Data Structures Problem Set #1

Nicholas Yang

18 September 2017

1 Exercise 1

(a) Rectangle rotate methods:

```
 public Rectangle DestRotate() {
    double temp;
    temp = xSpan;
    xSpan = ySpan;
    ySpan = temp;
    return this;
}

public Rectangle NonDestRotate() {
    Rectangle newRect = new Rectangle(xSpan, ySpan);
    newRect.DestRotate();
    return newRect;
}
```

LocatedRect rotate methods:

```
public LocatedRect DestRotate() {
    // First change the lengths of the sides:
    super.DestRotate();
    // Then shift it over by ySpan
    this.translateDest(super.getYSpan(), 0);
    return this;
}

public LocatedRect NonDestRotate() {
    LocatedRect newLocatedRect = new LocatedRect(
      xL,
      xL + xSpan,
      yL,
      yL + ySpan
    );
    return newLocatedRect.DestRotate();
}
```

(b) TestRotate driver program:

```
public class TestRotate {
    public static void main(String[] args) {
        LocatedRect lr = new LocatedRect(2, 10, 4, 7);
        System.out.println("Original Located Rectangle:");
        System.out.println(lr);
        System.out.println("Non destructive rotate");
        System.out.println(lr.NonDestRotate());
        // Note how the object state isn't changed
        System.out.println(lr);
        System.out.println("Destructive rotate:");
        System.out.println(lr.DestRotate());
        // Now the state has been mutated
        System.out.println(lr);
        Rectangle r = new Rectangle(4, 7);
        System.out.println("Original Rectangle:");
        System.out.println(r);
        System.out.println("Non destructive rotate");
        System.out.println(r.NonDestRotate());
        // Again, not mutated here
        System.out.println(r);
        System.out.println("Destructive rotate:");
        System.out.println(r.DestRotate());
        // But mutated here
        System.out.println(r);
    }
}
```

2 Exercise 2

(a) Square class:

```
public class Square extends Rectangle {
    public Square(double side) {
        super(side, side);
    }
    // Doesn't matter what side
    public double getSide() {
        return super.getYSpan();
    }
    public void setSide(double side) {
        super.setSpans(side, side);
    }
    public void setSpans(double x, double y) {
        System.out.println(
            "You may not use setSpans to set the sides of a square!"
        );
    }
}
```

(b) LocatedSquare class:

```
public class LocatedSquare extends Square {
    // Coords of lower left corner
    private double x, y;
    public LocatedSquare(double side, double x, double y) {
        super(side);
        setCorner(x, y);
    }
    public void setCorner(double newX, double newY) {
        x = newX;
        y = newY;
    }
    // Returns rightmost x value (x2)
    public double right() {
        return x + super.getSide();
    }
    // Returns leftmost x value (x1)
    public double left() {
        return x;
    }
    // Returns highest y value (y2)
    public double top() {
        return y + super.getSide();
    }
    // Returns lowest y value (y1)
    public double bottom() {
        return y;
    }
    public String toString() {
            return "LS[ x = " + x + " y = " + y + " side = " + getSide() + "]";
        }
    }
```

(c) Can you do part (b) by having LocatedSquare extend both Square and LocatedRectangle?
Not with traditional inheritance. You could define Square and LocatedRectangle as interfaces, then simply have LocatedSquare implement both. However, it's probably best to just utilize object composition with a Point class for the location and a Square class for the actual object.

3 Exercise 3

Write the following code:

(a) A data field spouse, of class Person:

```
private Person spouse;
```

(b) A getter `getSpouse()`

```
public Person getSpouse() {
    return spouse;
}
```

(c) A method `marry(Person q)`

```
// Ideally would do in intermediary class, maybe a Priest class
public void marry(Person potentialPartner) {
    // Check if related
    boolean isRelated = potentialPartner.getParent2() == this ||
            potentialPartner.getParent1() == this ||
            this.parent1 == potentialPartner ||
            this.parent2 == potentialPartner;

    if (potentialPartner == null) {
        System.out.println("You can't marry imaginary people, " + name);
        return;
    }
    if (potentialPartner == this) {
        System.out.println("You can't marry yourself");
        return;
    }
    if (potentialPartner.getSpouse() != null) {
            System.out.println("Uh oh, they're already married!");
            return;
        }
        if (this.spouse != null) {
            System.out.println("Uh oh, you're already married!");
            return;
        }
        if (isRelated) {
            System.out.println("Ewww");
            return;
        }
        potentialPartner.setSpouse(this);
        setSpouse(potentialPartner);
    }
```

(d) A method `divorce()`

```
    public void divorce() {
        if (this.spouse != null) {
            this.spouse.setSpouse(null);
            this.spouse = null;
        }
    }
```