MODULE 9q - Class Shape Concluded

A THIRD VARIATION - INTRODUCTION

The inheritance diagram associated with the most recent version of the ShapeB program was given as:

```
           Shape     .Sortable [+ SortableHelp]
           /   \  . .
          /   . \  .
         /.      \.
     Square   Circle
```

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

The interface Sortable specified an abstract greaterThan() method which had to be duplicated in Square and Circle. The duplicated method couldn't be in Sortable itself since an interface is not allowed to incorporate a non-abstract method.

The saving grace for the duplicated greaterThan() method was that it had a body of just one line. It invoked the more serious once-off greaterThan() method in the helper class SortableHelp. If a new release required extra code this would go in the once-off method rather than in the duplicated ones.

An obvious variation on the theme is to revise the inheritance diagram so that interface Sortable is at a higher level than class Shape:

```
            Sortable
                .
               .
              .
           Shape
           /   \
          /     \
         /       \
     Square   Circle
```

This reflects a common-sense view that 'being Sortable' is more generic than 'having Shape'. Many classes of object may be sorted (given an appropriate criterion for sorting) but rather fewer objects merit the attribute Shape.

Much more is gained by using this extra level than merely satisfying common sense. In particular, the undesired duplication can be avoided AND the helper class can be eliminated...

Although Shape is an abstract class it couldn't previously accommodate an appropriate non-abstract greaterThan() method because such a method would have an argument of type Sortable, and class Shape could have no knowledge of Sortable. It is now proposed that Shape should inherit from Sortable so class Shape will know about Sortable and can incorporate an appropriate non-abstract greaterThan() method.

This greaterThan() method is once-off and can be seriously ambitious if required. It is inherited by any child class so there is no need to refer to it in Square or Circle.

The strategy just suggested is followed in the third variation of the ShapeB program, shown overleaf. All changes to the previous version are indicated by comments.

Notice that the abstract method greaterThan() is still specified in the interface Sortable but that a non-abstract implementation of greaterThan() now appears in class Shape. There are no longer greaterThan() methods in class Square or class Circle.

An obvious comment is that, since there is now no attempt at multiple inheritance, there is no need for Sortable to be an interface. It could perfectly well be an abstract class, at least in this case. Observe that inheritance from a grandparent through a parent does not count as multiple inheritance.

Although Sortable could indeed be an abstract class it has been left as an interface to allow for possible generalisation later. If Sortable were an abstract class no new class could inherit both from it AND some other yet-to-be-designed class. Leaving Sortable as an interface permits any new class to implement Sortable AND inherit from some other class.

In summary, just because class Shape does not wish to inherit from anywhere except Sortable, there is no reason to impose this restriction on other classes.

A THIRD VARIATION - PROGRAM

Set up this version now.

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

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

   private static void sort(Sortable[] s)
    { 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];
            s[i-1] = s[i];
            s[i] = t;
            i--;
          }
       }
    }
 }

interface Sortable                                  // unchanged interface Sortable
 { public abstract double rank();                   // placed before abstract class
                                                    // Shape to indicate that it is
   public abstract boolean greaterThan(Sortable that);// higher in the inheritance
 }                                                  // diagram

abstract class Shape implements Sortable            // Shape implements Sortable
 { public abstract double perimeter();

   public abstract double area();

   public boolean greaterThan(Sortable that)        // Once-off non-abstract
    { return this.rank() > that.rank();             // greaterThan() incorporated
    }
 }
                                                    // helper class removed

class Square extends Shape                          // no longer 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;
    }
                                                    // greaterThan() 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                          // no longer 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;
    }
                                                         // greaterThan() 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());
    }
 }
```

TRY IT OUT

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

