CS2030 Programming Methodology
Semester 2 2019/2020

6 February 2020
Problem Set #3

1. Given the following interfaces.

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

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

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

```
Circle c = new Circle(new Point(0,0), 10);
Shape s = c;
Printable p = c;
```

   consider the case where Circle doesn't implement Printable. We can still use casting: Printable p = (Printable)(new Circle()); It will pass the compiling step but will return error at running time because Circle cannot be casted into Printable.

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

   i. s.print(); no, because at compile time s is a Shape Interface's object. This interfact doesn't implement the print method.
   ii. p.print(); yes
   iii. s.getArea(); yes
   iv. p.getArea(); no
   v. c.print(): yes
   vi. c.getArea(): yes

   (b) Someone proposes to re-implement `Shape` and `Printable` as abstract classes instead? Would this work? No, Circle cannot implements 2 Parent classes

   (c) Can we define another interface `PrintableShape` as

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

   interface can inherit from multiple interfaces

   and let class `Circle` implement `PrintableShape` instead? Yes

2. Using examples of overriding methods, illustrate why a Java class cannot inherit from multiple parent classes, but can implement multiple interfaces.

3. 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 extends Class
Class implements Interface
Interface extends Interface

Interface A has f(), Interface B has f()
Class AB implements A,B {
    @Override f()
}
=> both A and B will be implemented with the @overriding f()

```
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;
    }
}
```

By LSP, if we assign an object of type URL into the FormattedText style, then let the client use it, the client will expect toggleUnderline() to remove the underline of the url. However, since URL's overriding toggleUnderline() method doesn't satisfy this desired property, it violates LSP

Does the above violate Liskov Substitution Principle? Explain.

4. Consider the following program fragment.

```
class A {
    int x;
    A(int x) {
        this.x = x;
    }
    public A method() {
        return new A(x);
    }
}

class B extends A {
    B(int x) {
        super(x);
    }
    @Override
    public B method() {
        return new B(x);
    }
}
```

Does it compile? What happens if we switch the method definitions between class `A` and class `B` instead? Give reasons for your observations.

It will compile, because the return type of the overriding method is more specific than the return type of the overriden method. If we switch the method definition between class A and B, it won't compile.