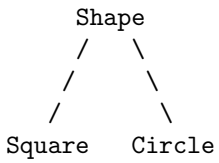
MODULE 7q - Redeeming Class Shape

SQUARES AND CIRCLES REVISITED

It was noted earlier that the examples which exploited class Shape and its children manifested poor style. The examples were all variants of the program ShapeA (A for Awful!).

It is almost time to put matters right and consider a sequence of variants ShapeB (B for Better!).

It is convenient to consider a final variation of the ShapeA program which largely restores an earlier version in which there are no interfaces but the single abstract class Shape. The inheritance diagram is:



In the fourth (most recent) variation of the ShapeA program there were two interfaces Shape and Sortable but no abstract classes. In the final version, shown below, these two interfaces are combined into one abstract class (a reincarnation of Shape) despite not containing any data fields or non-abstract classes.

Accordingly, the restoration is not complete. The abstract class Shape once again incorporates the abstract methods perimeter(), area() and compare() but now has no data fields or non-abstract classes. The only reason for this partial restoration is to keep the illustrative programs to a manageable length.

In the absence of the interface Sortable, it is no longer possible to use Sortable as a type. In consequence the polymorphic array once again has to be a Shape array and all other references to type Sortable will need changing back to Shape.

These and all other changes to the previous version of the ShapeA program are indicated by comments. Set up this version now.

```
public class ShapeA
 { public static void main(String[] args)
    { Shape[] sa = {new Square(2d),                  // back to type Shape
                    new Square(3d),
                    new Circle(1.5d)};
      printOut(sa);
      sort(sa);
      printOut(sa);
    }

   private static void printOut(Shape[] s)           // type Shape
```

```
    { for (int i=0; i<s.length; i++)
         System.out.printf("sa[%d]: %s%n", i, s[i]);
    }

   private static void sort(Shape[] s)                // type Shape
    { for (int k=1; k<s.length; k++)
       { int i=k;
         while (i>0 && s[i-1].compare(s[i]))
          { Shape t = s[i-1];                         // type Shape
            s[i-1] = s[i];
            s[i] = t;
            i--;
          }
       }
    }
 }

abstract class Shape                                  // back to an abstract class
 { public abstract double perimeter();

   public abstract double area();

   public abstract boolean compare(Shape that);       // retrieved from interface
 }                                                    // Sortable; uses type Shape

                                                      // interface Sortable removed

class Square extends Shape                            // extends Shape
 { public double side;

   public Square(double side)
    { this.side = side;
    }

   public double perimeter()
    { return 4d*this.side;
    }

   public double area()
    { return this.side*this.side;
    }

   public boolean compare(Shape that)                 // type Shape
    { return this.side > (that instanceof Square
                         ? ((Square)that).side
                         : ((Circle)that).radius);
    }

   public String toString()
    { return String.format("  Square -%n" +
                           "  Side is %.2f%n" +
                           "  Perimeter is %.2f%n" +
                           "  Area is %.2f%n",
                           this.side, this.perimeter(), this.area());
    }
 }
```

```
class Circle extends Shape                              // extends Shape
 { public double radius;

   public Circle(double radius)
    { this.radius = radius;
    }

   public double perimeter()
    { return 2d*Math.PI*this.radius;
    }

   public double area()
    { return Math.PI*this.radius*this.radius;
    }

   public boolean compare(Shape that)                   // type Shape
    { return this.radius > (that instanceof Circle
                            ? ((Circle)that).radius
                            : ((Square)that).side);
    }

   public String toString()
    { return String.format("  Circle -%n" +
                           "  Radius is %.2f%n" +
                           "  Circumference is %.2f%n" +
                           "  Area is %.2f%n",
                              this.radius, this.perimeter(), this.area());
    }
 }
```

TRY IT OUT

Compile and run the program. The output should be:

```
sa[0]:   Square -
  Side is 2.00
  Perimeter is 8.00
  Area is 4.00

sa[1]:   Square -
  Side is 3.00
  Perimeter is 12.00
  Area is 9.00

sa[2]:   Circle -
  Radius is 1.50
  Circumference is 9.42
  Area is 7.07

sa[0]:   Circle -
  Radius is 1.50
  Circumference is 9.42
  Area is 7.07
```

```
sa[1]:   Square -
  Side is 2.00
  Perimeter is 8.00
  Area is 4.00

sa[2]:   Square -
  Side is 3.00
  Perimeter is 12.00
  Area is 9.00
```

AN IMPROVEMENT - INTRODUCTION

The critical remarks about the ShapeA examples began when the abstract method compare() was introduced into class Shape and code was duplicated in the two child classes.

