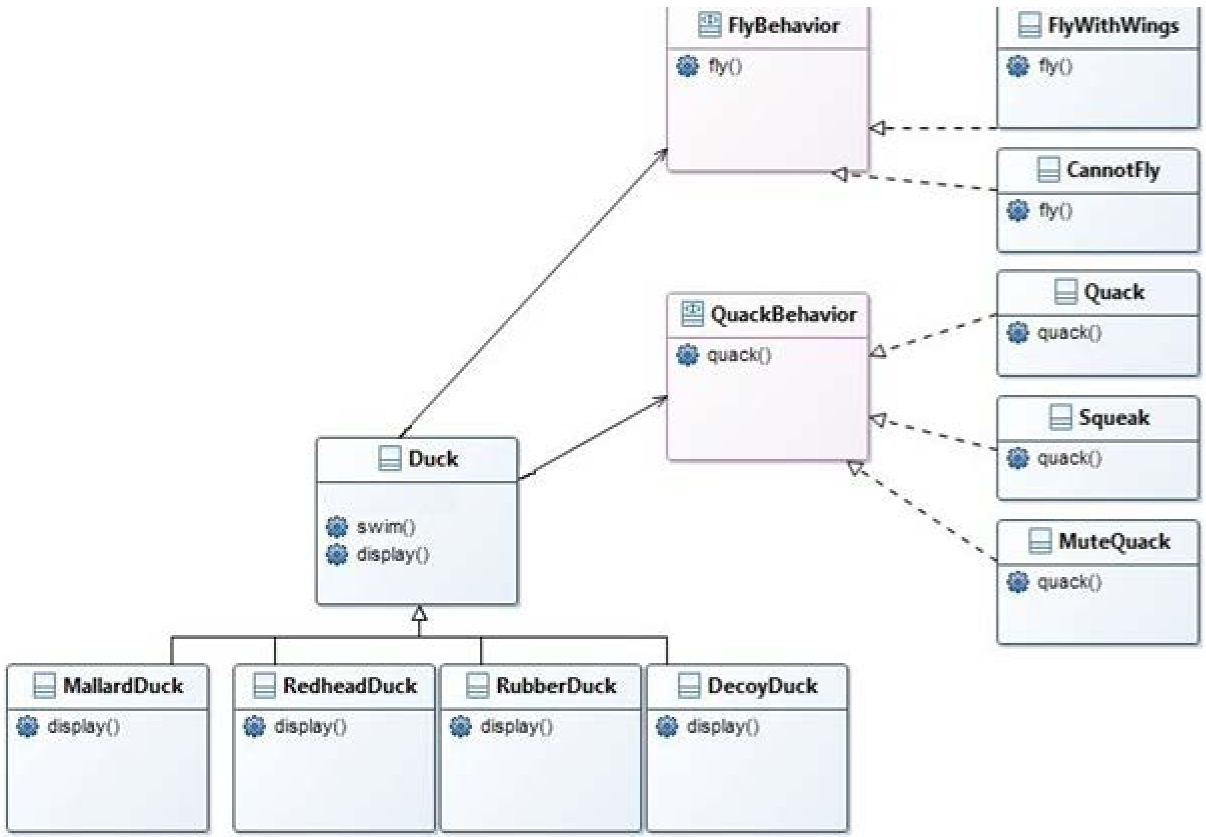
Lab 5

1. A class diagram for the DuckApp, produced below. How will the Duck class use the FlyBehavior and QuackBehavior interfaces? Implement the diagram in Java, and make sure the answer to this question is clear in your code. To implement the methods like fly() and quack(), just print a phrase to the console, like “Flying with wings” or “Quack by squeaking.”



