1. **Composition and other constructors** Implement the following constructor of `Ball` so that it uses composition instead of aggregation:

```
/**
 * Initialize the ball so that its position and velocity are
 * equal to the given position and velocity.
 *
 * @param position
 *            the position of the ball
 * @param velocity
 *            the velocity of the ball
 */
public Ball(Point2 position, Vector2 velocity) {

    this.position =

    this.velocity =

}
```

2. **Composition and the copy constructor**

   (a) Assume that `Ball` has the constructor from Question 1. Suppose `Ball` had the following copy constructor:

```
/**
 * Initialize the ball so that its position and velocity are
 * equal to the position and velocity of another ball.
 *
 * @param other
 *            the ball to copy
 */
public Ball(Ball other) {
    this.position = other.position;
    this.velocity = other.velocity;
}
```

   What would the following code fragment print? How many `Point2` objects are created? How many `Vector2` objects are created?

```
Point2 p = new Point2();
Vector2 v = new Vector2();
Ball b1 = new Ball(p, v);
Ball b2 = new Ball(b1);
p.setX(-100.0);
b1.setPosition(p);
System.out.println(b2.getPosition());
```

   (b) Implement the copy constructor of `Ball` so that it uses composition:

```
public Ball(Ball other) {

    this.position =

    this.velocity =
```

```
}
```

(c) If `Point2` and `Vector2` were immutable, would the copy constructor from Question 2a be acceptable? Why?

3. **Composition and accessor methods**

(a) Suppose `Ball` had the following accessor method:

```
public Vector2 getVelocity() {
    return this.velocity;
}
```

What does the following code fragment print?

```
Ball b = new Ball(new Point2(), new Vector2());
Vector2 v = b.getVelocity();
v.set(-1000.0, 500.0);
System.out.println(b.getVelocity());
```

(b) Implement the following accessor methods so that they use composition:

```
public Point2 getPosition() {

    return

}

public Vector2 getVelocity() {

    return

}
```

(c) Re-implement the copy constructor so that it uses the above accessor methods:

```
public Ball(Ball other) {

    this.position =

    this.velocity =

}
```

4. **Composition and mutator methods**

   (a) Suppose `Ball` had the following mutator method:

```
public void setVelocity(Vector2 velocity) {
    this.velocity = velocity;
}
```

   What does the following code fragment print?

```
Ball b = new Ball(new Point2(), new Vector2());
Vector2 v = new Vector2(100.0, 200.0);
b.setVelocity(v);
v.set(-1.0, -5.0);
System.out.println(b.getVelocity());
```

   (b) Implement the following mutator methods so that they use composition:

```
public void setPosition(Point2 position) {

    this.position =

}

public void setVelocity(Vector2 velocity) {

    this.velocity =

}
```

   (c) Re-implement the following constructor so that it uses the above mutator methods:

```
public Ball(Point2 position, Vector2 velocity) {

    this.position =

    this.velocity =

}
```

5. **Composition, class invariants, and constructors**

   The class `java.util.Date` has a method named `getTime` that returns the number of milliseconds since January 1, 1970, 00:00:00 GMT as a `long` value. It also has a method with signature `setTime(long)` that sets the date using the number of milliseconds since January 1, 1970, 00:00:00 GMT. It also has a constructor with signature `Date(long)` that initializes the date using the number of milliseconds since January 1, 1970, 00:00:00 GMT. It has no copy constructor; to copy another `Date` instance, you need to use `getTime`:

```
Date d = new Date();
Date dCopy = new Date(d.getTime());
```

(a) Suppose that `Period` has the constructor shown in the lecture slides. Add one more line of code to show how the client can break the class invariant:

```
Date start = new Date();
// note: Date has no copy constructor
Date end = new Date( start.getTime() + 10000 );
Period p = new Period( start, end );
```

(b) Fix the constructor so that it maintains the class invariant.

```
public Period(Date start, Date end) {
    if (start.compareTo(end) > 0) {
        throw new IllegalArgumentException("start after end");
    }

    this.start =

    this.end =

}
```

6. **Composition, class invariants, and the copy constructor**

(a) What does the following code fragment print?

```
Date start = new Date();
Date end = new Date( start.getTime() + 10000 );
Period p1 = new Period( start, end );
Period p2 = new Period( p1 );
System.out.println( p1.getStart() == p2.getStart() );
System.out.println( p1.getEnd() == p2.getEnd() );
```

(b) Fix the copy constructor so that it maintains the class invariant.

```
public Period(Period other) {

    this.start =

    this.end =

}
```

7. **Composition, class invariants, and accessor methods**

(a) Suppose that `Period` has the `getStart` and `getEnd` methods shown in the lecture slides. Add one more line of code using either `getStart` or `getEnd` to show how the client can break the class invariant:

```
Date start = new Date();
// note: Date has no copy constructor
Date end = new Date( start.getTime() + 10000 );
```

```
Period p = new Period( start, end );
```

(b) Fix the accessors so that they maintain the class invariant.

```
public Date getStart() {

    return

}

public Date getEnd() {

    return

}
```

8. **Composition, class invariants, and mutator methods**

(a) Suppose that `Period` has the `setStart` method shown in the lecture slides. Add one more line of code to show how the client can break the class invariant:

```
Date start = new Date();
Date end = new Date( start.getTime() + 10000 );
Period p = new Period( start, end );
p.setStart( start );
```

(b) Fix the mutators so that they maintain the class invariant.

```
public boolean setStart(Date newStart) {
    boolean ok = false;
    if (newStart.compareTo(this.end) < 0) {

        this.start =

        ok = true;
    }
    return ok;
}
```

9. **Collections as fields**

(a) What does the following code fragment print?

```
ArrayList<Point> pts = new ArrayList<Point2>();
Point2 p = new Point2(0., 0.);
pts.add(p);
p.setX( 10.0 );
```

```
System.out.println(p);
System.out.println(pts.get(0));
```

(b) Is `ArrayList<Point2>` an aggregation or composition of points?

10. **Collections as fields: Aliasing**

Suppose that the copy constructor of `Firework` is implemented using aliasing.

Assume that `f1` is a firework having 100 particles; what does the following code fragment print?

```
Firework f2 = new Firework(f1);
f1.removeAllParticles();         // removes all particles from f1
System.out.println( f1.size() );
System.out.println( f2.size() );
```

11. **Collections as fields: Shallow copying**

Suppose that the copy constructor of `Firework` is implemented using shallow copying.

Assume that `f1` is a firework having 100 particles; what does the following code fragment print?

```
Firework f2 = new Firework(f1);
Particle p1 = f1.getParticle(0); // reference to first particle of f1
Particle p2 = f2.getParticle(0); // reference to first particle of f2
System.out.println(p1 == p2);

f1.removeAllParticles();         // removes all particles from f1
System.out.println( f1.size() );
System.out.println( f2.size() );
```

12. **Collections as fields: Deep copying**

Suppose that the copy constructor of `Firework` is implemented using deep copying.

Assume that `f1` is a firework having 100 particles; what does the following code fragment print?

```
Firework f2 = new Firework(f1);
Particle p1 = f1.getParticle(0); // reference to first particle of f1
Particle p2 = f2.getParticle(0); // reference to first particle of f2
System.out.println(p1 == p2);
System.out.println(p1.equals(p2));

f1.removeAllParticles();         // removes all particles from f1
System.out.println( f1.size() );
System.out.println( f2.size() );
```