CS2030 Programming Methodology
Semester 2 2019/2020

30 January 2020
Problem Set #2
Testability of Object-Oriented Programs

1. Study the following `Point` and `Circle` classes.

```
public class Point {
    private final double x;
    private final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}

public class Circle {

    private final Point centre;
    private final int radius;

    public Circle(Point centre, int radius) {
        this.centre = centre;
        this.radius = radius;
    }

    @Override
    public boolean equals(Object obj) {
        System.out.println("equals(Object) called");
        if (obj == this) {
            return true;
        }
        if (obj instanceof Circle) {
            Circle circle = (Circle) obj;
            return (circle.centre.equals(centre) && circle.radius == radius);
        } else {
            return false;
        }
    }

    public boolean equals(Circle circle) {
        System.out.println("equals(Circle) called");
        return circle.centre.equals(centre) && circle.radius == radius;
    }
}
```

Given the following program fragment,

```
Circle c1 = new Circle(new Point(0, 0), 10);
Circle c2 = new Circle(new Point(0, 0), 10);
Object o1 = c1;
Object o2 = c2;
```

equal(Object) : 1
equal(Circle) : 2

what is the output of the following statements? reason for false: centre != centre

equal(Object) is called because o1 at compile-time is an Object, therefore, it only allows the default equals of the java.lang.Object Then, during running time, the program checks if equals(Object) is overriden in the Circle class (because o1 has the running-time type of Circle) . In this case, yes => it call the equal(Object) method of the Circle class

1 (a) `o1.equals(o2);` false

1 (b) `o1.equals((Circle) o2);` false

1 (c) `o1.equals(c2);` false

1 (d) `o1.equals(c1);` true

(e) `c1.equals(o2);` false 1

(f) `c1.equals((Circle) o2);` false 2

(g) `c1.equals(c2);` false 2

(h) `c1.equals(o1);` true 1

2. We would like to design a class `Square` that inherits from `Rectangle`. A square has the constraint that the four sides are of the same length.

   (a) How should `Square` be implemented to obtain the following output from `JShell`?

   ```
   jshell> new Square(5)
   $3 ==> area 25.00 and perimeter 20.00
   ```

   (b) Now implement two separate methods to set the width and height of the rectangle:

   ```
   public Rectangle setWidth(double width) { ... }

   public Rectangle setHeight(double height) { ... }
   ```

   What undesirable design issues would this present?
   the square class cannot use setWidth and setHeight method properly.

   (c) Now implement two overriding methods in the `Square` class

   ```
   @Override
   public Square setHeight(double height) {
       return new Square(height);
   }

   @Override
   public Square setWidth(double width) {
       return new Square(width);
   }
   ```

   Do you think that it is now sensible for to have `Square` inherit from `Rectangle`? Or should it be the other way around? Or maybe they should not inherit from each other? Square shouldn't inherit from rectangle
   Because due to Liskov Substitution principle, if the square is the subtype of the Rectangle, then a Rectangle can be replaced by a square without changing the desirable property of a Rectangle.