CS2030 Programming Methodology II
Semester 1 2022/2023

1 & 2 February 2023
Problem Set #2 Suggested Guidance
Inheritance and Polymorphism

1. Study the following `Circle` class.

```
class Circle {
    private final int radius;

    Circle(int radius) {
        this.radius = radius;
    }

    @Override
    public String toString() {
        return "Circle with radius " + this.radius;
    }
}
```

We have seen how the `toString` method can be defined in the `Circle` class that overrides the same method in its parent `java.lang.Object` class. There is another `equals(Object obj)` method defined in the `Object` class which returns `true` only if the object from which `equals` is called, and the argument object is the same.

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)

```
jshell> Circle c = new Circle(10)
c ==> Circle with radius 10

jshell> c.equals(c)
$.. ==> true

jshell> c.equals("10")
$.. ==> false

jshell> c.equals(new Circle(10))
$.. ==> false
```

In particular for the latter test, since both `c` and `new Circle(10)` have radius of 10 units, we would like the `equals` method to return `true` instead.

(a) We define an overloaded method `equals(Circle other)` in the `Circle` class:

```
boolean equals(Circle circle) {
    System.out.println("Running equals(Circle) method");
    return circle.radius == radius;
}
```

such that

```
jshell> new Circle(10).equals(new Circle(10))
Running equals(Circle) method
$.. ==> true

jshell> new Circle(10).equals("10")
$.. ==> false
```

Why is the outcome of the following test `false`?

```
jshell> Object obj = new Circle(10)
obj ==> Circle with radius 10

jshell> obj.equals(new Circle(10))
$.. ==> false
```

Declare `obj` *of type* `Object` *and assign it to a* `Circle` *object. Calling the* `equals` *method by passing in another* `Circle` *with radius* 1 *will result in* `false`*!*

(b) Instead of an overloaded method, we now define an overriding method.

```
@Override
public boolean equals(Object obj) {
    System.out.println("Running equals(Object) method");
    if (obj == this) { // trivially true since it's the same object
        return true;
    } else if (obj instanceof Circle circle) { // is obj a Circle?
        return circle.radius == this.radius;
    } else {
        return false;
    }
}
```

Why does the same test case in question 1a now produce the correct expected outcome?

```
jshell> Object obj = new Circle(10)
obj ==> Circle with radius 10

jshell> obj.equals(new Circle(10))
Running equals(Object) method
$.. ==> true
```

Since `obj` *has a runtime type of* `Circle`*, the overriding* `equals` *method in the* `Circle` *class will be invoked which checks the radius for equality.*

```
jshell> Object obj = new Circle(10)
obj ==> Circle with radius 10

jshell> obj.equals(new Circle(10))
Running equals(Object) method
$.. ==> true
```

As an aside, one should avoid using `instanceof` *to check the type of the object and decide what method will run, hence avoiding overriding methods entirely. For example, if* `Shape s` *can either be assigned to* `Circle` *or* `Rectangle`*, then one could check the type of the object and invoke the appropriate method to say, get the area. Restrict the use of* `instanceof` *only in the equals method of class* `A` *(that checks* `instanceof A`*). The following is considered bad code:*

```
class Shape {
   double radius;
   double height;
   double width;

   Shape(double r, double h, double w) {
       this.radius = r;
       this.height = h;
       this.width = w;
   }

   double getCircleArea() {
       return Math.PI * this.radius * this.radius;
   }

   double getRectangleArea() {
       return this.height * this.width;
   }
}

class Circle extends Shape {
    Circle(double radius) {
        super(radius, 0.0, 0.0);
    }
}

class Rectangle extends Shape {
    Rectangle(double height, double width) {
        super(0.0, height, width);
    }
}

jshell> double getArea(Shape s) {
   ...>    if (s instanceof Circle) {
   ...>       return s.getCircleArea();
   ...>    }
   ...>    if (s instanceof Rectangle) {
   ...>       return s.getRectangleArea();
   ...>    }
   ...> }
|  created method getArea(Shape)

jshell> getArea(new Circle(1.0))
$.. ==> 3.141592653589793

jshell> getArea(new Rectangle(4.0, 5.0))
$.. ==> 20.0
```

(c) With both the overloaded and overriding `equals` method in questions 1a and 1b defined, given the following program fragment,

```
Circle c1 = new Circle(10);
Circle c2 = new Circle(10);
Object o1 = c1;
Object o2 = c2;
```

what is the output of the following statements?

