Polymorphism

In this recitation, instead of just trying to solve some problems, we will introduce concepts closely related to what we covered in the lectures under the topic of polymorphism and then see how these concepts affect some OOP (object-oriented programming) codes. These exercises are meant to be studied and discussed, and not to be just treated as individual problem-solving sessions. We have seen that in a type system, objects can be viewed as more general or more specific. E.g., if a class **`Sub`** extends another class **`Super`**, the former is a more specific category than the latter, exhibiting the "is-a" relationship. In Java's type system, **`java.lang.Object`** is the most general object, and every other object is its subtype. This may lead to **runtime polymorphism**. A simple example is the following:

```
class Vehicle {
  String name;

  public Vehicle(String name) { this.name = name; }

  public void start() {
    System.out.println("Starting vehicle " + this.name + ".";
  }

  public void drive() {
    System.out.println("Driving vehicle " + this.name + ".";
  }

  public void stop() {
    System.out.println("Stopping vehicle " + this.name + ".";
  }
}
// continued on page 2
class ElectricVehicle {
  double charge = 0.0; // percentage of battery charged, between 0 and 100

  public ElectricVehicle(String name) {
    super(name);
    this.charge = 100; // new instances are fully charged
  }

  public void start() {
    System.out.println("Starting electric vehicle " + name + ".";
  }

  public void drive() {
    System.out.println("Driving electric vehicle " + name + ".";
  }

  public void stop() {
    System.out.println("Stopping electric vehicle " + name + ".";
```

```
  }

  public double getCharge() { return this.charge; }

  public static void main(String... args) {
    Vehicle v1 = new Vehicle("Truck");
    ElectricVehicle v2 = new ElectricVehicle("Tesla");
    Vehicle v3 = new ElectricVehicle("Leaf");
    v1.start();
    v2.start();
    v3.start();
    v2.getCharge();
    v3.getCharge();
  }
}
```

In this code, each of the three instances have a runtime type and a compile-time type. For v1 and v2, the runtime and compile-time types are the same. For v3, however, they are different.

1. Will the above code compile? If no, which line(s) should be removed to make it compile? (the commenting should be minimal, because otherwise we could just comment out everything and run an empty code that does nothing)
2. What was wrong with the commented line(s), and why?
3. What are the printed outputs of the code once you comment the problematic line(s)?
4. We have also seen that single inheritance can be overly restrictive, but it has the advantage of avoiding the "diamond problem". We have also seen that Java provides the mechanism of interfaces – where multiple inheritance is supported – to avoid this restriction. Design a codebase to model an electric vehicle as a subtype of a type **`Vehicle`** and also another type **`Chargeable`**, which will be responsible for modeling the existence of a battery and the functionalities regarding its charging. Based on that, also create a class called **`SmartPhone`**.
   This part of the question is intentionally vague, and it's your job to figure out how to model the subtypes within the constraints of Java. The key to this design is to understand when to extend a class and when to implement an interface. Your code need not be very detailed (like the code on the first page of this document, one or two methods for each type is sufficient).

Figuring out the types gets a bit more complicated when there is a mix of parametric polymorphism and subtype polymorphism. For example,

```
class Vehicle {
  public void service() { System.out.println("Generic vehicle servicing"); }
}

class Bike extends Vehicle {
  @Override
  public void service() { System.out.println("Bike servicing"); }
}

class Truck extends Vehicle {
  @Override
```

```
  public void service() { System.out.println("Truck servicing"); }
}

class Mechanic {
  public void serviceVehicles(List<Vehicle> vehicles) {
    for (Vehicle v: vehicles) v.service();
  }

  public static void main(String[] args) {
    // what kind of polymorphism do we see here? try to find the variable and
    // the different types it exhibits.
    List<Vehicle> vehicles = new ArrayList<Vehicle>();
    vehicles.add(new Vehicle());
    vehicles.add(new Vehicle());
    List<Bike> bikes = new ArrayList<Bike>();
    bikes.add(new Bike());
    bikes.add(new Bike());
    List<Truck> trucks = new ArrayList<Truck>();
    trucks.add(new Truck());
    trucks.add(new Truck());

    // will the following code work? why or why not? try to understand what
    // works in the following lines and what doesn't.
    Mechanic mechanic = new Mechanic();
    mechanic.serviceVehicles(vehicles);
    mechanic.serviceVehicles(bikes);.
    mechanic.serviceVehicles(cars);
  }
}
```

Try to answer the questions in the above code. It's ok if you can't figure it out right away. I would urge you to write the code in your Java IDE and see for yourself what happens. The error messages will be meaningful, and you should spend some time connecting the lecture material to what you see with the code here.