CS2030 Programming Methodology II
Semester 1 2023/2024

13 & 14 September 2023
Problem Set #3 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* 10 *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. 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.

   Unfortunately we did not have enough time to cover this question. Nonetheless I have appended the suggested guidance below, so as to hasten the discussion during the next session.

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.*

4. 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
```