# CS2030 Programming Methodology II

Semester 1 2024/2025

Week of 16 – 20 September 2024

Problem Set #4 Suggested Guidance

## Inheritance and Substitutability

1. The `equals(Object obj)` method defined in the `Object` class 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/21/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)

   Now we define an *overloaded* `equals` method, as well as an *overriding* `equals` method in the `Circle` class.

```
class Circle {
    private final int radius;

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

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

    @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;
        }
        if (obj instanceof Circle circle) { // is obj a Circle?
            return this.radius == circle.radius;
        }
        return false;
    }

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

   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 via compile-time type* `Object` *would invoke the* `equals(Object)` *method of the* `Object` *class. 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`*.*

*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 execute the appropriate actions say, getting the area of circle or rectangle. In general, you should restrict to using* `instanceof` *only in the equals method of class* `A` *(to check an* `instanceof A`*).*

2. Consider the following program.

```
class A {
    protected final int x;

    A(int x) {
        this.x = x;
    }

    A method() {
        return new A(this.x);
    }
}

class B extends A {
    B(int x) {
        super(x);
    }

    @Override
    B method() {
        return new B(super.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` *(that returns a* `A` *object) would still work if the code invokes* `B`*'s* `method` *(that returns a* `B` *object), since* `B` *is-a* `A`*. To see this, consider additional methods* `f()` *defined in* `A`*, and* `g()` *defined in* `B`*.*

```
class A {
    ...

    A method() {
        return new A(x);
    }
    void f() { }
}
```

```
class B extends A {
    ...
    @Override
    B method() {
        return new B(super.x);
    }
    void g() { }
}
```

*Now suppose there is a client method* `foo` *that takes in an argument of type* `A`*.*

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

*Clearly, the above compiles when calling* `foo(new A(1))`*. Moreover, it also compiles when calling* `foo(new B(1))` *as the overriding* `method()` *in line 2 returns a* `B` *object. Since* `B` *is-a* `A`*, it inherits the* `f()` *method.*

*When we switch the method definitions,*

```
class A {
    ...

    B method() {
        return new B(x);
    }
    void f() { }
}
```

```
class B extends A {
    ...
    @Override
    A method() {
        return new A(super.x);
    }
    void g() { }
}
```

*and considering the client method* `bar` *below:*

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

*While calling* `bar(new A(1))` *compiles,* `bar(new B(1))` *on the other hand will return an* `A` *object in line 2 which does not have the functionality of method g().*

*Now suppose class* `B` *uses a version that returns* `Object` *instead.*

```
class A {
    ...

    A method() {
        return new A(x);
    }
    void f() { }
}
```

```
class B extends A {
    ...
    @Override
    Object method() {
        return new B(x);
    }
    void g() { }
}
```

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

*This version causes a compilation error as well. It seems that calling* `foo(new B(1))` *would return a* `B` *object and still have the functionality of method* `f()`*. However, the compile-time type of* `a.method()` *is* `Object` *which does not define* `f()`*. Indeed,* `method()` *in class* `B` *could return a* ***String****!*

***Liskov Substitution Principle (LSP): To ensure that a child class is substitutable for the parent class, the return type of an overriding method of a child class cannot be more general (i.e. a super-class or super-type) than the return type of the overridden method of the parent class.***

*Yet another aspect of the application of LSP can be seen from the accessibility of overriding and overridden methods. In general, the* ***accessibility of a overriding method in a child class cannot be more restrictive than that of the overridden method in the parent class.*** *See if you can convince yourself as to why this is so.*

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
```

*More on the Liskov Substitution Principle...* Consider the class `FormattedText` where calling `toggleUnderline()` will 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() {
        System.out.println("Toggling formatted text");
        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() {
        System.out.println("Toggling plain text");
        return this;
    }
}
```

Does the above violate the Liskov Substitution Principle? Notice that `toggleUnderline()` is supposed to toggle the `isUnderlined` flag, i.e. from `false` to `true`, or from `true` to `false`, which is the expected behaviour for clients of `FormattedText`.

```
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
```

However, substituting `FormattedText` with `PlainText` changes the toggling behavior of `toggleUnderline()` and hence, breaks the expected behaviour of the clients.

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

Unlike return types or accessibility modifiers, the compiler cannot check whether LSP is violated as this is due to the behaviour of the program.