Unit 9: Composition

After learning this unit, students should understand:

- how to compose a new class from existing classes using composition
- how composition models the HAS-A relationship
- how sharing reference values in composed objects could lead to surprising results

Adding more Abstractions

Our previous implementation of `Circle` stores the center using its Cartesian coordinate (x, y). We have a method `contains` that takes in the Cartesian coordinate of a point. As such, our implementation of `Circle` assumes that a 2D point is best represented using its Cartesian coordinate.

Recall that we wish to hide the implementation details as much as possible, protecting them with an abstraction barrier, so that the client does not have to bother about the details and it is easy for the implementer to change the details. In this example, what happens if the application finds that it is more convenient to use polar coordinates to represent a 2D point? We will have to change the code of the constructor to `Circle` and the method `contains`. If our code contains other shapes or other methods in `Circle` that similarly assume a point is represented with its Cartesian coordinate, we will have to change them as well. It is easy for bugs to creep in. For instance, we might pass in the polar coordinate (r, θ) to a method, but the method treats the two parameters as the Cartesian (x, y).

We can apply the principle of abstraction and encapsulation here, and create a new class `Point`. The details of which are omitted and left as an exercise.

With the `Point` class, our `Circle` class looks like the following:

```
1  // version 0.5
2  import java.lang.Math;
3
4  /**
5   * A Circle object encapsulates a circle on a 2D plane.
6   */
7  class Circle {
8    private Point c;   // the center
9    private double r;  // the length of the radius
```

```
10
11    /**
12     * Create a circle centered on Point c with given radius r
13    */
14    public Circle(Point c, double r) {
15      this.c = c;
16      this.r = r;
17    }
18
19    /**
20     * Return the area of the circle.
21     */
22    public double getArea() {
23      return Math.PI * this.r * this.r;
24    }
25
26    /**
27     * Return true if the given point p is within the circle.
28     */
29    public boolean contains(Point p) {
30      // TODO: Left as an exercise
31      return false;
32    }
33  }
```

This example also illustrates the concept of *composition*. Our class `Circle` has been upgraded from being a bundle of primitive types and its methods, to a bundle that includes a reference type `Point` as well. In OOP, composition is a basic technique to build up layers of abstractions and construct sophisticated classes.

We have mentioned that classes model real-world entities in OOP. The composition models that HAS-A relationship between two entities. For instance, a circle *has a* point as the center.

Example: `Cylinder`

Now let's build up another layer of abstraction and construct a 3D object -- a cylinder. A cylinder has a circle as its base and has a height value. Using composition, we can construct a `Cylinder` class:

```
1   class Cylinder {
2     private Circle base;
3     private double height;
4
5     public Cylinder(Circle base, double height) {
6       this.base = base;
7       this.height = height;
8     }
9       :
10  }
```

Sharing References (aka Aliasing)

Recall that unlike primitive types, reference types may share the same reference values. This is called *aliasing*. Let's look at the subtleties of how this could affect our code and catch us by surprise.

Consider the following, where we create two circles `c1` and `c2` centered at the origin (0, 0).

```
Point p = new Point(0, 0);
Circle c1 = new Circle(p, 1);
Circle c2 = new Circle(p, 4);
```

Let's say that we want to allow a Circle to move its center. For the sake of this example, let's allow mutators on the class `Point`. Suppose we want to move `c1` and only `c1` to be centered at (1,1).

```
p.moveTo(1, 1);
```

You will find that by moving `p`, we are actually moving the center of *both* `c1` and `c2`! This result is due to both circles `c1` and `c2` sharing the same point. When we pass the center into the constructor, we are passing the reference instead of passing a cloned copy of the center.

This is a common source of bugs and we will see how we can reduce the possibilities of such bugs later in this module, but let's first consider the following "fix" (that is still not ideal).

Let's suppose that instead of moving `p`, we add a `moveTo` method to the `Circle` instead:

```
class Circle {
  private Point c;   // the center
  private double r;  // the length of the radius
    :
  /**
   * move the center of this circle to the given point
   */
  void moveTo(Point c) {
      this.c = c;
  }

   :
}
```

Now, to move `c1`,

```
Point p = new Point(0, 0);
Circle c1 = new Circle(p, 1);
Circle c2 = new Circle(p, 4);
c1.moveTo(new Point(1, 1));
```

You will find that `c1` will now have a new center, but `c2`'s center remains at (0,0). Why doesn't this solve our problem then? Recall that we can further composed circles into other objects. Let's say that we have two cylinders:

```
Cylinder cylinder1 = new Cylinder(c1, 1);
Cylinder cylinder2 = new Cylinder(c1, 1);
```

that share the same base, then the same problem repeats itself!

One solution is to avoid sharing references as much as possible. For instance,

```
Point p1 = new Point(0, 0);
Circle c1 = new Circle(p1, 1);

Point p2 = new Point(0, 0);
Circle c2 = new Circle(p2, 4);

p1.moveTo(1, 1);
```

Without sharing references, moving `p1` only affects `c1`, so we are safe.

The drawback of not sharing objects with the same content is that we will have a proliferation of objects and the computational resource usage is not optimized. This is an example of the trade offs we mentioned in the introduction to this module: we are sacrificing the computational cost to save programmers from potential suffering!

Another approach to address this issue is *immutability*. We will cover this later in the module.