A simple fix was suggested but not implemented because, in the subsequent variations of ShapeA, sides of squares were to be compared with radii of circles. These variations eliminated the duplication but the style was deemed atrocious! Four particular complaints have been noted:

1. As a name for a method compare() is not very satisfactory.

2. Duplicating code is usually bad news.

3. Using the type of an item as a means of deciding which variant of an operation to perform is very bad practice.

4. Requiring data fields side and radius to be declared public is bad form.

It is simple enough to attend to the first complaint. Simply rename the method greaterThan().

The second complaint refers to the earlier duplication of the method compare() when areas were to be compared. The simple fix is to incorporate the duplicated method in class Shape as a non-abstract method. If the current variation of the ShapeA program (in which class Shape is rather cut down from its original version) were to incorporate this fix, the abstract class Shape would appear thus:

```
abstract class Shape
 { public abstract double perimeter();

   public abstract double area();

   public boolean greaterThan(Shape that)     // non-abstract method
    { return this.area() > that.area();
    }
 }
```

The method name greaterThan() has been used as suggested [in place of compare()] and is written as a non-abstract method alongside the abstract perimeter() and area() methods.

This approach won't work if the side of a Square is to be compared with the radius of a Circle and it was the earlier attempt to deal with that difficulty that prompted the third complaint.

One way to attend to the third complaint is to introduce another abstract method rank() which each child class can override to specify just which observable is to be used for ranking.

The rank() methods in both Square and Circle could return this.area() or that in Square could return this.side and that in Circle could return this.radius. The greaterThan() method could then compare the general-purpose rank and could CONTINUE to be in class Shape.

The fourth complaint could have been attended to by introducing a public getSide() method in class Square and a public getRadius() method in class Circle but the device of using the rank() methods makes this approach unnecessary. Access to the data fields side and radius will now be via the public rank() methods of Square and Circle so these data fields can once again be private.

AN IMPROVEMENT - PROGRAM

The initial ShapeB program, which is shown overleaf, incorporates these suggestions. Shape incorporates the new abstract method rank() as well as the renamed compare() method, now called greaterThan(). The interface Sortable has not been restored.

The two child classes each include an appropriate rank() method and happen to specify side and radius respectively. Neither calls the greaterThan() method which is accessed only from sort(). Notice also the call of greaterThan() [rather than compare()] in the while-statement of the sort() method.

To prepare this program, first copy ShapeA.java to ShapeB.java and then make all the amendments indicated by comments.

