

Basis of Computer Programming (java A)

Tutorial 12

(The original document was designed by ZHAO Yao in 2018/12, and given simple modified by ZHU Yueming in 2019/5)

[Experimental Objective]

- Learn abstract class.
- Learn how to define and implement an interface.
- Learn how to use the interface `java.lang.Comparable<T>`.

[Before Exercises]

1. Inheritance exercise.

Step1: Download **Lab11.zip** from sakai which is the source code of the last exercise. We should find that:

1. There are two public methods, which have no valid code.

```
public void checkColor() {  
  
  
public void draw() {  
}
```
2. The most important thing is that we have no need to instantiate **Shape**. In this case, we should change the **Shape** class to an abstract class.
(1) Add "abstract" before "class":

```
public abstract class Shape
```


(2) Change `draw()` to abstract method:

```
abstract public void draw();
```

Step2: In **ShapeTest**, we try to write this following code in `main()`:

```
Shape shape=new Shape();
```

There will be an error: Cannot instantiate the type Shape

Step3: Learn how to use Comparable interface.

Suppose we have several circles, and for each of them has different radius. We want to sort them from big to small according radius. How can we do?

Comparable is very useful. This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the `compareTo` method in it is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`).

(1) Let **Circle** implements **Comparable**

```
public class Circle extends Shape implements  
Comparable<Circle>
```

After implements the interface Comparable, it arise a mistake that if a class implements an interface, it must override all abstract methods in it.

(2) Override the method **compareTo** defined in **Comparable**.

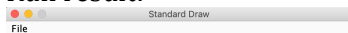
```
@Override
public int compareTo(Circle o) {
    if(this.radius < o.radius){
        return 1;
    }else if(this.radius > o.radius){
        return -1;
    }
    return 0;
}
```

Normally, this method compares current object with the parameter object to determine a sort order. The return value of the method can be a negative integer, zero, or a positive integer, which means current object would in former place, all equal, or in latter place than parameter respectively. However, in this case, we want to sort the Circles in descending order of its radius, so that when the radius of current object is less than the parameter object, the return value would be 1(a positive integer).

(3) Rewrite the **ShapeTest**, using the following code:

```
public static void main(String[] args) {
    List<Circle> circleList = new ArrayList<>();
    Circle.setScreenSize(14);
    StdDraw.setScale(-Shape.getScreenSize(),
        Shape.getScreenSize());
    for (int i = 0; i < Shape.getScreenSize(); i += 2) {
        circleList.add(new Circle(i, 0, -Shape.getScreenSize()));
    }
    Collections.sort(circleList);
    for (int i = 0; i < circleList.size(); i++) {
        circleList.get(i).setColor(ShapeColor.values()[i%3]);
        circleList.get(i).draw();
    }
}
```

Run result:



Step4: In **Step3**, we can see that the color scheme is not beautiful and the **ShapeColor** is mainly used to check if the shape is in the boundary. We can define an interface named **ColorDraw**, in which declare an abstract method **customizedColor**.

(1) Define an enum class **ColorScheme**:

```
public enum ColorScheme {
    SKY(new Color[]{new Color(0, 102, 204),
        new Color(0, 128, 255),
        new Color(51, 153, 255),
        new Color(102, 178, 255),
        new Color(153, 204, 255),
        new Color(204, 229, 255)}),
    RAINBOW(new Color[]{
        Color.RED,
        Color.ORANGE,
        Color.YELLOW,
        Color.GREEN,
        Color.CYAN,
        new Color(0, 128, 255),
        new Color(204, 153, 255)}),
    GRAY(new Color[]{
        Color.DARK_GRAY,
        Color.GRAY,
        Color.LIGHT_GRAY});

    Color[] colorList;

    private ColorScheme(Color[] color) {
        colorList = color;
    }

    public Color[] getColorScheme() {
        return colorList;
    }
}
```

(2) Define an interface:

```
public interface ColorDraw {
    public void customizedColor(ColorScheme colorScheme, int
index);
}
```

(3) Implements the **ColorDraw** in **Circle**

```
public class Circle extends Shape implements
Comparable<Circle>, ColorDraw

@Override
public void customizedColor(ColorScheme colorScheme, int index)
{
    Color[] colorList = colorScheme.getColorScheme();
    if (index < 0){
        index = 0;
    }
    if (index >= colorList.length){
        index = index % colorList.length;
    }
    StdDraw.setPenColor(colorList[index]);
    StdDraw.filledCircle(x, y, radius);
}
```

(4) In **ShapeTest**, we change main method to the following code:

```
List<Circle> circleList = new ArrayList<Circle>();
Shape.setScreenSize(14);
StdDraw.setScale(-Shape.getScreenSize(),
Shape.getScreenSize());

for (int i = 1; i < Shape.getScreenSize(); i += 2) {
    circleList.add(new Circle(i, 0, -Shape.getScreenSize()));
}

Collections.sort(circleList);

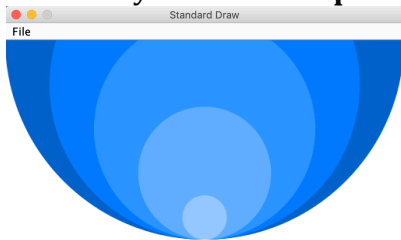
for (int i = 0; i < circleList.size(); i++) {
    circleList.get(i).customizedColor(ColorScheme.RAINBOW, i);
}
```

Run result:



[Exercises]

1. Modify the class **ShapeTest**, draw some circles like the following image:



Hint: ColorScheme.Sky

x	y	radius
0	1	1
0	3	3
0	5	5
0	7	7

0	9	9
---	---	---

screen size = 9

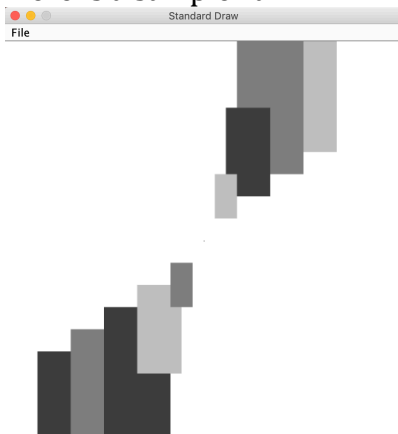
2. Modify the class **Rectangle** in **JavaLab10.zip**.
 - a. Make **Rectangle** implements **Comparable**, override the method **compareTo** to order the rectangles from the largest to smallest according their area. If two rectangles have the same area, order the rectangles from smallest to largest according **x**.
 - b. Make **Rectangle** implements **ColorDraw**, override the method **customizedColor** to draw the rectangle according to the specific **ColorScheme** and the index.
3. Create a class **RectangleTest** for test.

```
public class RectangleTest {
    public static void main(String[] args) {
        Shape.setScreenSize(9);
        StdDraw.setScale(-Shape.getScreenSize(),
Shape.getScreenSize());

        List<Rectangle> rectanglList = new ArrayList<Rectangle>();
        for (int i = -5; i < 5; i++) {
            rectanglList.add(new Rectangle(i, 2*i, Math.abs(i),
2*Math.abs(i)));
        }
        Collections.sort(rectanglList);

        for (int i = 0; i < rectanglList.size(); i++) {
            rectanglList.get(i).customizedColor(ColorScheme.GRAY, i);
            System.out.println(rectanglList.get(i));
        }
    }
}
```

Here is a sample run:



4. You can design yourself pattern that contains circles and rectangles, or other yourself defined shapes.