1. Ferrari and Audi extend the abstract notion of a Car. For a Ferrari int getMaxSpeed() returns 175 and for Audi it returns 150. Different options are available which affects the maximum speed. For instance the Turbo option adds 10 and the Spyder (convertible) option subtracts 10. The following uses the Decorator Pattern:

```java
Car f1 = new Ferrari();
Car f2 = new Turbo(f1);
Car f3 = new Spyder(f2); // now we have a Turbo Spyder Ferrari
```

1a) [5%] what does f3.getMaxSpeed() return? Explain your answer.

1b) [5%] After the above three lines of java are executed, is the instance that f2 references now a convertible (instanceof Spyder)? Explain.

1c) [15%] Draw a UML class diagram of the above (careful: the Decorator Pattern design requires at least six classes/interfaces) and make sure you show how the parts interrelate.
1d) [10%] Finish the implementation using the Decorator Pattern:

```java
abstract public class Car extends Observable {
    abstract public int getMaxSpeed();
}

public class Ferrari extends Car {

abstract public class Decorator {

    public Decorator()
}

public class Turbo
    public Turbo(Car c) {
        public int getMaxSpeed() {

            public class Spyder
                public Spyder(Car c) {
                    public int getMaxSpeed() {

1e) [10%] What’s an alternative to using the Decorator Pattern that would allow creating all these combinations of variations (e.g., a Ferrari with both Turbo and Spyder options, or an Audi with only Turbo)? What if we also wanted to add some additional options in the future that further affected the maximum speed (e.g., Nitro or, Hybrid) and apply that to all the various types of cars? Is there any advantage or disadvantage to this method?
2) Cars use three forms of **EngineStrategy**: Electric, Gas, and Hybrid. The **Strategy Pattern** will be used to implement the String method `getPowerSource()`, which returns “electricity”, “petrol”, or “hybrid”. For example,

```java
Car c = new Ferrari();
System.err.println(c.getEngineType()); // prints “petrol”.
```

2a) [10\%] While the entire hierarchy consists of **EngineStrategy**, Electric, Gas, and Hybrid, just write **EngineStrategy** and **Gas**:

```java
public abstract class EngineStrategy {

    public class Gas

}
```

2b) [15\%] While adding this method to the hierarchy affects many parts of the code, only indicate the additional lines of code needed for **Car** and **Ferrari** to use the Strategy Pattern so that the above code would work. Be specific about where the changes would be added (or add them to the earlier code).

```java
abstract public class Car {
    protected EnergyStrategy energyStrategy;
    
    abstract public int getMaxSpeed();

    public String getEnergyStrategy() {
        energyStrategy.getEngineType();
    }
}
```

```java
public class Ferrari extends Car {
```
3.) For the Observer Pattern, we’ll use Java’s Observer and Observable.java. 
Presume that Car implements Vehicle and extends Observable:

```java
public interface Vehicle {
    public void setSpeed(int speed);      // adjust this Car’s speed
    public int  getSpeed();               // returns its current speed
    public void receiveSpeedingTicket();  // bad news
}
```

Here is the following driver code (the observe method is discussed later):

```java
PoliceCar p = new PoliceCar();  // PoliceCar extends Car.
Ferrari   f = new Ferrari();

p.observe(f);   // this is what Poooooolice do, they observe others
```

3a) [5%] Whenever the Ferrari’s setSpeed method is called, any observers (including the police or those gawking kids in the van in the slow lane) are notified. Finish setSpeed (just deal with the Observer Pattern aspects):

```java
public void setSpeed(int speed) {
    currentSpeed = speed;  // store the new speed
}
```

3b) [5%] PoliceCar is a subclass of Car. Finish writing observe (hint: remember all Cars are Observable):

```java
public void observe(Car c) {
```

3c) [10%] For any Car c observed by PoliceCar p, whenever c changes speed (by setSpeed), p will check(by getSpeed) and if it is faster than 65 (MPH), c will receiveSpeedingTicket. Add this functionality to PoliceCar (first decide what method you need to write):

```java
public void
3d) [10%] Complete the **UML Sequence Diagram** for the following, assuming p is a PoliceCar and f is a Ferrari.

```java
    p.observe(f);
    f.setSpeed(100);
```