Set up this version now.

```
public class ShapeB                                    // new name
 { public static void main(String[] args)
    { Shape[] sa = {new Square(2d),
                    new Square(3d),
                    new Circle(1.5d)};
      printOut(sa);
      sort(sa);
      printOut(sa);
    }

   private static void printOut(Shape[] s)
    { for (int i=0; i<s.length; i++)
         System.out.printf("sa[%d]: %s%n", i, s[i]);
    }
```

```
   private static void sort(Shape[] s)
    { for (int k=1; k<s.length; k++)
       { int i=k;
         while (i>0 && s[i-1].greaterThan(s[i]))      // greaterThan()
          { Shape t = s[i-1];
            s[i-1] = s[i];
            s[i] = t;
            i--;
          }
       }
    }
 }

abstract class Shape
 { public abstract double perimeter();

   public abstract double area();

   public abstract double rank();                     // new abstract method

   public boolean greaterThan(Shape that)             // non-abstract method
    { return this.rank() > that.rank();               // which uses rank()
    }
 }

class Square extends Shape
 { private double side;                               // private again

   public Square(double side)
    { this.side = side;
    }

   public double perimeter()
    { return 4d*this.side;
    }

   public double area()
    { return this.side*this.side;
    }

   public double rank()                               // method rank() added
    { return this.side;
    }
                                                      // method compare removed

   public String toString()
    { return String.format("  Square -%n" +
                           "  Side is %.2f%n" +
                           "  Perimeter is %.2f%n" +
                           "  Area is %.2f%n",
                              this.side, this.perimeter(), this.area());
    }
 }

class Circle extends Shape
```

```
 { private double radius;                                  // private again

   public Circle(double radius)
    { this.radius = radius;
    }

   public double perimeter()
    { return 2d*Math.PI*this.radius;
    }

   public double area()
    { return Math.PI*this.radius*this.radius;
    }

   public double rank()                                    // method rank() added
    { return this.radius;
    }
                                                           // method compare removed

   public String toString()
    { return String.format("  Circle -%n" +
                           "  Radius is %.2f%n" +
                           "  Circumference is %.2f%n" +
                           "  Area is %.2f%n",
                              this.radius, this.perimeter(), this.area());
    }
 }
```

This is a great improvement. In particular the inelegant uses of instanceof in the compare() methods have gone. Every new Shape descendant defines its own chosen feature as the rank() and all is well. If a third Shape is introduced, say a Triangle, no changes need be made to the rank() methods in Square or Circle.

TRY IT OUT

Compile and run the program. The output should be as before.

A FIRST VARIATION - INTRODUCTION

A problem occurs when an attempt is made to split the abstract class Shape into two. It would be nice to have two classes such as:

```
abstract class Shape
 { public abstract double perimeter();                     // methods concerned
                                                           // with shapes
   public abstract double area();
 }

abstract class Sortable
 { public abstract double rank();                          // methods concerned
                                                           // with sorting
   public boolean greaterThan(Sortable that)
```

```
    { return this.rank() > that.rank();
    }
 }
```

This attempt fails because multiple inheritance is not permitted in Java and classes Square and Circle cannot have both Shape and Sortable as parents.

The previous solution was to keep Shape as an abstract class and make Sortable an interface but this is ruled out now because greaterThan() is not an abstract method. Accordingly, Sortable cannot be an interface.

It happens that class Shape contains only abstract classes so one solution would be to convert Shape to an interface. This, though, would not be possible if Shape had been fully restored and incorporated the data field name and associated non-abstract methods.

The question arises: what can be done if both Shape and Sortable are to contain non-abstract classes AND both are to be inherited by child classes such as Square and Circle?

The problem could be solved by duplicating code. Remove the greaterThan() method from Sortable and put a copy in each child class but this is what we have been trying to avoid!

A standard solution is to have what is called a 'helper class' (shown as SortableHelp in the program overleaf). A helper class is a class associated with an interface and contains the non-abstract methods that the interface is not permitted to contain. These non-abstract methods are declared static since they are to be used as class methods rather than as instance methods.

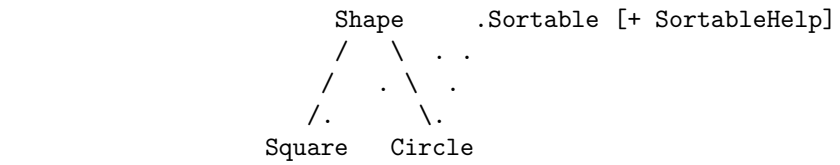
This is not the ideal solution since it still involves duplication. The idea is that every class implementing Sortable is required to have a greaterThan() method which uses the implementation of greaterThan() in SortableHelp.

Although regrettable, this duplication is less worrying than it ordinarily would be. The duplicated code will generally consist of a method with a one-line body which need never be changed. The (potentially) serious code which might need maintenance appears only once as the body of the method in the helper class.

