Unit 13: Overloading

After reading this unit, students should

- understand what is overloading
- understand how to create overloaded methods

Method overloading

In the previous unit, we introduced *method overriding*. That is, when a subclass defines an instance method with the same *method descriptor* as an instance method in the parent class.

In contrast, *method overloading* is when we have two or more methods in the same class with the same name but a differing *method signature*[1]. In other words, we create an overloaded method by changing the type, order, and number of parameters of the method but keeping the method name identical.

Lets consider an `add` method which allows us to add two numbers, and returns the result. What if we would like to create an `add` method to sum up three numbers?

```
1  public int add(int x, int y) {
2    return x + y;
3  }
4
5  public int add(int x, int y, int z) {
6    return x + y;
7  }
```

In the example above, the methods `add(int, int)` and `add(int, int, int)` are overloaded. They have the same name but a different number of parameters. We can see that this allows us to write methods to handle differing inputs.

Now lets consider our `Circle` class again. Our `Circle::contains(Point)` method allows us to check if a `Point` is within the radius of the current instance of the `Circle`. We would like to create a new method `Circle::contains(double, double)` which will allow us to check if an `x` and `y` co-ordinate (another valid representation of a point) is within our circle.

```
1  import java.lang.Math;
2
```

```
class Circle {
  private Point c;
  private double r;

  public Circle(Point c, double r) {
    this.c = c;
    this.r = r;
  }

  public double getArea() {
    return Math.PI * this.r * this.r;
  }

  public boolean contains(Point p) {
    return false;
    // TODO: Left as an exercise
  }

  public boolean contains(double x, double y) {
    return false;
    // TODO: Left as an exercise
  }

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

In the above example, `Circle::contains(Point)` and `Circle::contains(double, double)` are overloaded methods.

Recall that overloading requires changing the order, number, and/or type of parameters and says nothing about the names of the parameters. Consider the example below, where we have two `contains` methods in which we swap parameter names.

```
  public boolean contains(double x, double y) {
    return false;
    // TODO: Left as an exercise
  }

  public boolean contains(double y, double x) {
    return false;
    // TODO: Left as an exercise
  }
```

These two methods have the same method signature, and therefore `contains(double, double)` and `contains(double, double)` are not distinct methods. They are not overloaded, and therefore this above example will not compile.

As it is also a method, it is possible to overload the class *constructor* as well. As in the example below, we can see an overloaded constructor which gives us a handy way to instantiate a `Circle` object that is the unit circle.

```java
class Circle {
  private Point c;
  private double r;

  public Circle(Point c, double r) {
    this.c = c;
    this.r = r;
  }

  // Overloaded constructor
  public Circle() {
    this.c = new Point(0, 0);
    this.r = 1;
  }
  :
}
```

```java
// c1 points to a new Circle object with a centre (1, 1) and a radius of 2
Circle c1 = new Circle(new Point(1, 1), 2);
// c2 points to a new Circle object with a centre (0, 0) and a radius of 1
Circle c2 = new Circle();
```

It is also possible to overload `static` class methods in the same way as instance methods. In the next unit, we will see how Java chooses which method implementation to execute when a method is invoked.

---

1. Note that this is not the same as the *method descriptor*. You can not overload a method by changing the return type. ↩