CS2030 Programming Methodology
Semester 1 2019/2020

6 September 2019
Problem Set #2 Suggested Guidance
Inheritance, Interfaces and Polymorphism

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;
```

   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 this work?

   No, you cannot inherit from multiple parent classes.

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

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

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

Yes, it is allowed. Interfaces can inherit from multiple parent interfaces. That said, do consider whether it violates the design principles of Interface Segration (and also more generally Single Responsibility which is also applicable to concrete classes).

2. 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.

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`*. Saying that LSP is not violated is not exactly right, as Java does not check LSP violations during compilation (notice that the questions asks whether the program compiles, and not whether the program violates LSP).*

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. I Imagine someone wrote the following code, 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;
    }
    public A method() {
        return new A(x);
    }
}

class B extends A {
    B(int x) {
        super(x);
    }
    @Override
    public 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 not 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`*.*

```
void f(A a) {
    A aNew = a.method();
    aNew.h();
}
```

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.*

3. Give practical reasons as to why a Java class cannot inherit from multiple parent classes, but can implement multiple interfaces.

   We use the example of overriding methods. 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.*

   In the case of 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`*?*

4. A solid cuboid is a box-shaped solid object made up of a specific material, and having six flat sides with all right-angled corners. From the three sides of the solid cuboid, one can compute the volume. In addition, using the density of the material, the mass can be found. By using object-oriented modeling and applying the *abstraction principle*, design a Java program to facilitate the creation of a solid cuboid object. Show how you can obtain its volume, density as well as mass.

   Tip: When designing the classes and interfaces, keep in mind the SOLID principles, as well as possible extensions you might want to model to generalize your program.

Here is a suggested solution that caters to other solid shapes.

```
interface SolidShape {
    public double getVolume();
    public double getDensity();
    public double getMass();
}

class SolidCuboid {
    private final double length;
    private final double width;
    private final double height;
    private final double density;

    public SolidCuboid(double length, double width, double height, double density) {
        this.length = length;
        this.width = width;
        this.height = height;
        this.density = density;
    }

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

    public double getDensity() {
        return this.density;
    }

    public double getMass() {
        return this.getDensity() * this.getVolume();
    }
}
```

Now what if we would like to include another cuboid (i.e. hollow cuboid) object? How do we fit this into our current model? You may think that since cuboid is a simpler object, then solid cuboid can subclass from it.

```
interface Shape3D {
    public double getVolume();
}

abstract class Solid3D implements Shape3D {
    private final double density;

    protected Solid3D(double density) {
        this.density = density;
    }
```

```
    public abstract double getDensity();
    public double getMass() {
        return getMass() * getDensity();
    }
}

class Cuboid implements Shape3D {
    private final double length;
    private final double width;
    private final double height;

    public Cuboid(double length, double width, double height) {
        this.length = length;
        this.width = width;
        this.height = height;
    }

    @Override
    public double getVolume() {
        return length * width * height;
    }
}
```

This would require that `SolidCuboid` *extends both* `Cuboid` *and* `Solid3D` *(due to the density). But multiple inheritance is not allowed!*

Notice that we have been modeling with **is-a** *relationships thus far. One way to address the problem is to use a* **has-a** *relationship instead. In particular the density property is the responsibility of the* `Material` *class. Now a* `Solid3D` *has a* `Shape3D`*, as well as has a* `Material`*.*

```
public interface Shape3D {
    public double getVolume();
}

public class Cuboid implements Shape3D {
    private final double length;
    private final double width;
    private final double height;

    public Cuboid(double length, double width, double height) {
        this.length = length;
        this.width = width;
        this.height = height;
    }

    @Override
```

```
    public double getVolume() {
        return length * height * width;
    }
}

class Material {
    private final double density;

    public Material(double density) {
        this.density = density;
    }

    public double getDensity() {
        return this.density;
    }
}

public abstract class Solid3D {
    private final Shape3D shape;
    private final Material material;

    public Solid3D(Shape3D shape, Material material) {
        this.shape = shape;
        this.material = material;
    }

    public double getVolume() {
        return this.shape.getVolume();
    }

    public double getDensity() {
        return this.material.getDensity();
    }

    public double getMass() {
        return getVolume() * getDensity();
    }
}

public class SolidCuboid extends Solid3D {

    public SolidCuboid(Cuboid cuboid, Material material) {
        super(cuboid, material);
    }

    public SolidCuboid(double length, double width, double height, double density) {
        this(new Cuboid(length, width, height), new Material(density));
    }
}
```

5. For each of the following program fragments, will it result in a compilation or runtime error? If not, what is the output?

(a)
```
class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
}

