Practice Assignment: W05_09-28_2

CISC 3120 Section ER6
Design and Implementation of Software Applications I

This exercise helps you review the concept of `interface` in Java. Interfaces are widely used in Java and many 3rd party Java libraries. It is arguably more often to interact with many of these library classes and methods via interfaces than to design and write your own classes that implement your own interfaces.

Since we have seen a few example programs that uses `ArrayList`, we shall use this exercise to explore the `sort` method in `ArrayList`. This exploration help you review the concept of `interface`.

The instructor provides the start-up code of this exercise at the `ShapeObjectSorter` folder at the `sampleprograms` repository on Github. For your convenience, you may browse the start-up code at,

https://github.com/CISC3120/sampleprograms/tree/master/ShapeObjectSorter

In the start-up code, you will find an abstract class, `Shape`, and two concrete subclasses of the `Shape` class, `Rectangle` and `Circle`. The `Shape` class has an abstract method `area()` that computes the area of a concrete shape. You have the following tasks:

- to implement the `area()` method in the `Rectangle` class,
- to sort a list of concrete shapes randomly generated by the `makeRandomShapes(int)` method based on *the shapes' areas*,
- and to write one or more unit tests to show that you sort the shapes correctly based on *their areas*.

Discussion

To sort an `ArrayList`, we can use the following method of `ArrayList`,

```
    public void sort(Comparator<? super E> c)
```

where `Comparator<? super E>` is an interface whose declaration you may consider as follows,

```
    public interface Comparator<? super E> {
        public int compare(E lhs, E rhs);
    }
```

where “`? super E`” represents a generic type whose super type is `E`.

According to the above understanding, we must create a concrete class that implements the `compare` method in the `Comparator` interface to sort list of `Shape` objects in an `ArrayList` using the `ArrayList`'s `sort` method. For instance, if we want to sort a list of `Shape` objects based on the length of their names, we can create a `ShapeNameComparator` class as follows,

```
package edu.cuny.brooklyn.cisc3120.ShapeObjects;

import java.util.Comparator;

```

```java
public class ShapeNameComparator implements Comparator<Shape>  {

  /*
   * lhs: Left Hand Side
   * rhs: Right Hand Side
   */
  @Override
  public int compare(Shape lhs, Shape rhs) {
    if (lhs.getName().length() > rhs.getName().length()) {
      return 1;
    } else if (lhs.getName().length() < rhs.getName().length()) {
      return -1;
    } else {
      return 0;
    }
  }

}
```

We can now examine the effect of the sorting via the following code snippet where the sorting happens at Line 6,

```java
1    ArrayList<Shape> shapeList = makeRandomShapes(5);
2    System.out.println("List␣of␣shapes␣before␣sorted␣on␣the␣length␣of␣names");
3    for(Shape s: shapeList) {
4      System.out.println(s.getName() + ":␣" + s.getName().length());
5    }
6    shapeList.sort(new ShapeNameComparator());
7    System.out.println("\nList␣of␣shapes␣after␣sorted␣on␣the␣length␣of␣names");
8    for(Shape s: shapeList) {
9      System.out.println(s.getName() + ":␣" + s.getName().length());
10   }
```

an example of whose output is,

```
List of shapes before sorted on the length of names
Circle_0: 8
Circle_1: 8
Rectangle_2: 11
Circle_3: 8
Circle_4: 8

List of shapes after sorted on the length of names
Circle_0: 8
Circle_1: 8
Circle_3: 8
Circle_4: 8
Rectangle_2: 11
```

An `JUnit 4` unit test is written as follows,

```java
package edu.cuny.brooklyn.cisc3120.ShapeObjects;

import java.util.ArrayList;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class ShapeSorterTest {
  @Test
  public void testShapeSorterByName() {
    ArrayList<Shape> shapeList = new ArrayList<Shape>();
    shapeList.add(new Circle("C123456789", 10.));
    shapeList.add(new Circle("C12345678", 8.));
    shapeList.add(new Circle("C1234567890", 11.));
    // shapeList.add(new Rectangle("C12", 10., 80.));
17
18     ArrayList<Shape> expectedShapeList = new ArrayList<Shape>();
19     expectedShapeList.add(new Rectangle(”C12”, 10., 80.));
20     expectedShapeList.add(new Circle(”C12345678”, 8.));
21     expectedShapeList.add(new Circle(”C123456789”, 10.));
22     // expectedShapeList.add(new Circle(”C1234567890”, 11.));
23
24     shapeList.sort(new ShapeNameComparator());
25     assertEquals(expectedShapeList, shapeList);
26
27   }
28 }
```

Additional Consideration

With the `Comparator` implemented in the above, the `Shape` objects are sorted in *ascending* order. How do you revise the `Comparator` to sort the objects in *descending* order? Can you also write a `Comparator` that sorts the shapes based on their names in *dictionary* order?

In addition, examine carefully how we determine whether two `Shape` objects are *equal*, and how it helps the `JUnit` unit test.

At Line 6 in the above code snippet, we instantiate an object of class `ShapeNameComparator`. It is *more than often* that the statement at the line is written *instead* as follows,

```
shapeList.sort(new Comparator<Shape>() {

  @Override
  public int compare(Shape lhs, Shape rhs) {
    if (lhs.getName().length() > rhs.getName().length()) {
      return 1;
    } else if (lhs.getName().length() < rhs.getName().length()) {
      return -1;
    } else {
      return 0;
    }
  }
});
```

where “`new Comparator<Shape>()` {...}” is *not* to instantiate an object of “`Comparator`” interface. As we discussed in class, it is not allowed to instantiate an `interface` in Java. What does it really do?

- First, it defines a class that implements the `compare` method in the `Comparator` interface. However, the class has *no name*. In Java, this is called an *anonymous* class. Do you notice the “`()`” that follows “`new Comparator<Shape>`”? This is to call the constructor of the anonymous class. Do you also notice the “{...}” that immediately follows? This is to provide the body of the anonymous class.

- Second, it instantiates an object of this *anonymous* class, and the object is the argument passed the `sort` method of the `ArrayList`.

Finally, you may also notice that the sorting algorithm implemented in the `ArrayList`'s `sort` method appears to be *stable* as it preserves the *natural order* or the *relative order* of the items with equal keys in the list.