Unit 12: Overriding

After reading this unit, students should

- be aware that every class inherits from `Object`
- be familiar with the `equals` and `toString` methods
- understand what constitutes a method signature
- understand method overriding
- appreciate the power of method overriding
- understand what Java annotations are for, and know when to use `@Override`
- be exposed to the `String` class and its associated methods, especially the `+` operator

`Object` and `String`

In Java, every class that does not extend another class inherits from the class `Object` implicitly. `Object` is, therefore, the "ancestor" of all classes in Java and is at the root of the class hierarchy.

The `Object` class does not encapsulate anything in particular. It is a very general class that provides useful methods common to all objects. The two useful ones that we are going to spend time with are:

- `equals(Object obj)`, which checks if two objects are equal to each other, and
- `toString()`, which returns a string representation of the object as a `String` object.

The `toString` Method

The `toString` method is very special, as this is invoked *implicitly* by Java, by default, to convert a reference object to a `String` object during string concatenation using the operator `+`.

We showed you that in Python, `4 + "Hello"` would result in a type mismatch error. In Java, however, `4 + "Hello"` will result in the string `"4Hello"`. In this example, the primitive value 4 is converted to a string before concatenation.

A more interesting scenario is what happens if we try to concatenate, say, a `Circle` object with a string. Let's say we have:

```java
Circle c = new Circle(new Point(0, 0), 4.0);
String s = "Circle c is " + c;
```

You will see that `s` now contains the string "Circle c is Circle@1ce92674 " (the seemingly gibberish text after @` is the reference to the object and so your result will be different).

What happened here is that the `+` operator sees that one of the operands is a string but the other is not, so it converts the one that is not a string to a string by calling its `toString()` method automatically for us. This is equivalent to[1]

```java
Circle c = new Circle(new Point(0, 0), 4.0);
String s = "Circle c is " + c.toString();
```

Recall that in our `Circle` class (up to version 0.5) we do not have any `toString` method. The `toString` method that we invoked here is the `toString` method inherited from its parent `Object`.

`jshell` and `toString`

Recall that `jshell` is a REPL tool. After evaluating an expression, `jshell` prints the resulting value out. If the resulting value is a reference type, `jshell` will invoke `toString` to convert the reference type to a string first, before printing the string.

Customizing `toString` for `Circle`

The `Object::toString` method (that is our notation for the method `toString` from the class `Object`) is not very user friendly. Ideally, when we print a `Circle` object, say, for debugging, we want to see its center and its radius. To do so, we can define our own `toString` method in `Circle`. Let's upgrade our `Circle` class to do this:

```java
// version 0.6
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

  /**
```

```java
   * 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 + " }";
  }
}
```

The body of the method `toString` simply constructs a string representation for this circle object and returns it. With this `toString` implemented, the output will look something like this:

```
Circle c is { center: (0.0, 0.0), radius: 4.0 }
```

Note that when the center `this.c` is converted to a string, the `toString` method of `Point` is invoked. We leave the implementation of `Point::toString` as an exercise.

Method Overriding

What we just did is called *method overriding* in OOP. Inheritance is not only good for extending the behavior of an existing class but through method overriding, we can *alter* the behavior of an existing class as well.

Let's define the *method signature* of a method as the method name and the number, type, and order of its parameters, and the *method descriptor* as the method signature plus the return type.

When a subclass defines an instance method with the same *method descriptor* as an instance method in the parent class, we say that the instance method in the subclass *overrides* the instance method in the parent class[2]. In the example above, `Circle::toString` has overridden `Object::toString`.

The `@Override` Annotation

Line 37 in the example above contains the symbol `@Override`. This symbol is an example of *annotation* in Java. An annotation is not part of the program and does not affect the bytecode generated. Instead, it is a *hint* to the compiler. Remember that the compiler is our friend who will do its best to help detect errors early, during compilation. We must do our part to help the compiler help us. Here, `@Override` is a hint to the compiler that the following method, `toString`, is intended to override the method in the parent class. In case, there is a typo and overriding is not possible, the compiler will let us know.

It is therefore recommended and expected that all overriding methods in your code are annotated with `@Override`.

Using `super` To Access Overridden Methods

After a subclass overrides a method in the superclass, the methods that have been overridden can still be called, with the `super` keyword. For instance, the following `Circle::toString` calls `Object::toString` to prefix the string representation of the circle with `Circle@1ce92674`.

```
@Override
public String toString() {
  return super.toString() + " { center: " + this.c + ", radius: " + this.r
+ " }";
}
```

---

1. Calling `toString` explicitly is not wrong, but we usually omit the call to keep the code readable and succinct. ↩
2. It is possible to override a method in some cases when the return type is different. We will discuss this during recitations. ↩