CSCE 314: Programming Languages

Homework Assignment 1

Objective

- In this homework, you will practice using inheritance, dynamic dispatching in Java.

Submission

- This homework is will be graded as a 0, 1, or 2 for each question. (The questions are: Animals, Shapes, Generics). With 0 = inadequate or no attempt; 1 = solid effort showing a good attempt at all questions; 2 = faultless/exemplary submission. You may find it helpful to discuss your homework with the peer teachers, TAs, or instructors (say, at their office hours). The skills you develop and familiarity you attain will be vital, as the next homeworks; subsequent programming assignments will assume you have these abilities.

  Submission will be via Canvas, due: ~~20 September 2021~~ updated: 23 September 2021.

Q1: Animals

The `Animal` class represents the animals in an animal shelter, which contains only `Cat` and `Dog` currently. Each animal in the shelter has its own name and order, which can be obtained with methods `String getName()` and `int getOrder()`. Each animal can emit a cry with method `String cry()`, where for a dogs this produces "bark" while for cats is "meow".

1. Implement the abstract class `Animal`.
2. Implement the two subclasses of `Animal` to represent a cat or a dog.
3. Implement a class `AnimalShelter`, which operates with a "first in first out" basis. When an animal comes into the shelter with the method `addAnimal()`, it will be assigned an order number that represents its order in the waiting queue. People who want to adopt the animal must adopt either the "oldest" one (based on arrival time) or they can select whether they prefer a dog or a cat (and will receive the oldest animal of that type). After an animal is adopted, the order number of the remaining animals should change correspondingly. You should implement `AnimalShelter` with three different methods for adopting: `adopt()`, `adoptCat()`, `adoptDog()`. In order to check the remaining animals, cats and dogs in the shelter, you should also implement methods: `remainingAnimals()`, `remainingCats()`, `remainingDogs()`, which show the name, order and cry of remaining animals, cats and dogs.
4. Implement a class `Main`, of which users can make a selection through the options provided by the `main()` method. The main method should provide an interactive prompt as below:

   ```
   1: Add new animal
   2: Adopt an animal
   3: Adopt a cat
   4: Adopt a dog
   5: Show anaimals in the shelter
   6: Show cats in the shelter
   7: Show dogs in the shelter
   0: Exit
   Enter a number:
   ```

   The `main()` method can accept command line arguments as input, it will process the input string, construct corresponding animals and store them in a shelter. The input is given according to the

following specification:

<animals> ::= "<animal>" <animals> | ε
<animal> ::= <cat> | <dog>
<cat> ::= c <name>
<dog> ::= d <name>

This specification is given as a grammar in a structured notation called Backus–Naur form (BNF). The use of BNF is very commmon in describing program language syntax and we will see more of it. If you've not encountered BNF before, the wikipedia article provides an explanation and some simple examples to help you understand how to make sense of the specification above.

Note that <name> is a string, which we will assume doesn't contain any white space. You can assume that animals come in the same order as input.

Here is an example with a sequence of cats and dogs:

```
> java Main "d Adam" "d Bob" "d Carmy" "c Daisy"
```

Q2: Shapes

Inheritance and dynamic dispatching

The `Shape` class represents a geometric figure in some coordinate position. `Shape` allows for finding out its position and area with the methods `Point position()` and `double area()`. `Point` is a class that can represent a two-dimensional coordinate. The `Shape` class must be an abstract class, from which you will derive three subclasses: `Triangle`, `Rectangle`, and `Circle` to represent different kinds of shapes.

Pick some definition of `position()` that makes sense in order for the `equalsTo()`, which is described below, will work as we might expect. Something like centroid would be a good choice.

1. Implement the class `Point`.
2. Implement the class `Shape`.
3. Implement the three subclasses of `Shape` to represent triangles (with three points), rectangles (points for upper-left and lower-right corners), and circles (with a center and radius). Each of `Triangle`, `Rectangle`, and `Circle` should inherit from the class Shape and define `area()` appropriately.
4. Implement a class `AreaCalculator` with one static method `calculate(Shape[] shapes)` that will calculate the total area of an array of shapes.
5. Implement a class `Main`, of which the `main()` method will accept command line arguments as input. It should process the input string, constructing shapes, and storing them in an array. The input is given according to the following specification:

   <shapes> ::= "<shape>" <shapes> | ε
   <shape> ::= <triangle> | <rectangle> | <circle>
   <triangle> ::= t <point> <point> <point>
   <rectangle> ::= r <point> <point>
   <circle> ::= c <point> <number>
   <point> ::= <number> <number>
   Note that <number> is anything that Java can interpret as a number.

Here is an example with a circle, a triangle, a rectangle, and another circle:

```
> java Main "c 1.0 0.2 4.4" "t 0.4 5.3 0.4 5.6 3.0 1.2" "r 3.0 -3.4 2.3 0.0" "c 0 0 10."
The total area for the 4 objects is 377.75 units squared.
```

Your main should produce the preceding output via the following code:

```
public static void main(String args[]) {
Shape shape[] = new Shape[args.length];


/* Some initialization from the args ... */


System.out.printf("The total area for the %d objects is %1.2f units squared.\n", shape.length, AreaCalc

}
```

Defining equality

We say that two different shapes to be equal if and only if (i) they are of the same kind, (ii) their position is the same, and (iii) the geometric figures they represent are equal (i.e., the two shapes are congruent). The textbook (Section 3.8) discusses implementing equality.

