Unit 11: Inheritance

After taking this unit, students should:

- understand inheritance as a mechanism to extend existing code
- understand how inheritance models the IS-A relationship
- know how to use the `extends` keyword for inheritance
- understand inheritance as a subtype
- be able to determine the run-time type and compile-time type of a variable

Extension with Composition

We have seen how composition allows us to compose a new, more complex, class, out of existing classes, without breaking the abstraction barrier of existing classes. Sometimes, however, composition is not the right approach. Let's consider the following example. Let's suppose that we, as a client, want to add color as a property to our `Circle`.

Without penetrating the abstraction barrier of `Circle`, we can do the following:

```java
// version 0.1 (using composition)
class ColoredCircle {
  private Circle circle;
  private Color color;

  public ColoredCircle(Circle circle, Color color) {
    this.circle = circle;
    this.color = color;
  }
}
```

where `Color` is another abstraction representing the color of shapes.

What should we do if we want to calculate the area of our colored circle? Suppose we already have a `ColoredCircle` instance called `coloredCircle`. We could make `circle` public and call `coloredCircle.circle.getArea()`, or we could add an accessor and call `coloredCircle.getCircle().getArea()`. Both of these are not ideal, since it breaks the abstraction barrier and reveals that the `ColoredCircle` class stores a `circle` (the latter being slightly better than the first).

A better alternative is to let `ColoredCircle` provide its own `getArea()` method, and *forward* its call to `Circle`.

```java
// version 0.2 (using composition)
class ColoredCircle {
  private Circle circle;
  private Color color;

  public ColoredCircle(Circle circle, Color color) {
    this.circle = circle;
    this.color = color;
  }

  public double getArea() {
    return circle.getArea();
  }
}
```

Then, the client to `ColoredCircle` can just call `coloredCircle.getArea()` without knowing or needing to know how a colored circle is represented internally. The drawback of this approach is that we might end up with many such boilerplate forwarding methods.

Extension with Inheritance

Recall the concept of subtyping. We say that $S <: T$ if any piece of code written for type T also works for type S.

Now, think about `ColoredCircle` and `Circle`. If someone has written a piece of code that operates on `Circle` objects. Do we expect the same code to work on `ColoredCircle`? In this example, yes! A `ColoredCircle` object should behave just like a circle -- we can calculate its area, circumference, check if two circles intersect, check if a point falls within the circle, etc. The only difference, or more precisely, extension, is that it has a color, and perhaps has some methods related to this additional field. So, `ColoredCircle` *is a subtype of* `Circle`.

We now show you how we can introduce this subtype relationship in Java, using the `extends` keyword. We can reimplement our `ColoredCircle` class this way:

```java
// version 0.3 (using inheritance)
class ColoredCircle extends Circle {
  private Color color;

  public ColoredCircle(Point center, double radius, Color color) {
    super(center, radius);  // call the parent's constructor
    this.color = color;
  }
}
```

We just created a new type `ColoredCircle` as a class that extends from `Circle`. We call `Circle` the *parent class* or *superclass* of `ColoredCircle`; and `ColoredCircle` a *subclass* of `Circle`.

We also say that `ColoredCircle` *inherits* from `Circle`, since all the public fields of `Circle` (center and radius) and public methods (like `getArea()`) are now accessible to `ColoredCircle`. Just like a parent-child relationship in real-life, however, anything private to the parent remains inaccessible to the child. This privacy veil maintains the abstraction barrier of the parent from the child, and creates a bit of a tricky situation -- technically a child `ColoredCircle` object has a center and a radius, but it has no access to it!

Line 6 of the code above introduces another keyword in Java: `super`. Here, we use `super` to call the constructor of the superclass, to initialize its center and radius (since the child has no direct access to these fields that it inherited).

The concept we have shown you is called *inheritance* and is one of the four pillars of OOP. We can think of inheritance as a model for the "*is a*" relationship between two entities.

With inheritance, we can call `coloredCircle.getArea()` without knowing or needing to know how a colored circle is represented internally and without forwarding methods.

When NOT to Use Inheritance

Inheritance tends to get overused. *In practice, we seldom use inheritance*. Let's look at some examples of how *not* to use inheritance, and why.

You may come across examples online or in books that look like the following:

```
class Point {
  private double x;
  private double y;
    :
}

class Circle extends Point {
  private double radius;
    :
}

class Cylinder extends Circle {
  private double height;
    :
}
```

The difference between these implementations and the one you have seen in Unit 9 is that it uses inheritance rather than composition.

`Circle` implemented like the above would have the center coordinate inherited from the parent (so it has three fields, x, y, and radius); `Cylinder` would have the fields corresponding to a circle, which is its base and height. In terms of modeling the properties of circle and cylinder, we have all the right properties in the right class.

When we start to consider methods encapsulated with each object, things start to break down. Consider a piece of code written as follows:

```
void foo(Circle c, Point p) {
  if (c.contains(p)) {
    // do something
  }
}
```

Since `Cylinder` is a subtype of `Point` according to the implementation above, the code above should still work also if we replace `Point` with a `Cylinder` (according to the semantic of subtyping). But it gets weird -- what is the meaning of a `Circle` (in 2D) containing a Cylinder (in 3D)? We could come up with a convoluted meaning that explains this, but it is likely not what the original implementer of `foo` expects.

The message here is this: *Use composition to model a has-a relationship; inheritance for a is-a relationship. Make sure inheritance preserves the meaning of subtyping.*

Run-Time Type

Recall that Java allows a variable of type T to hold a value from a variable of type S only if $S <: T$. Since `ColoredCircle` <: `Circle`, the following is not allowed in Java:

```
ColoredCircle c = new Circle(p, 0); // error
```

but this is OK:

```
Circle c = new ColoredCircle(p, 0, blue); // OK
```

where `p` is a `Point` object and `blue` is a `Color` object.

Also, recall that `Circle` is called the compile-time type of `c`. Here, we see that `c` is now referencing an object of subtype `ColoredCircle`. Since this assignment happens during run-time, we say that the *run-time type* of `c` is `ColoredCircle`. The distinction between these two types will be important later.