CS2030 Programming Methodology
Semester 1 2019/2020

30 August 2019
Problem Set #1 Suggested Guidance
Object-Oriented Programming Principles

1. Consider the following two classes:

```
public class P {
    private int x;
    public void changeSelf() {
        x = 1;
    }
    public void changeAnother(P p) {
        p.x = 1;
    }
}

public class Q {
    public void changeAnother(P p) {
        p.x = 1;
    }
}
```

   (a) Which line(s) above violate the private access modifier of `x`?
   The abstraction barrier sits between the client and the implementer. Here class `P` *is the implementer, and* `Q` *is the client that makes use of the* `p`*, an object of* `P`*.*

   (b) What does this say about the concept of an "abstraction barrier"?
   The barrier is not broken when one one object of type `P` *accesses the instance variables of another type* `P` *object, since* `P` *is the sole implementer.*

2. 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;
```

what is the output of the following statements?

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

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

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

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

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

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

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

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

```
jshell> o1.equals(o2)
equals(Object) called
...

jshell> o1.equals((Circle) o2)
equals(Object) called
...

jshell> o1.equals(c2)
equals(Object) called
...

jshell> o1.equals(c1)
equals(Object) called
...

jshell> c1.equals(o2)
equals(Object) called
...

jshell> c1.equals((Circle) o2);
equals(Circle) called
...

jshell> c1.equals(c2)
equals(Circle) called
...

jshell> c1.equals(o1)
equals(Object) called
...
```

Calling the `equals` *method though a reference of type* `Object` *would invoke the* `toString` *method of* `Object`*, but which is overridden by the same method of the sub-class* `Circle`*.*

The only time that the overloaded method `equals(Circle circle)` *can be called is when the method is invoked through an object of* `Circle` *type, and the argument is an object of* `Circle` *type also.*

The output of `true` *or* `false` *largely depends on the presence of an overriding* `equals` *method in the* `Point` *class.*

3. Which of the following program fragments will result in a compilation error?

(a)
```
class A {
    public void f(int x) {}
    public void f(boolean y) {}
}
```

(b)
```
class A {
    public void f(int x) {}
    public void f(int y) {}
}
```

(c)
```
class A {
    private void f(int x) {}
    public void f(int y) {}
}
```

(d)
```
class A {
    public int f(int x) {
        return x;
    }
    public void f(int y) {}
}
```

(e)
```
class A {
    public void f(int x, String s) {}
    public void f(String s, int y) {}
}
```

Method overloading supports a class to have more than one method of the same name (or constructor) with different argument lists (number/type/order of parameters).

(a) Compilable

(b)
```
A.java:3: error: method f(int) is already defined in class A
    public void f(int y) {}
                ^
1 error
```

(c)
```
A.java:3: error: method f(int) is already defined in class A
    public void f(int y) {}
                ^
1 error
```

(d)
```
A.java:5: error: method f(int) is already defined in class A
    public void f(int y) {}
                ^
1 error
```

(e) Compilable

4. Consider the following classes: `FormattedText` that adds formatting information to the text. We call `toggleUnderline()` to add or remove underlines from the text. A `URL` is a `FormattedText` that is always underlined.

```
class FormattedText {
    public String text;
    public boolean isUnderlined;

    public void toggleUnderline() {
        isUnderlined = (!isUnderlined);
    }
}

class URL extends FormattedText {
    public URL() {
        isUnderlined = true;
    }

    @Override
    public void toggleUnderline() {
        return;
    }
}
```

Does the above violate Liskov Substitution Principle? Explain.

Yes. The "desirable property" here is that `toggleUnderline()` *toggles the* `isUnderlined` *flag, i.e. from* `false` *to* `true`*, or from* `true` *to* `false`*.*

Since `URL` *changes the behavior of* `toggleUnderline()`*, this property no longer holds for subclass* `URL`*. Places in a program where the super-class (i.e.* `FormattedText`*) is used cannot be simply replaced by the sub-class (i.e* `URL`*).*

5. 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

public class Rectangle {
    private final double width;
    private final double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
```

```
    }

    public double getArea() {
        return width * height;
    }

    public double getPerimeter() {
        return 2 * (width + height);
    }

    @Override
    public String toString() {
        return "area " + String.format("%.2f", getArea()) +
            " and perimeter " + String.format("%.2f", getPerimeter());
    }
}

public class Square extends Rectangle {
    public Square(double length) {
        super(length, length);
    }
}
```

(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?
A square can be changed to a rectangle

```
jshell> new Square(5.0).setHeight(10.0)
$3 ==> area 50.00 and perimeter 30.00
```

(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?

Based on the substitutability principle, if `Square` *inherits from* `Rectangle`*, then anywhere we expect a* `Rectangle`*, we can always substitute it with a* `Square`*. Consider the following example,*

```
jshell> Rectangle[] rects = {new Rectangle(3.0, 5.0), new Square(5.0)}
rects ==> Rectangle[2] { area 15.00 and perimeter 16.00, area 25.00
and perimeter 20.00 }

jshell> rects[0].setHeight(4.0).setWidth(8.0)
$4 ==> area 32.00 and perimeter 24.00

jshell> rects[1].setHeight(4.0).setWidth(8.0)
$6 ==> area 64.00 and perimeter 32.00
```

Notice that setting `rects[1]` *(of type* `Rectangle`*) to a height of* 4.0 *and a width of* 8.0 *does not produce the desired rectangle.*