A FIRST VARIATION - PROGRAM

In the first variation of the ShapeB program, shown below, the abstract class Shape is kept as an abstract class (to allow the possibility of restoring the data field name and the associated non-abstract classes).

The methods rank() and greaterThan() have been removed from class Shape and incorporated into interface Sortable, with greaterThan() now being an abstract method. The inheritance diagram is:



The helper class SortableHelp is shown associated with but independent of and separate from the interface Sortable.

Whenever the while-statement calls greaterThan() it invokes either the greaterThan() method in class Square or the greaterThan() method in class Circle. Each of these duplicated methods has a single formal argument that in its heading but, in its body, has to hand over both this AND that to the greaterThan() method in the helper class.

Notice the name of the helper class in front of the dot in the call SortableHelp.greaterThan(this, that) indicating that the method is a class method (and not an instance method) and has to be declared static. The helper class SortableHelp is never instantiated.

The two actual arguments this and that in the call are associated with the two formal arguments s1 and s2 in the greaterThan() method in the helper class.

Given the reappearance of the interface Sortable, it is again possible to use Sortable as a type. In consequence the polymorphic array is once again a Sortable array and various other references to type Shape will need changing back to Sortable.

These and all other changes from the original version are indicated by comments. Set up this version now.

```
public class ShapeB
 { public static void main(String[] args)
    { Sortable[] sa = {new Square(2d),               // back to type Sortable
                       new Square(3d),
                       new Circle(1.5d)};
      printOut(sa);
      sort(sa);
      printOut(sa);
    }

   private static void printOut(Sortable[] s)         // type Sortable
    { for (int i=0; i<s.length; i++)
         System.out.printf("sa[%d]: %s%n", i, s[i]);
    }

   private static void sort(Sortable[] s)             // type Sortable
    { for (int k=1; k<s.length; k++)
       { int i=k;
         while (i>0 && s[i-1].greaterThan(s[i]))
          { Sortable t = s[i-1];                      // type Sortable
            s[i-1] = s[i];
```

```
            s[i] = t;
            i--;
          }
       }
    }
 }

abstract class Shape                                    // two methods removed
 { public abstract double perimeter();

   public abstract double area();
 }


interface Sortable                                      // restored interface
 { public abstract double rank();

   public abstract boolean greaterThan(Sortable that);// now abstract and has
 }                                                      // argument type Sortable

class SortableHelp                                      // helper class containing...
 { public static boolean greaterThan(Sortable s1, Sortable s2)
    { return s1.rank() > s2.rank();
    }                                                   // ...once-off code which
 }                                                      // could be ambitious

class Square extends Shape implements Sortable         // implements Sortable
 { private double side;

   public Square(double side)
    { this.side = side;
    }

   public double perimeter()
    { return 4d*this.side;
    }

   public double area()
    { return this.side*this.side;
    }

   public double rank()
    { return this.side;
    }

   public boolean greaterThan(Sortable that)           // duplicate code
    { return SortableHelp.greaterThan(this, that);     // one line body
    }                                                   // never changed

   public String toString()
    { return String.format("  Square -%n" +
                           "  Side is %.2f%n" +
                           "  Perimeter is %.2f%n" +
                           "  Area is %.2f%n",
                              this.side, this.perimeter(), this.area());
    }
```

```
 }

class Circle extends Shape implements Sortable        // implements Sortable
 { private double radius;

   public Circle(double radius)
    { this.radius = radius;
    }

   public double perimeter()
    { return 2d*Math.PI*this.radius;
    }

   public double area()
    { return Math.PI*this.radius*this.radius;
    }

   public double rank()
    { return this.radius;
    }

   public boolean greaterThan(Sortable that)          // duplicate code
    { return SortableHelp.greaterThan(this, that);    // one line body
    }                                                 // never changed

   public String toString()
    { return String.format("  Circle -%n" +
                           "  Radius is %.2f%n" +
                           "  Circumference is %.2f%n" +
                           "  Area is %.2f%n",
                              this.radius, this.perimeter(), this.area());
    }
 }
```

TRY IT OUT

Compile and run the program. The output should be as before.