There are two kinds of eating behavior or strategy: Herbivorous and Carnivorous. The corresponding two eat() methods return “I’m nibbling leaves” and “I’m eating flesh” respectively.

```java
public interface EatingBehavior {
    public String eat();
}

public class HerbivorousEating implements EatingBehavior {
    public String eat() { return "I'm nibbling leaves"; }
}

public class CarnivorousEating implements EatingBehavior {
    public String eat() { return "I'm eating flesh"; }
}
```

Lions observe Deer and Deer observe Lions. All animals have a move() method. The move method prints (by System.out.println) a String saying “I am a lion moving” or “I am a Deer moving”. But, also, whenever a Lion moves() it causes any Deer that is observing that Lion to move(). (But if a Lion moves it doesn’t cause other Lions to move. Similarly, if a Deer moves, any Lion that is observing the Deer will “notice” it and move() as a result (but one Deer moving doesn’t cause another Deer to move).

1) [10%] First, complete the driver code (adding what it takes to get bambi and leo to observe one another (hint: use addObserver).

```java
Deer bambi = new Deer();
Lion leo = new Lion();
// add code below to get them to observe each other:

bambi.addObserver(leo);
leo.addObserver(bambi);

bambi.move();
```

1) [10%] What would be the SOP output? Give a few lines of output that results from bambi.move() and explain your answer (you could use a sequence diagram here to very efficiently explain what happens). Does it terminate?

```
“I am a Deer moving”
“I am a lion moving”
“I am a Deer moving”
“I am a lion moving”… (until stack overflow)
```
3) (80%) finish writing **Animal** and the subclasses **Deer** and **Lion** using the **Strategy Pattern** for their `eat()` method and the Observer pattern for the above effect of one animal causing another to move. Remember the Java interface Observer is:

```java
public interface Observer { public void update(Observable o, Object obj); }
```

And remember that any Observable needs to setChanged() and notifyObservers() whenever it wishes to have its observers notified.

```java
abstract public class Animal extends Observable implements Observer {
    EatingBehavior eb;

    abstract public void update(Observable o, Object obj);
    public void eat() { System.out.println(eb.eat()); }
    abstract public void move();
}
```

```java
public class Deer extends Animal {
    public Deer() {
        eb = new HerbivorousEating();
    }

    public void move() {
        System.out.println("I am a Deer moving");
        setChanged();
        notifyObservers();
    }

    public void update(Observable o, Object obj) {
        if (o instanceof Lion)
            move();
    }
}
```

```java
public class Lion extends Animal {
    public Lion() { eb = new CarnivorousEating(); }

    public void move() {
        System.out.println("I am a Lion moving");
        setChanged();
        notifyObservers();
    }

    public void update(Observable o, Object obj) {
        if (o instanceof Deer)
            move();
    }
}
```