B b = new B();
b.f();
A a = b;
a.f();
```

(b)
```
class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
    void f() {
        System.out.println("B f");
    }
}

B b = new B();
b.f();
A a = b;
a.f();
a = new A();
a.f();
```

(c)
```
class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
    void f() {
        super.f();
        System.out.println("B f");
    }
}

B b = new B();
b.f();
A a = b;
a.f();
```

(d)
```
class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
    void f() {
        this.f();
        System.out.println("B f");
    }
}

B b = new B();
b.f();
A a = b;
a.f();
```

(e)
```
class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
    int f() {
        System.out.println("B f");
        return 0;
    }
}

B b = new B();
b.f();
A a = b;
a.f();
```

(f)
```
class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
    void f(int x) {
        System.out.println("B f");
    }
}

B b = new B();
b.f();
b.f(0);
A a = b;
a.f();
a.f(0);
```

(g)
```
class A {
    public void f() {
        System.out.println("A f");
    }
}

class B extends A {
    public void f() {
        System.out.println("B f");
    }
}

B b = new B();
A a = b;
a.f();
b.f();
```

(h)
```
class A {
    private void f() {
        System.out.println("A f");
    }
}

class B extends A {
    public void f() {
        System.out.println("B f");
    }
}

class Main {
    public static void main(String[] args) {
        B b = new B();
        A a = b;
        a.f();
        b.f();

    }
}
```

(i)
```
class A {
    static void f() {
        System.out.println("A f");
    }
}

class B extends A {
    public void f() {
        System.out.println("B f");
    }
}

B b = new B();
A a = b;
a.f();
b.f();
```

(j)
```
class A {
    static void f() {
        System.out.println("A f");
    }
}

class B extends A {
    static void f() {
        System.out.println("B f");
    }
}

B b = new B();
A a = b;
A.f();
B.f();
a.f();
b.f();
```

(k)
```
class A {
    private int x = 0;
}

class B extends A {
    public void f() {
        System.out.println(x);
    }
}

B b = new B();
b.f();
```

(l)
```
class A {
    private int x = 0;
}

class B extends A {
    public void f() {
        System.out.println(super.x);
    }
}

B b = new B();
b.f();
```

(m)
```
class A {
    protected int x = 0;
}

class B extends A {
    public void f() {
        System.out.println(x);
    }
}

B b = new B();
b.f();
```

(n)
```
class A {
    protected int x = 0;
}

class B extends A {
    public int x = 1;
    public void f() {
        System.out.println(x);
    }
}

B b = new B();
b.f();
```

(o)
```
class A {
    protected int x = 0;
}

class B extends A {
    public int x = 1;
    public void f() {
        System.out.println(super.x);
    }
}

B b = new B();
b.f();
```

You are encouraged try these out themselves. Just some noteworthy mention below:

- *(d) results in an infinite recursion leading to stack overflow*
- *(e) is a compilation error as method* `f()` *has the same method signature, which implies the method in* `B` *should override that of* `A`*, but the return type is different.*
- *In (f),* `a.f(0)` *is not accessible, only* `a.f()` *is ok.*
- *In (h),* `a.f()` *has private access.a*
- *(i) is a compilation error as* `f()` *in* `B` *cannot override* `f()` *in* `A` *which is declared* `static`*.* `static` *methods cannot be overridden.*
- *(k) is a compilation error as* `x` *has private access; likewise for (l)*

More detailed output below:

```
AB1.java
A f
A f
AB2.java
B f
B f
A f
AB3.java
A f
B f
A f
B f
AB4.java
|  java.lang.StackOverflowError thrown:
|        at B.f (#2:3)
:
|        at B.f (#2:3)
AB5.java
|  Error:
|  f() in B cannot override f() in A
|    return type int is not compatible with void
|      int f() {
|      ^--------...
AB6.java
A f
B f
A f
|  Error:
|  method f in class A cannot be applied to given types;
|    required: no arguments
|    found: int
|    reason: actual and formal argument lists differ in length
|  a.f(0);
|  ^-^
AB7.java
```

```
B f
B f
AB8.java
|  Error:
|  f() has private access in A
|  a.f();
|  ^-^
B f
AB9.java
|  Error:
|  f() in B cannot override f() in A
|    overridden method is static
|      public void f() {
|      ^-----------------...
AB10.java
A f
B f
A f
B f
AB11.java
|  Error:
|  x has private access in A
|          System.out.println(x);
|                             ^
AB12.java
|  Error:
|  x has private access in A
|          System.out.println(super.x);
|                             ^-----^
AB13.java
0
AB14.java
1
AB15.java
```