CS2030 Programming Methodology II
Semester 2 2022/2023

8 & 9 February 2023
Problem Set #3 Suggested Guidance
Abstract Class and Interface

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(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) Someone proposes to re-implement `Shape` and `Printable` as abstract classes instead? What happens?

   Compilation error. 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.

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. Suppose Java allows a class to inherit from multiple parent classes. Give a concrete example why this could be problematic. Why does Java allow classes to implement multiple interfaces then?

   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.*

3. Consider the following program.

```
class A {
    protected final 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 {
    protected final 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.*