Data Structures Problem Set #1

Nicholas Yang

18 September 2017

1 Exercise 1

(a) For both `Rectangle` and `LocatedRect`, write two methods that rotate the rectangle by 90^o clockwise around its lower right hand corner (see picture)

Rectangle rotate methods:

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

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

LocatedRect rotate methods:

```
public LocatedRect DestRotate() {
  // First change the lengths of the sides:
  super.DestRotate();
  // Then shift it over by ySpan
  // (rotating around lower right corner
  //  but positioning is done by lower left corner)
  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;
    }
  }
```

(e) D.i requires you to check that `q` is not `null`. Explain why there is no point in checking that `p`, the owner of that method, is not null.

If `p` was null, there would be a NullPointerException. Plus where would that checking happen? `p` is null, so it can't happen there. It would have to happen in the caller.

4 Exercise 4

Consider the code for Hwk1Ex4A.java and Hwk1Ex4B.java on the attached handout

(a) What do these output?
Hwk1Ex4A.java outputs:

```
p = 1
q = 184
r = 184
```

Hwk1Ex4B.java outputs:

```
p = 175
q = 45
r = 25
s = 175
```

(b) Explain these output

(a) Hwk1Ex4A.java defines two classes: A and B. A has two instance variables: name of type String and value of type int. A has a function `f` that updates the value by adding a parameter `x` and 100 to the current value. B extends A and replaces the `f` function with a function `f` that updates value by adding a parameter `x` and 20 to the current value.

The main program starts by creating `p` with class `A` and value 1 and `q` with class `B` and value 1. From there, it sets `r` with class `A` to `q`. Then, by setting `r`'s value to 10, `q`'s value is also set to 10. Afterwards it calls `q.f` with `p.value` as an argument. This sets both `q` (and `r`'s) value to $10 + 1 + 20$ or 31. Calling `r.f` on `q.value` also changes this value, instead this time to $31 + 31 + 20$ or 82. Finally, we repeat this exact same process, only with `q` and `r` switched (they reference the same object so this does the exact same thing as above). This gives us a total of $82+82+20$ or 184 for q and r's value. Since `p`'s value has never been changed, this gives us a final output of 1, 184, and 184. Note that since `q/r` were referencing an object of type `B`, Java dynamically dispatches the `f` method for `B`

(b) Hwk1Ex4B.java defines two classes: `A` and `B`. `Hwk1Ex4B` also defines two static functions, both named `f`. One takes an object of class `A` and one takes an object of class `B`. The first one mutably adds 100 to the object's value, while the second immutably adds 20 to the object's value.

The main program initializes three variables, `p` with declared and actual type `A` and value 1, `q` with declared and actual type `B` and value 5 and `r` with declared and actual type `B` and value 22. Then, `r` is assigned to `f(q)`. Since `q` has a declared type of `B`, `f(q)` resolves to the `B` function. Therefore `r` now has a value of $5+20 = 25$. From there, `q` is assigned the output of `f(r)`. Since `r` has a declared type of `B`, `f(r)` resolves to the `B` function. This gives `q` a value of $25 + 20 = 45$. Next, a new variable `s`, with declared type `A` gets assigned to `f(r)`. Therefore, `s` has a actual type `B` and a value of 45. After that, `p` is assigned to `f(s)`. Since `s` has a declared type of `A`, `f(s)` resolves to the `A` function. Therefore, `s`'s value is incremented by 100, to become 145. Additionally, `p` now references the same

object as `s`, so its value is also 145. Finally, `p`'s value is incremented by 30, which also increments `s`'s value to a final value of 175.