1. Implement the `equals` method for `Shape` and all of its derived classes.
2. Override `hashCode` for these classes (as you should whenever you override an Object's `equals` method).

Comparison

Two shapes can be compared based on area, so that shape `A` is less than or equal to shape `B` if and only if `A`'s area is less than or equal to `B`'s area.

1. Make this ordering the *natural ordering* of `Shape` and all its derived classes. (Read sections 4.1 and 21.3, which discuss natural orderings and making classes comparable).
2. Extend the implementation of your `Main.main` so that it also prints out the shapes in an increasing order according to your natural ordering. To be able to print out shapes, add the `toString` method to each of the shape classes.

```
> java Main "c 1.0 0.2 4.4" "t 0.4 5.3 1.0 3.9 4.0 2.0" "r 3.0 -3.4 2.3 0.0" "c 0 0 10."
1) Triangle (0.4, 5.3)-(1.0, 3.9)-(4.0, 2.0)             area=1.5299999999999998
2) Rectangle (2.3, -3.4)-(3.0, 0.0)              area=2.3800000000000003
3) Circle (1.0, 0.2), radius = 4.4               area=60.821233773498406
4) Circle (0.0, 0.0), radius = 10.0              area=314.1592653589793
```

That output should come from the following code:

```
public static void main(String args[]) {
        Shape shape[] = new Shape[args.length];


        /* Some initialization from the args ... */


        Arrays.sort(shape);
        int count = 0;
        for (Shape s: shape) {
            System.out.println(++count + ") "+s+"\t\t area="+s.area());
        }
}
```

Q3: Generics

A linked list of shapes via generics

The following `Node` class can represent a singly-linked list of shapes.

```
public final class Node<T extends Shape> {
    public final T v;
    public Node<T> next;
    public Node (T val, Node<T> link) { v = val; next = link; }
}
```

1. Define a class `NodeIterator<T>` to iterate over the values stored in a linked list of `Node<T>` objects. The constructor of that class should take a `Node<T>` as a parameter, and thus have the header:
   `public NodeIterator(Node<T> n)`
   Your `NodeIterator<T>` class must implement the `java.util.Iterator<T>` interface (see the Iterator API docs).
2. Now make `Node<T>` *iterable*; see the API docs.
3. Now, if `list` is of type `Node<T>`, you should be able to iterate over list using Java's "for each" for-loop:

   ```
   for (T e : list) { /* do something with e */ }
   ```

   Now modify your `class Main` so that contains the static methods:
   1. `maxArea()` that accepts a linked list of type `Node<Shape>` and returns the shape with the largest area within in the linked list, and
   2. `boundingRect()` that accepts a linked list of type `Node<Rectangle>` and computes the smallest single rectangle that surrounds (or encompasses) all the rectangles in the list.

A better linked list

You may have noticed from the previous problem that it is rather inconvenient to build lists with `Node`. Implement another generic class `ShapeList<T>` that has a nicer interface.

1. Make `ShapeList<T>` iterable by implementing the `Iterable<T>` interface.
2. Define the two constructors for the `ShapeList<T>` class:
   1. `public ShapeList(); // create an empty list`
   2. `public ShapeList(Iterable<T> iterable);`
3. Implement these member methods (with their expected meaning):
   1. `public ShapeList<T> reverse();`
   2. `public String toString();`

   A call `x.reverse()` should reverse `x`, and return the reversed `x` as the result. The `toString()` method should match the examples given below.

```
> java Main "c 1.0 0.2 4.4" "t 0.4 5.3 0.4 5.6 3.0 1.2" "r 3.0 -3.4 2.3 0.0" "c 0 0 10."
emptyShapes = []
reversed emptyShapes = []
someCircles = [{Circle (1.0, 1.0), radius = 1.0}, {Circle (2.0, 2.0), radius = 2.0}, {Circle (3.0, 3.0)
reversed someCircles = [{Circle (3.0, 3.0), radius = 3.0}, {Circle (2.0, 2.0), radius = 2.0}, {Circle (
Some of Xs = 6.0
Some of Ys = 6.0
```

```
public static void main(String args[]) {
        Shape shape[] = new Shape[args.length];

        /* Some initialization from the args ... */

        Circle c1 = new Circle(1,1,1);
        Circle d1 = new Circle(2,2,2);
        Circle e1 = new Circle(3,3,3);
```

```
        ShapeList<Shape> emptyShapes = new ShapeList<Shape>();
        ShapeList<Circle> someCircles = new ShapeList<Circle>(Arrays.asList(c1, d1, e1));

        System.out.println("emptyShapes = " + emptyShapes);
        System.out.println("reversed emptyShapes = " + emptyShapes.reverse());
        System.out.println("someCircles = " + someCircles);
        System.out.println("reversed someCircles = " + someCircles.reverse());

        double sumOfXs = 0.0;
        double sumOfYs = 0.0;
        for (Circle c: someCircles) {
            sumOfXs += c.position().x;
            sumOfYs += c.position().y;
        }
        System.out.println("Some of Xs = " + sumOfXs);
        System.out.println("Some of Ys = " + sumOfYs);
}
```

Acknowledgements

Animals are adapted from "Cracking the coding interview" chapter 3.
Shapes and Generics is adapted from Dr. Shell's previous assignments (originally dating to the 2016 Spring semester).