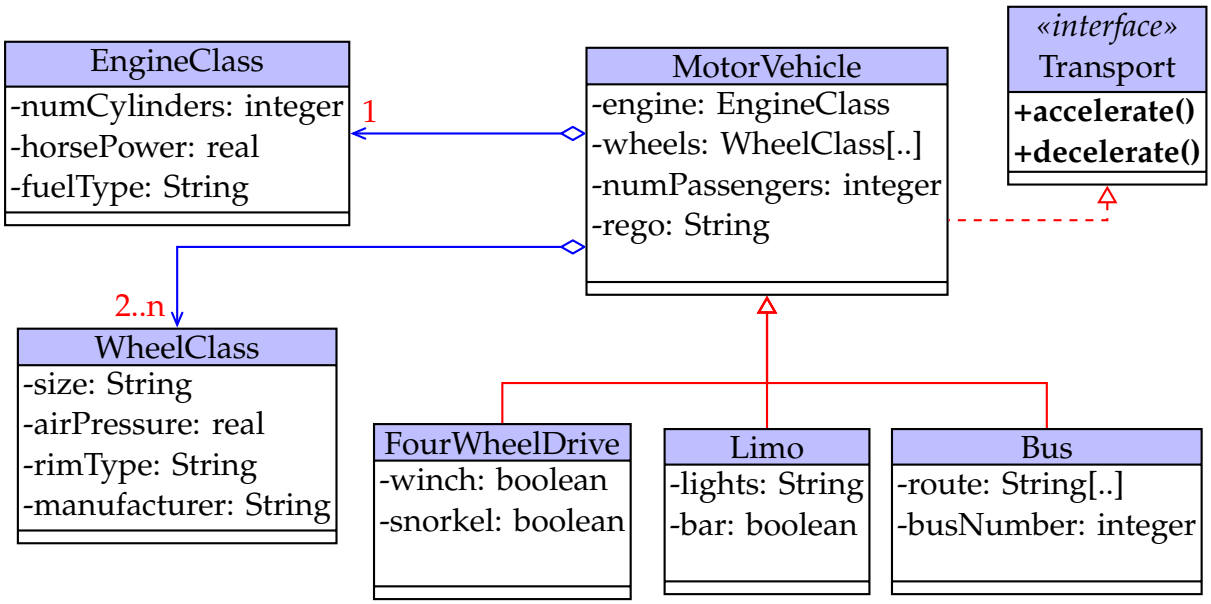
The objectives of this practical are:

- Implement several subclasses and understand the inheritance relationship with a concrete super class.
- Implement a single interface and understand the relationship between the interface, superclass, and subclass.
- Use an array of interfaces in an algorithm.
- Deal with run-time polymorphism decisions.

Note: You should copy all the files in your P07 directory (ie: last weeks work) and place them in your P09 directory.

In other words make a back up of your classes **before** making any changes.

In last weeks practical you developed and implemented EngineClass and MotorVehicle, now we need the ability to model attributes unique to different types of motor vehicles. The class diagram below adds classes for Limo, Bus, and FourWheelDrive.



1. Woops! We made a mistake!

In last weeks practical the valid values for fuel type were as follows.

- "ULP"
- "PULP"
- "DIESEL"
- "LPG"
- "98RON"

While refining the design we discovered the final use of the classes will require the full names of the fuel types, not abbreviations. Hence, you need to change your EngineClass to use the following values.

- "UNLEADED PETROL"
- "PREMIUM UNLEADED PETROL"
- "DIESEL"
- "LIQUEFIED PETROLEUM GAS"
- "PREMIUM UNLEADED PETROL (OCTANE RATING 98)"

Note: If you used class constants this change will only take a minute, if you hard coded all the fuel types this change will take significantly longer to implement.

2. Limo Class

The first subclass you need to implement is Limo, a type of MotorVehicle.

A Limo may or may not have a bar (boolean), and can have a variety of lighting options (a descriptive string).

Two limos are the same if and only if they have the same engine, the same set of wheels, the same number of passengers, and both have (or lack) a bar.

Your class must have all the constructors, accessors, and mutators emphasised in the lecture and last weeks practical.

Remember to construct a test harness for your class!

3. Bus Class

The second subclass you need to implement is Bus, a type of MotorVehicle.

A Bus has a route (array of Strings, with at least 1 element), and bus number.

Two buses are the same if and only if they have the same engine, the same set of wheels, the same number of passengers, and the same bus number.

Your class must have all the constructors, accessors, and mutators emphasised in the lecture and last weeks practical.