```
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
```

A FOURTH VARIATION - INTRODUCTION

The next level of interest centres on entities which are sortable (once a suitable criterion is defined) but cannot be described by a rank consisting of a single number (of type double).

For example, one may want to have something described by a String or a complex number or a vector or something else altogether.

The interface Sortable used above is not sufficiently general because a String (say) cannot readily be converted to type double, though it clearly ought to be sortable.

This shows that specifying the type of rank in interface Sortable is too restrictive. It is better to specify only greaterThan() and leave the rank (or equivalent) out. This still leaves the crucial question of how to avoid writing a truly sinful statement involving instanceof (or some other equivalent contrivance) in the implementation of greaterThan().

The point is that the criterion for comparing Strings is different from the criterion from comparing Shapes, and both are different from the criterion for comparing complex numbers.

Happily, and significantly, one can sensibly compare only Strings with Strings and Shapes with Shapes and so on, but not MIXED pairs of (say) a String and a complex number.

For a given 'family' of Sortable things (that is a set of types that one can happily mix in a polymorphic array AND can expect to be sorted), there ought to be ONE specific greaterThan() algorithm, which can operate regardless of the specific type of the underlying object.

All this can be achieved by having four levels of inheritance:

1. At the top is interface Sortable which specifies the single abstract method greaterThan(), just to make the point that something is Sortable if a binary ordering relation is defined on its members.

2. Next come generic interfaces which group together families of items which may be sorted by the same criterion. Each such interface would specify the method or methods needed for sorting. For example any items which can be sorted once a value of type double is determined (like Shapes) could be required to supply a rank() method.

3. Next, for every head-of-family such as Shape, String, complex number, etc. (that is things that share the same comparison algorithm), provide a class with the appropriate implementation of the greaterThan() method.

4. Finally, provide the child classes such as Square and Circle.

An appropriate illustrative inheritance diagram might be:

```
                                 Sortable
                                .        .
                              .            .
                            .                .
                 SortableForLengths   SortableForComplex
                    .
                   .
                  .
                Shape
                /   \
               /     \
              /       \
           Square   Circle
```

A FOURTH VARIATION - PROGRAM

Much of the above discussion is illustrated in the fourth variation of the ShapeB program, shown on the facing page. All changes to the previous version are indicated by comments.

This version introduces the intermediate SortableForLengths interface but not the SortableForComplex interface. Notice that SortableForLengths extends Sortable. (It doesn't implement Sortable.)

The abstract class Shape implements interface SortableForLengths, which in turn extends interface Sortable, so it has to provide a greaterThan() method whose argument that is of type Sortable (because an argument of type Sortable for greaterThan() is dictated by the abstract method greaterThan() in the top-level interface Sortable).

This presents a problem. Since that is of type Sortable that.rank() cannot be used because rank() is unknown in interface Sortable. The solution is to provide a cast, (SortableForLengths), as shown. This cast is permitted because interface SortableForLengths extends Sortable. Moreover, since SortableForLengths includes rank() as an abstract method, ((SortableForLengths)that).rank() is permitted too and the problem is solved.

It is of minor note that a second level of bracketing is needed with the cast, thus ((SortableForLengths)that).rank() works whereas (SortableForLengths)that.rank() will not.

No cast is needed with this.rank() because this is of type Shape which implements SortableForLengths and therefore knows about rank().

Set up this version now.

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

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

   private static void sort(Sortable[] s)
    { 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];
            s[i-1] = s[i];
            s[i] = t;
            i--;
          }
       }
    }
 }

interface Sortable                                    // top-level interface Sortable
 { public abstract boolean greaterThan(Sortable that);// now specifies greaterThan()
 }                                                    // and nothing else

interface SortableForLengths extends Sortable         // intermediate interface
 { public abstract double rank();                     // extends Sortable and
 }                                                    // specifies rank()

abstract class Shape implements SortableForLengths    // implements SortableForLengths
 { public abstract double perimeter();

   public abstract double area();

   public boolean greaterThan(Sortable that)
```

```
    { return this.rank() > ((SortableForLengths)that).rank();
                // cast here ^^^^^^^^^^^^^^^^^^^^
    }
 }

class Square extends Shape
 { 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 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;

   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 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.

A FIFTH VARIATION - INTRODUCTION

To illustrate the right-hand side of the inheritance diagram one needs to provide an interface SortableForComplex. A possible interpretation of a complex number is as a Vector and one use of a Vector is to describe the components of tidal flow at a particular place.

Here is an augmentation of the inheritance diagram that applies to the final variation of the current program:

