CS2030/S Programming Methodology

Semester 1 2020/2021

9 September 2020

Problem Set #3

1. Given the following interfaces.

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

interface Printable {
    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;
```

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

   cannot inherit multiple parents in java, this is where interfaces come in handy

   i. s.print(); no

   ii. p.print(); yes

   interfaces can inherit from other interfaces

   iii. s.getArea(); no

   extends -> inherit class
   implements -> inherit interface

   iv. p.getArea(); yes

   (b) Someone proposes to re-implement `Shape` and `Printable` as abstract classes instead? Would statements (i) to (iv) be allowed?

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

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

   and let class `Circle` implement `PrintableShape` instead. Would statements (i) to (iv) be allowed now? Can an interface inherit from multiple parent interfaces?
   yes yes

2. Suppose Java allows a class to inherit from multple parent classes. Give a concrete example why this could be problematic. name resolution error – if parents have methods of the same name, child class would not know which method to call

   On the other hand, Java does allow classes to implement multiple interfaces. Explain why this isn't problematic. all methods in interfaces are abstract, child is forced to implement them

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 `PlainText` is a `FormattedText` that is always NOT underlined.

```
class FormattedText {
    private final String text;
    private final boolean isUnderlined;

    FormattedText(String text) {
        this.text = text;
        this.isUnderlined = false;
    }

    /*
     * Overloaded constructor, but made private to prevent
     * clients from calling it directly.
     */
    private FormattedText(String text, boolean isUnderlined) {
        this.text = text;
        this.isUnderlined = isUnderlined;
    }

    FormattedText toggleUnderline() {
        return new FormattedText(this.text, !this.isUnderlined);
    }

    @Override
    public String toString() {
        if (this.isUnderlined) {
            return this.text + "(underlined)";
        } else {
            return this.text;
        }
    }
}

class PlainText extends FormattedText {
    PlainText(String text) {
        super(text); // text is NOT underlined
    }

    @Override
    PlainText toggleUnderline() {
        return this;
    }
}
```

to ensure that the return value is returned in plainText (not underlined)

Does the above violate Liskov Substitution Principle? Explain.

This particular parent-child relationship violates LSP. Parent class (FormattedText) allow the toggling of whether the text is underlined, however the child class (PlainText) prevents this behaviour by overriding its parent's toggleUnderline() method.

This can be resolved by not having the parent-child link (don't inherit from parent).

4. Consider the following program.

```
class A {
    int x;

    A(int x) {
        this.x = x;
    }

    A method() {
        return new A(x);
    }
}

class B extends A {
    B(int x) {
        super(x);
    }

    @Override
    B method() {
        return new B(x);
    }
}
```

Does it compile? What happens if we swap the entire definitions of `method()` between class `A` and class `B`? Does it compile now? Give reasons for your observations.