Exercise 13

Polymorphism

By the end of this exercise you will be able to

- Use *method overriding* and understand how this gives rise to the feature of *dynamic method binding*, also known as *polymorphism*.
- Understand why it is better to use polymorphism than run-time type enquiry.
- Use an `abstract` method.

Introduction

Consider the code below, in which the `Kiwi` class inherits from the `Bird` class:

```
class Bird {

   // Properties of the class...
   public boolean canFly;

   // Constructor of the class...
   public Bird() {
      canFly = true;
   }

   // Methods of the class...
   public void talk() {
      System.out.println("tweet tweet!");
   }
}

class Kiwi extends Bird {

   // Constructor of the class...
   public Kiwi() {
      canFly = false;
   }
}
```

When instances of the `Bird` class are created the value of the `canFly` property is set to `true` by the constructor, reflecting the idea that birds can fly. The constructor for the `Kiwi` class resets this property to `false` reflecting the idea that while most birds can fly, the kiwi cannot.

From this idea of resetting the value of a superclass' property in a subclass so that its value is more appropriate to the subclass, one is led to the idea of redefining a superclass' method in a subclass so that its behaviour is more appropriate to the subclass.

In Java, this is called *method overriding* and to override the `talk` method in the `Bird` class we add the following lines of code to the `Bird` class:

```
public void talk() {
   System.out.println("kiii-weee!");
}
```

With the `talk` method overridden as above, if `x` is a reference then `x.talk()` causes either `"tweet tweet!"` or `"kiii-weee!"` to be printed out, depending on whether `x` is currently pointing to a `Bird` object or to a `Kiwi` object. The decision of which `talk` method gets called is made at run time, so overriding is also called *dynamic method binding*. The more impressive sounding name of *polymorphism* is also used to refer to the same thing.

The questions of this exercise take you through the steps of reworking the classes of exercise 12 to use polymorphism rather than run-time type enquiry. You will then see the advantage of polymorphism over run-time type enquiry.

Questions

1. Finish exercise 12 if you haven't already and then load the file `ShapeTest.java` into your text editor.

2. Add a `getName` method to each of the classes `Circle`, `Triangle` and `Rectangle` which returns a `String` containing the name of the objects in the class. Here is the `Circle` class's `getName` method so you get the idea:

   ```
   public String getName() {
      return "circle";
   }
   ```

3. So you can use dynamic method binding to choose the appropriate `getName` method for each of the objects in the `myShapes` array, you will need to add an abstract `getName` method to the `Shape` class like so:

   ```
   public abstract String getName();
   ```

   The `abstract` label for a method a promise that all non-abstract subclasses will override this method. Now you can replace the `if` statements in the `printNames` method with a single call to the `getName` method:

   ```
   System.out.println(myShapes[i].getName());
   ```

4. Add a `getArea` method to each of the classes `Circle`, `Triangle` and `Rectangle` which returns a `double` containing the computed area of the objects of each class. Here is the `Circle` class's `getArea` method so you get the idea:

   ```
   public double getArea() {
      return Math.PI * radius * radius;
   }
   ```

5. So you can use dynamic method binding to choose the appropriate `getArea` method for each of the objects in the `myShapes` array, you will need to add an abstract `getArea` method to the `Shape` class like so:

```
public abstract double getArea();
```

Now you can replace the `if` statements in the `printAreas` method with a single call to the `getArea` method:

```
System.out.println(myShapes[i].getArea());
```

This completes the process of replacing run-time type enquiry with polymorphism. Now that polymorphism is used, all the computations and information about each different type of shape is stored within each subclass of `Shape`, rather than spread around classes like `ShapeTest` that use the shapes. This has the advantage that whenever you want to define a new kind of shape such as `Pentagon`, you simply write a `Pentagon` class containing all the data for the dimensions of the pentagon and the methods `getName` and `getArea` for returning the name and area of the `Pentagon` object.

With run-time type enquiry, you would have to search through all of the rest of the code that uses `Shape` objects for places that test the run-time type and modify every test to have an extra `if` statement to allow for the new class that was just defined.