CS2030 Programming Methodology II
Semester 1 2023/2024

6 & 7 September 2023
Problem Set #2 Suggested Guidance
Interface

1. Given the following interfaces.

```
interface Shape {
    public double getArea();
}

interface Printable {
    public void print();
}
```

   (a) Suppose class `Circle` implements both interfaces above. Given the following program fragment,

```
Circle c = new Circle(10);
Shape s = c;
Printable p = c;
```

   Are the following statements allowed? Why do you think Java does not allow some of the following statements?

   i. `s.print();`
   ii. `p.print();`
   iii. `s.getArea();`
   iv. `p.getArea();`

   Only `s.getArea()` *and* `p.print()` *are permissible. Suppose* `Shape s` *references an array of objects that implements the* `Shape` *interface, so each object is guaranteed to implement the* `getArea()` *method.*

   Other than that, each object may or may not implement other interfaces (such as `Printable`*), so* `s.print()` *may or may not be applicable.*

   In addition, we say that for the above statement `Shape s = c`*, variable* `s` *has a compile-time type of* `Shape` *but a runtime type of* `Circle`*.*

   (b) Now let's define another interface `PrintableShape` as

```
interface PrintableShape extends Printable, Shape { }
```

   and let class `Circle` implement `PrintableShape` instead.

   Can an interface inherit from multiple parent interfaces? Would the following statements be allowed?

```
Circle c = new Circle(10);
PrintableShape ps = c;
```

i. `ps.print();`

ii. `ps.getArea();`

Yes, it is allowed. Interfaces can inherit from multiple parent interfaces. That said, do consider whether it violates the design principle of Single Responsibility — a class (or interface) should have only one reason to change.

2. Given the following `Circle` and `Rectangle` classes that implement the `Shape` interface,

```
interface Shape {
    public double getArea();
}

class Circle implements Shape {
    private final double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    public double getArea() {
        return Math.PI * this.radius * this.radius;
    }

    public String toString() {
        return "Circle with radius " + this.radius;
    }
}

class Rectangle implements Shape {
    private final double length;
    private final double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    public double getArea() {
        return this.length * this.width;
    }

    public String toString() {
        return "Rectangle " + this.length + " x " + this.width;
    }
}
```

we have seen how both a `Circle` and `Rectangle` object can be passed to the following `findVolume` method to have its volume computed.

```
double findVolume(Shape shape, double height) {
    return shape.getArea() * height;
}
```

Now your friend decided to create `Shape` as a class to represent both a circle and rectangle:

```
class Shape {
    private final String type;
    private final double a;
    private final double b;

    Shape(double radius) {
        this.type = "Circle";
        this.a = radius;
        this.b = 0;
    }

    Shape(double length, double width) {
        this.type = "Rectangle";
        this.a = length;
        this.b = width;
    }

    double getArea() {
        if (this.type.equals("Circle")) {
            return Math.PI * this.a * this.a;
        } else {
            return this.a * this.b;
        }
    }

    public String toString() {
        if (this.type.equals("Circle")) {
            return "Circle with radius " + this.a;
        } else {
            return "Rectangle " + this.a + " x " + this.b;
        }
    }
}
```

which when passed to `findVolume` would still return the same outcome. Justify why *programming to an interface* is a better implementation?
Hint: what if we need to include a `Square` into our implementation?

By making `Shape` *a class and subsuming the responsibilities of* `Circle` *and* `Rectangle` *into it, we have inevitably transformed* `Shape` *into a "God Class" which results in a number of design issues:*

- *As it now oversees different shapes, a "type" has to be defined to denote the exact shape during object creation leading to a misrepresentation in terms of properties of the class (e.g.* `Circle` *and* `Square` *only needs one property, but* `Rectangle` *requires two);*
- *Method calls (e.g.* `getArea` *and* `toString`*) requires that the "type" to be determined before deciding on the appropriate implementation that is invoked;*

More importantly, adding a square will require the `Shape` *class to be modified. We desire a design solution where extensions to the existing code can be "plugged" into the code base with no modifications. By defining* `Shape` *as an interface, with its implementation classes* `Circle` *and* `Rectangle`*, extending the solution with a square simply requires another* `Square` *implementation to be defined.*

3. You are given the following method that returns the maximum integer within a non-empty list of integer elements.

```
int maximum(List<Integer> list) {
    int m = list.get(0);
    int i = 1;
    for (i = 1; i < list.size(); i++) {
        if (list.get(i) > m) {
            m = list.get(i);
        }
    }
    return m;
}
```

We would like to include an equivalent `minimum` method that returns the minimum integer element. How do we define the two methods while avoiding code duplication?

```
int optimum(List<Integer> list, Comparator<Integer> cmp) {
    int m = list.get(0);
    int i = 1;
    for (i = 1; i < list.size(); i++) {
        if (cmp.compare(list.get(i),m) > 0) {
            m = list.get(i);
        }
    }
    return m;
}

class AscCmp implements Comparator<Integer> {
    public int compare(Integer i1, Integer i2) {
        return i1 - i2;
    }
}
```

```
int maximum(List<Integer> list) {
    return optimum(List.of(1,2,3), new AscCmp())
}
```

While one can create another implementation of `Comparator<Integer>` where a smaller integer is deemed to be "larger",

```
class DscCmp implements Comparator<Integer> {
    public int compare(Integer i1, Integer i2) {
        return i2 - i1;
    }
}

int minimum(List<Integer> list) {
    return optimum(List.of(1,2,3), new DscCmp())
}
```

we can also make use of a `reversed()` method of the `Comparator` interface. Note the definition of the `reversed()` method within `Comparator` makes it an "impure" interface.

```
int minimum(List<Integer> list) {
    return optimum(List.of(1,2,3), new AscCmp().reversed())
}
```