(a) `o1.equals(o2);`
(b) `o1.equals(c2);`
(c) `o1.equals(c1);`
(d) `c1.equals(o2);`
(e) `c1.equals(c2);`
(f) `c1.equals(o1);`

```
jshell> Circle c1 = new Circle(10)
c1 ==> Circle with radius 10

jshell> Circle c2 = new Circle(10)
c2 ==> Circle with radius 10

jshell> Object o1 = c1
o1 ==> Circle with radius 10

jshell> Object o2 = c2
o2 ==> Circle with radius 10

jshell> o1.equals(o2) // Object::equals(Object) chosen,
                      // but overridden by Circle::equals(Object)
Running equals(Object) method
$.. ==> true

jshell> o1.equals(c2) // same as above
Running equals(Object) method
$.. ==> true

jshell> o1.equals(c1) // same as above
Running equals(Object) method
$.. ==> true

jshell> c1.equals(o2) // Circle::equals(Object) chosen; activated during runtime
Running equals(Object) method
$.. ==> true

jshell> c1.equals(c2)// Circle::equals(Circle) chosen; activated during runtime
Running equals(Circle) method
$.. ==> true

jshell> c1.equals(o1)// Circle::equals(Object) chosen; activated during runtime
Running equals(Object) method
$.. ==> true
```

Calling the `equals` *method though a reference of compile-time type* `Object` *would invoke the* `equals(Object)` *method of* `Object`*. This method is overridden by the overriding method in the sub-class* `Circle`*.*

The only time that the overloaded method `equals(Circle)` *can be called is when the method is invoked through a variable of compile-time type* `Circle`*.*

Moreover, defining only the overriding `equals` *method is sufficient to make all the above six test cases return* `true`*. On the other hand, defining only the overloaded* `equals` *method results in (a), (b) and (d) returning* `false`*.*

2. We would like to design a class `Square` that inherits from `Rectangle`.

```
class Rectangle {
    private final int width;
    private final int height;

    Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public String toString() {
        return this.width + " x " + this.height;
    }
}
```

As an example of constructing a rectangle,

```
jshell> new Rectangle(3, 4) // width = 3 and height = 4
$.. ==> 3 x 4
```

(a) A square has the constraint that the four sides are of the same length. Keeping in mind the *abstraction principle*, how should `Square` be implemented to obtain the following evaluation from `JShell`?

```
jshell> new Square(5)
$.. ==> 5 x 5
```

```
class Square extends Rectangle {
    Square(int side) {
        super(side, side);
    }
}
```

(b) Now implement two separate methods to set the width and height of the rectangle:

```
Rectangle setWidth(double width) { ... }
Rectangle setHeight(double height) { ... }

jshell> new Rectangle(3, 4).setHeight(2)
$.. ==> 3 x 2
```

Include the following in `Rectangle` *class.*

```
    Rectangle setWidth(int width) {
        return new Rectangle(width, this.height);
    }

    Rectangle setHeight(int height) {
        return new Rectangle(this.width, height);
    }
```

(c) What happens if `Square` inherits the methods `setWidth` and `setHeight` from `Rectangle`?

A square might no longer be a square.

```
jshell> new Square(5).setHeight(2)
$.. ==> 5 x 2

jshell> Square s = new Square(5)
s ==> 5 x 2

jshell> s = s.setHeight(2) // What happens here?
```

(d) How would you override the methods `setWidth` and `setHeight` in the `Square` class?

We can override the methods as such:

```
    @Override
    Square setHeight(int height) {
        return new Square(height);
    }

    @Override
    Square setWidth(int width) {
        return new Square(width);
    }
```

Running the same test now gives

```
jshell> new Square(5).setHeight(2)
$.. ==> 2 x 2
```

It would probably make more sense if we have a method `setSide`*. However, the scaling methods from* `Rectangle` *will still have to be inherited.*

(e) Do you think that it is now sensible to have `Square` inherit from `Rectangle`?

Based on the substitutability principle, if `Square` *inherits from* `Rectangle`*, then anywhere we expect a* `Rectangle`*, we can always substitute it with a* `Square`*. Consider the following example,*

```
jshell> Rectangle rectangle = new Square(5)
rectangle ==> 5 x 5

jshell> rectangle.setWidth(2)
$.. ==> 2 x 2
```

From the clients view of using rectangles, setting the width should only change the width; however height is also changed!

(f) Should `Rectangle` inherit from `Square`? Or maybe they should not inherit from each other at all?

If rectangle is a square, then is rectangle substitutable for a square? Rectangle have different width and height, but square has same sides!

```
jshell> Square square = new Rectangle(2, 3)
square ==> 2 x 3 // ???
```

3. Which of the following program fragments will result in a compilation error?

(a)
```
class A1 {
    void f(int x) {}
    void f(boolean y) {}
}
```

(b)
```
class A2 {
    void f(int x) {}
    void f(int y) {}
}
```

(c)
```
class A3 {
    private void f(int x) {}
    void f(int y) {}
}
```

(d)
```
class A4 {
    int f(int x) {
        return x;
    }
    void f(int y) {}
}
```

(e)
```
class A5 {
    void f(int x, String s) {}
    void f(String s, int y) {}
}
```

Methods of the same name can co-exist as long as their method signatures (number, type, order of arguments) are different.

```
A2.java:3: error: method f(int) is already defined in class A
    void f(int y) {}
                ^
1 error
A3.java:3: error: method f(int) is already defined in class A
    void f(int y) {}
                ^
1 error
A4.java:5: error: method f(int) is already defined in class A
    void f(int y) {}
                ^
1 error
```