Unit 14: Polymorphism

After reading this unit, students should

- understand dynamic binding and polymorphism
- be aware of the `equals` method and the need to override it to customize the equality test
- understand when narrowing type conversion and type casting are allowed

Taking on Many Forms

Method overriding enables *polymorphism*, the fourth and the last pillar of OOP, and arguably the most powerful one. It allows us to change how existing code behaves, without changing a single line of the existing code (or even having access to the code).

Consider the function `say` below:

```
1  void say(Object obj) {
2      System.out.println("Hi, I am " + obj.toString());
3  }
```

Note that this method receives an `Object` instance. Since both `Point` <: `Object` and `Circle` <: `Object`, we can do the following:

```
1  Point p = new Point(0, 0);
2  say(p);
3  Circle c = new Circle(p, 4);
4  say(c);
```

When executed, `say` will first print `Hi, I am (0.0, 0.0)`, followed by `Hi, I am { center: (0.0, 0.0), radius: 4.0 }`. *We are invoking the overriding* `Point::toString` *in the first call, and* `Circle::toString` *in the second call.* The same method invocation `obj.toString()` causes two different methods to be called in two separate invocations!

In biology, polymorphism means that an organism can have many different forms. Here, the variable `obj` can have many forms as well. Which method is invoked is decided *during run-time*, depending on the run-time type of the `obj`. This is called *dynamic binding* or *late binding* or *dynamic dispatch*.

Before we get into this in more detail, let consider overriding `Object::equals`.

The `equals` method

`Object::equals` compares if two object references refer to the same object. Suppose we have:

```
Circle c0 = new Circle(new Point(0, 0), 10);
Circle c1 = new Circle(new Point(0, 0), 10);
Circle c2 = c1;
```

`c2.equals(c1)` returns `true`, but `c0.equals(c1)` returns `false`. Even though `c0` and `c1` are *semantically* the same, they refer to the two different objects.

To compare if two circles are *semantically* the same, we need to override this method[1].

```
// version 0.7
import java.lang.Math;

/**
 * A Circle object encapsulates a circle on a 2D plane.
 */
class Circle {
  private Point c;   // the center
  private double r;  // the length of the radius

  /**
   * Create a circle centered on Point c with given radius r
  */
  public Circle(Point c, double r) {
    this.c = c;
    this.r = r;
  }

  /**
   * Return the area of the circle.
   */
  public double getArea() {
    return Math.PI * this.r * this.r;
  }

  /**
   * Return true if the given point p is within the circle.
   */
  public boolean contains(Point p) {
    return false;
    // TODO: Left as an exercise
  }

  /**
   * Return the string representation of this circle.
   */
  @Override
  public String toString() {
      return "{ center: " + this.c + ", radius: " + this.r + " }";
```

```
40    }
41
42    /**
43     * Return true the object is the same circle (i.e., same center, same
44  radius).
45     */
46    @Override
47    public boolean equals(Object obj) {
48      if (obj instanceof Circle) {
49        Circle circle = (Circle) obj;
50        return (circle.c.equals(this.c) && circle.r == this.r);
51      }
52      return false;
53    }
  }
```

This is more complicated than `toString`. There are a few new concepts involved here:

- `equals` takes in a parameter of compile-time type `Object`. It only makes sense if we compare (during run-time) a circle with another circle. So, we first check if the run-time type of `obj` is a subtype of `Circle`. This is done using the `instanceof` operator. The operator returns `true` if `obj` has a run-time type that is a subtype of `Circle`.
- To compare `this` circle with the given circle, we have to access the center `c` and radius `r`. But if we access `obj.c` or `obj.r`, the compiler will complain. As far as the compiler is concerned, `obj` has the compile-time type `Object`, and there is no such fields `c` and `r` in the class `Object`! This is why, after assuring that the run-time type of `obj` is a subtype of `Circle`, we assign `obj` to another variable `circle` that has the compile-time type `Circle`. We finally check if the two centers are equal (again, `Point::equals` is left as an exercise) and the two radii are equal[2].
- The statement that assigns `obj` to `circle` involves *type casting*. We mentioned before that Java is strongly typed and so it is very strict about type conversion. Here, Java allows type casting from type T to S if $S <: T$. [3]: This is called *narrowing type conversion*. Unlike widening type conversion, which is always allowed and always correct, a *narrowing type conversion* requires explicit typecasting and validation during run-time. If we do not ensure that `obj` has the correct run-time type, casting can lead to a run-time error (which if you recall, is bad).

All these complications would go away, however, if we define `Circle::equals` to take in a `Circle` as a parameter, like this:

```
1   class Circle {
2       :
3     /**
4      * Return true the object is the same circle (i.e., same center, same
5   radius).
6      */
7     @Override
```

```
8     public boolean equals(Circle circle) {
9         return (circle.c.equals(this.c) && circle.r == this.r);
10    }
    }
```

This version of `equals` however, does not override `Object::equals`. Since we hinted to the compiler that we meant this to be an overriding method, using `@Override`, the compiler will give us an error. This is not treated as method overriding, since the signature for `Circle::equals` is different from `Object::equals`.

Why then is overriding important? Why not just leave out the line `@Override` and live with the non-overriding, one-line, `equals` method above?

The Power of Polymorphism

Let's consider the following example. Suppose we have a general `contains` method that takes in an array of objects. The array can store any type of objects: `Circle`, `Square`, `Rectangle`, `Point`, `String`, etc. The method `contains` also takes in a target `obj` to search for, and returns true if there is an object in `array` that equals to `obj`.

```
1  // version 0.1 (with polymorphism)
2  boolean contains(Object[] array, Object obj) {
3    for (Object curr : array) {
4      if (curr.equals(obj)) {
5        return true;
6      }
7    }
8    return false;
9  }
```

With overriding and polymorphism, the magic happens in Line 4 -- depending on the run-time type of `curr`, the corresponding, customized version of `equals` is called to compare against `obj`.

However, if `Circle::equals` takes in a `Circle` as the parameter, the call to `equals` inside the method `contains` would not invoke `Circle::equals`. It would invoke `Object::equals` instead due to the matching method signature, and we can't search for `Circle` based on semantic equality.

To have a generic `contains` method without polymorphism and overriding, we will have to do something like this:

```
1  // version 0.2 (without polymorphism)
2  boolean contains(Object[] array, Object obj) {
3    for (Object curr : array) {
4      if (obj instanceof Circle) {
```

```
5         if (curr.equals((Circle)obj)) {
6           return true;
7         }
8       } else if (obj instanceof Square) {
9         if (curr.equals((Square)obj)) {
10          return true;
11        }
12      } else if (obj instanceof Point) {
13        if (curr.equals((Point)obj)) {
14          return true;
15        }
16      }
17       :
18    }
19    return false;
20  }
```

which is not scalable since every time we add a new class, we have to come back to this method and add a new branch to the `if-else` statement!

As this example has shown, polymorphism allows us *to write succinct code that is future proof*. By dynamically deciding which method implementation to execute during run-time, the implementer can write short yet very general code that works for existing classes as well as new classes that might be added in the future by the client, without even the need to re-compile!

---

1. If we override `equals()`, we should generally override `hashCode()` as well, but let's leave that for another lesson on another day. ↵
2. The right way to compare two floating-point numbers is to take their absolute difference and check if the difference is small enough. We are sloppy here to keep the already complicated code a bit simpler. You shouldn't do this in your code. ↵
3. This is not the only condition where type casting is allowed. We will look at other conditions in later units. ↵