Remember to construct a test harness for your class!

4. Transport

Note: You will need to download the Transport.java interface from blackboard and place it in your current working directory. This interface will need to be compiled.

Why do we need a Transport interface?
Because any thing capable of transportation (eg: Bus, Bike, Plane, Ship etc. . .) must be able to *accelerate* and *decelerate*, even if *how* they do so is different.

To do this you need to include velocity in your classes. This field should be added to MotorVehicle, velocity must be greater than or equal to 0.0 and in no circumstance exceed 220.0, a motor vehicle starts with a velocity of 0.0, and has no impact on equality.

You will need to add accessors and mutators for velocity, and include it in your toString method.

Once velocity has been added to MotorVehicle you can add the accelerate and decelerate methods to the subclasses:

- accelerate
  - Takes no import
  - Has no export
  - Increases velocity according to the appropriate formula
  - If the change would make velocity exceed the maximum then set it to the maximum.
  - **This method should NOT fail**
- decelerate
  - Takes no import
  - Has no export
  - Decreases velocity according to the appropriate formula
  - If the change would make velocity less than 0 then set it to 0
  - **This method should NOT fail**

Class	Max Velocity	Accelerate	Decelerate
MotorV.	220.0	$numCylinders * 0.75$	$\frac{42}{numberOfWheels} + \frac{1}{numPassengers}$
Bus	110.0	$((numCylindersMOD4) + 1) * 1.5$	$\frac{42}{numberOfWheels} + \frac{1}{numPassengers}$
Limo	150.0	numCylinders	$\frac{42}{numberOfWheels} + \frac{1}{numPassengers}$
4WD	220.0	$\frac{horsePower}{(numCylinders * 10.0)} * 1.75$	$\frac{42}{numberOfWheels} + \frac{1}{numPassengers}$

Remember to add the new methods to your test harnesses (this means recompiling and executing as well)!

5. Algorithm Design Using Object Orientation

You now have the ability to model Limo's and Buses, with the ability to accelerate and decelerate. Let's create a **simple** racing game, we'll call it `SimpleRacer`.

The user will need to provide the details of up to 5 vehicles, any combination of Limo and Bus, you will need to use a *single* array to store these objects.

When the program starts the user should be presented with the following menu.

```
Main Menu:
1 Add Vehicle
2 Race
3 Exit
```

When `Add Vehicle` is selected the user should encounter the following menu.

```
Add Vehicle:
1 Add Limo
2 Add Bus
3 Return to Main Menu
```

From here they can choose to enter either a Limo or Bus into the race. Based on the selection they should be prompted for the appropriate data **see the note on how to simplify this process**.

When the `Race` option is selected you will need to determine the results of the "race".

Each vehicles' result is calculated by determining how many times the accelerate method must be called until the *max speed* is reached.

The results should be displayed in the format of "<type>: <rego>: <score>", the menu should also be redisplayed. For example:

```
Race Results:
Limo: BSN-801: 5
Bus: BUN-2011: 4
Limo: FN-2187: 3
Limo: CT-7567: 4
Bus: TR-8R: 8

Main Menu:
1 Add Vehicle
2 Race
3 Exit
```

In order to implement this functionality you will need to keep track of an array of objects, and how many elements of the array are currently being used. You should implement another class `Race` that will track this information, handle adding a vehicle to the race, and have a method called `startRace` that is responsible for calculating how many times each vehicle must accelerate. This is *not* a container class so you can implement user input and output in this class, technically, this type of class is called a *"controller"*.

Note: Things to consider.

- How can you use a single array to store both Limo and Bus class objects?
- How is the user going to indicate the type of vehicle?
- How can you make sure you are using the correct maximum speed for each vehicle (not just the default)?
- Could you print out a message saying who the winner is?

Simplify your data entry! None of accelerate methods are based on wheel data, you may assume that **all** the vehicles have 6 default wheels.

A note on Menus: Your menu should be implemented in a separate class to your *main*.

Arrays: remember you should be catching ArrayIndexOutOfBounds exceptions.

ARE YOU STILL STRUGGLING: If you are still struggling to understand these concepts then go ahead and implement FourWheelDrive.

A FourWheelDrive may or may not have traction control (a boolean), and may or may not have a winch (a boolean).

Two FourWheelDrives are considered equal if and only if they have the same same engine, the same set of wheels, the same number of passengers, have (or lack) a winch, and have (or lack) a snorkel.

Remember to implement a test harness for this class.

You can then modify SimpleRacer to also include options for a FourWheelDrive.

End of Worksheet