To test your code, create a Main class like the following:

```java
public class Main {
    public static void main(String[] args) {
        Duck[] ducks =
            {new MallardDuck(), new DecoyDuck(), new RedheadDuck(), new RubberDuck()};
        for(Duck d: ducks) {
            System.out.println(d.getClass().getSimpleName() + ":");
            d.display();
            d.fly();
            d.quack();
            d.swim();
        }
    }
}
```

Output should look like this:

```
MallardDuck:
  display
  fly with wings
  quacking
  swimming
DecoyDuck:
  displaying
  cannot fly
  cannot quack
  swimming
RedheadDuck:
  displaying
  fly with wings
  quacking
  swimming
RubberDuck:
  displaying
  cannot fly
  squeaking
  swimming
```

2. Create Java classes for `Triangle, Rectangle,` and `Circle` which implements Figure Interface. The figure has one method.

```
public double computeArea()
```

Make all of these classes <u>immutable.</u> (Follow the guidelines in the slides for creating this type of class – included with this lab at end of this problem.) Provide one constructor for each class; the constructor should accept the data necessary to specify the figure, and to compute its area. The values accepted by the constructor should be stored in (private) instance fields of the class. For example, `Rectangle` should have instance fields `width` and `length`, and the constructor should look like this

```
public Rectangle(double width, double length)
```

For `Triangle`, you may use arguments `base` and `height`. And for `Circle`, use `radius` as the constructor argument.

Whenever you create instance fields for one of these classes, provide public accessors for them (but do not provide mutators since the class is supposed to be immutable – for instance, the dimensions of a `Rectangle` should be read-only). For example, you will have in the `Rectangle` class:

```
private double width;
public double getWidth() {
    return width;
}
```

Create a fourth class `Main` that will, in its `main` method, create multiple instances of these figures (you may invent your own input values), store them in a single list, and then *polymorphically* compute and write to the console the sum of the areas.

Typical output:

```
Sum of Areas = 37.38
```

Draw a class diagram that includes all the types mentioned in this problem.

Here are some area formulas:

```
Area of a rectangle  =  width * height

Area of a triangle = 1/2 * base * height

Area of a circle = PI * radius * radius
```

```
Notes :
```

Making Your Classes Immutable

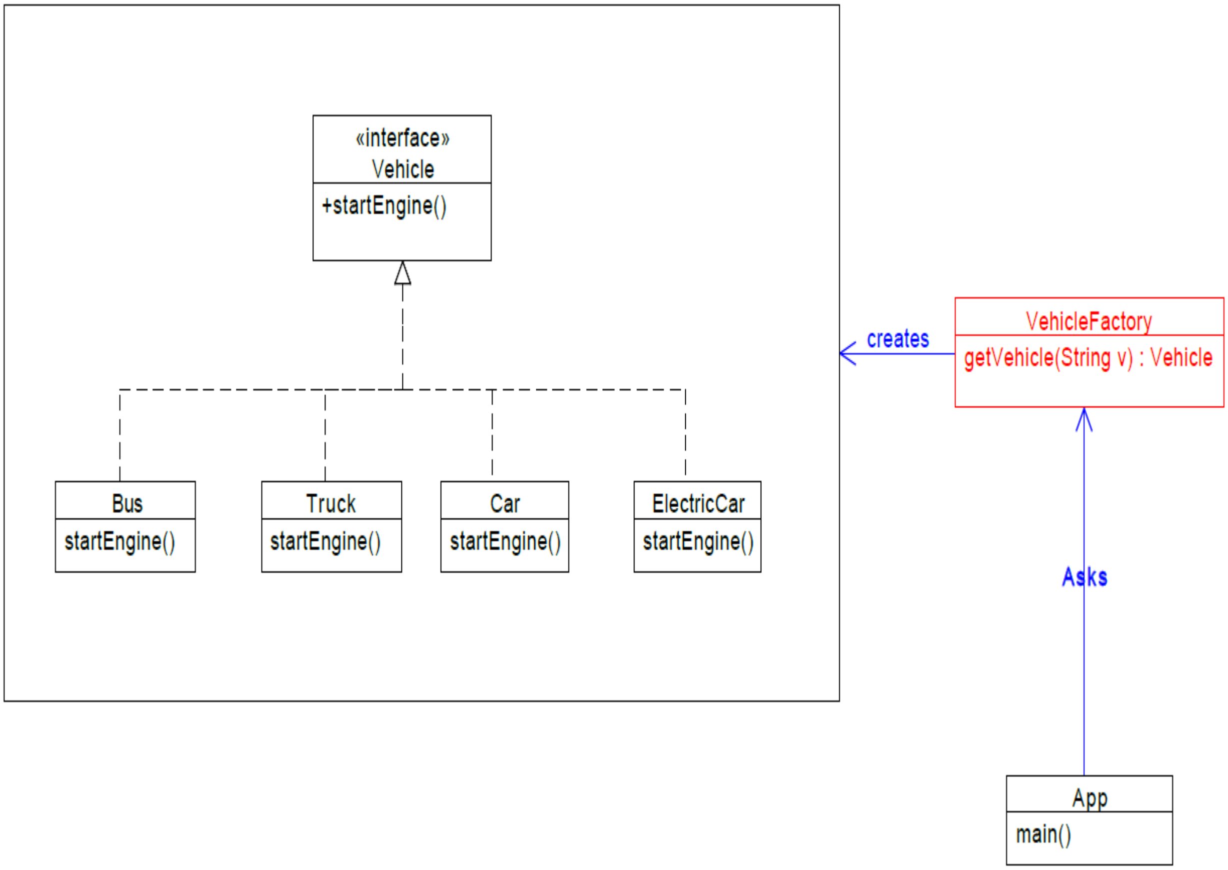
1. A class is immutable if the data it stores cannot be modified once it is initialized. Java's String and number classes (such as Integer, Double, BigInteger) are immutable. Immutable classes provide good building blocks for creating more complex objects. Java 8: LocalDate, as we saw earlier, is also immutable.

