CS2030/S Programming Methodology
Semester 1 2020/2021

9 September 2020
Problem Set #3 Suggested Guidance

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?

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

   Only `s.getArea()` *and* `p.print()` *are premissible. 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) Someone proposes to re-implement `Shape` and `Printable` as abstract classes instead? Would statements (i) to (iv) be allowed?

   No, you cannot inherit from multiple parent classes.

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

```
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, 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. Suppose Java allows a class to inherit from multple parent classes. Give a concrete example why this could be problematic.

   On the other hand, Java does allow classes to implement multiple interfaces. Explain why this isn't problematic.

   If classes A and B have the same method `f()` *defined, and class C inherits from them, which of the two parent method will be invoked in* `new C().f()`*? However for the case of two interfaces A and B, if they both specify* `f()` *to be defined by a class* `C` *that implements them, then an overridden method in* `C` *would satisfy both contracts.*

   Out-of-scope... For those of you heading towards the dark side... what about impure interfaces with `default` *methods? Try compiling the program fragment below:*

```
interface A {
    default void f() { }
}

interface B {
    default void f() { }
}

class AB implements A, B { }
```

   Does it compile? What if only one of the interface's `f()` *has a default implementation? Does it compile now? And what is the compilation error? What if an overriding method* `f()` *is implemented in class* `AB`*?*

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;
    }
}
```

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

```
jshell> void foo(FormattedText ft) {
   ...> System.out.println(ft.toggleUnderline());
   ...> System.out.println(ft.toggleUnderline().toggleUnderline());
   ...> }
```

```
|  created method foo(FormattedText)

jshell> foo(new FormattedText("cs2030"))
cs2030(underlined)
cs2030

jshell> foo(new PlainText("cs2030"))
cs2030
cs2030
```

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.

There is no compilation error in the given program fragment as any existing code that invokes `A`*'s* `method` *prior to being inherited would still work if the code invokes* `B`*'s* `method` *instead after* `B` *inherits* `A`*.*

When we switch the method definitions, `A`*'s* `method` *now returns a reference to a* `B` *object, but overriding it with a method that returns a reference-type* `A` *does not guarantee that the object is a* `B` *object. So the overriding is not allowed and results in a compilation error.*

Now suppose Java does allow the method() of class A and B to be swapped. Consider the following code fragment, where `g()` *is a method defined in class* `B` *(but not in class* `A`*).*

```
1:  void f(A a) {
2:      B bNew = a.method();
3:      bNew.g();
4:  }
```

Someone else calls `f(new B())`*.* `a.method()` *on Line 2 will invoke* `method()` *defined in* `B`*, which returns an object of class* `A`*. So now,* `bNew` *which has a compile-time type of* `B` *is referencing an instance of* `A`*. The next line* `bNew.g()` *invokes a method* `g()`*, which is defined only in* `B`*, through a reference of (run-time) type* `A`*. But since* `bNew` *is referencing to an object with run-time type* `A`*, this object does not know anything about* `g()`*!*

The following version uses a return type of `Object` *instead.*

```
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
    Object method() { // returns an Object instead
        return new B(x);
    }
}
```

This version causes a compilation error as well. The return type of `B`*'s* `method` *cannot be a supertype of the return type of* `A`*'s* `method`*. If this was allowed, then consider the code below, where* `h()` *is a method that is defined in* `A`*.*

```
1:  void f(A a) {
2:      A aNew = a.method();
3:      aNew.h();
4:  }
```

Now someone calls `f(new B())`. `a.method()` *on Line 2 will invoke* `method()` *defined in* `B`*, which returns an object of class* `Object`*. So now,* `aNew` *which has a compile-time type of* `A` *is referencing to an instance of* `B`*. This actually sounds plausible, since* `aNew` *is referencing to an object of type* `B`*, and calling* `h()` *on an instance of* `B` *should work! The problem, however, is that the return type of* `B`*'s* `method()` *is* `Object`*, and therefore there is no guarantee that* `B`*'s* `method()` *will return an instance of* `B`*. Indeed, the method could return a* `String` *object, for instance, in which case, Line 2 does not make sense anymore.*