2. Immutable classes tend to be smaller and focused (building blocks for more complex behavior). If many instances are needed, a "mutable companion" should also be created (for example, the mutable companion for String is StringBuilder) to handle the multiplicity without hindering performance.

3. Guidelines for creating an immutable class (from *Effective Java*, 2nd ed.)

    - **All fields should be *private* and *final*.** This keeps internals private and prevents data from changing once the object is created.
    - **Provide *getters* but no *setters* for all fields.** Not providing setters is essential for making the class immutable.
    - **Make the class *final*.** (This prevents users of the class from accessing the internals of the class in another way – to be discussed in Lesson 6.)
    - **Make sure that getters do not return mutable objects.**

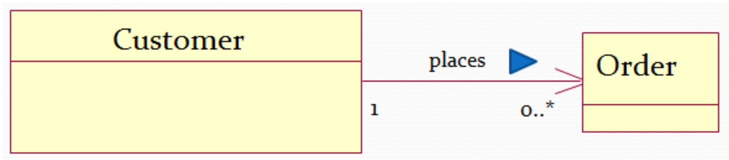
3. Do implementation for the given UML diagram using Factory Pattern. Just display a simple message inside startEngine() method to know about the specific type of Vehicle.



4. In Lesson 2, one way of maintaining a unidirectional one-many relationship was mentioned – in that approach, creation of secondary objects (in the example, these were of type Order) was controlled by limiting visibility of the constructor to *package* level. For this problem, modify the code mentioned there (which can be found in the code folder for Lab 5) so that construction of Order objects is controlled by a factory method. The guidelines given in Lesson 2 for maintaining a one-many unidirectional association are reproduced here:

One-many Multiplicity:
- Associated with each Customer, there are zero or more Orders
- A Customer object maintains a collection of Order objects.
- Associated with each Order, there is exactly one Customer
- It is possible to navigate from a Customer to any of his Orders, but not from an Order to the owning Customer.
- *Maintaining the relationship* means:
    - when new Customer object is created, it is equipped with a (possibly empty) collection of Orders
    - it is not possible to create an Order object independent of a Customer; each new Order object must belong to the collection of Orders for some Customer.

Note: An example of using a factory method to maintain an association relationship was given in the Lesson 5 slides (see lesson5.lecture.factorymethods6 for the code showing a factory method, and see lesson2.lecture.unidirectional.onemany to see another way to code a one-many unidirectional relationship).



In your implementation, make sure that Customer, Order, and Item all belong to the same package and that the only way any of these classes can be instantiated is by using a factory method in a factory class (which you may wish to name as CustOrderFactory).