```
                            Sortable
                           .        .
                         .            .
                       .                .
             SortableForLengths   SortableForComplex
                   .                          .
                  .                            .
                 .                              .
                Shape                          Vector
                /   \                          /
               /     \                        /
              /       \                      /
           Square   Circle                Tide
```

A FIFTH VARIATION - PROGRAM

The fifth variation of the ShapeB program, shown on the facing page, incorporates an interface SortableForComplex and an interface Vector as well as a representative child class Tide.

Note that three casts to type SortableForComplex are employed in the greaterThan() methods in class Vector.

No changes have been made to any aspect of class Shape or its child classes except that in method main() the array name has been changed to sal (sortable array dealing with lengths).

The printOut() method refers generically to elements of array sa rather than specifically to sal or sac.

A new array sac (sortable array dealing with complex) has been set up too with three elements each of type Tide. The Tide is specified as pair of values of type double, one showing the northing component of the tide (in knots) and the other showing the easting component of the tide (in knots).

All changes to the previous version are indicated by comments.

Set up this version now.

```
public class ShapeB
 { public static void main(String[] args)
    { Sortable[] sal = {new Square(2d),                // Array  sal  is of type
                        new Square(3d),                // sortable but all elements
                        new Circle(1.5d)};             // are SortableForLengths
      printOut(sal);                                   // sal
      sort(sal);                                       // sal
      printOut(sal);                                   // sal

      Sortable[] sac = {new Tide(4d,2d),               // Array  sac  is of type
                        new Tide(2d,4d),               // Sortable but all elements
                        new Tide(1.5d,1.5d)};          // are SortableForComplex
      printOut(sac);                                   // sac
      sort(sac);                                       // sac
      printOut(sac);                                   // sac
    }

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

   private static void sort(Sortable[] s)
    { 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];
            s[i-1] = s[i];
            s[i] = t;
            i--;
          }
       }
    }
 }

interface Sortable
 { public abstract boolean greaterThan(Sortable that);
 }

interface SortableForLengths extends Sortable
 { public abstract double rank();
 }

abstract class Shape implements SortableForLengths
 { public abstract double perimeter();

   public abstract double area();

   public boolean greaterThan(Sortable that)
```

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

class Square extends Shape
 { 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 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;

   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 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());
    }
 }

interface SortableForComplex extends Sortable         // new interface
 { public abstract double modulus();
                                                      // two methods needed
   public abstract double argument();                 // to determine criterion
 }                                                    // for sorting Complex

abstract class Vector implements SortableForComplex   // new abstract class
 { public boolean greaterThan(Sortable that)
    { if (this.modulus() > ((SortableForComplex)that).modulus())
         return true;
      else if (this.modulus() == ((SortableForComplex)that).modulus())
              return this.argument() > ((SortableForComplex)that).argument();
           else return false;
    }
 }

class Tide extends Vector                             // new class
 { private double northing, easting;

   public Tide(double northing, double easting)
    { this.northing = northing;
      this.easting  = easting;
    }

   public double modulus()
    { return Math.sqrt(this.northing*this.northing + this.easting*this.easting);
    }

   public double argument()
    { return Math.atan2(this.northing, this.easting)*180d/Math.PI;
    }

   public String toString()
    { return String.format("  Tide -%n" +
                           "  Northing is %.2f%n" +
                           "  Easting is %.2f%n" +
                           "  Modulus is %.2f%n" +
                           "  Argument is %.2f%n",
               this.northing, this.easting, this.modulus(), this.argument());
    }
 }
```

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

sa[0]:   Tide -
  Northing is 4.00
  Easting is 2.00
  Modulus is 4.47
  Argument is 63.43

sa[1]:   Tide -
  Northing is 2.00
  Easting is 4.00
  Modulus is 4.47
  Argument is 26.57

sa[2]:   Tide -
  Northing is 1.50
  Easting is 1.50
  Modulus is 2.12
  Argument is 45.00

sa[0]:   Tide -
  Northing is 1.50
  Easting is 1.50
  Modulus is 2.12
  Argument is 45.00

sa[1]:   Tide -
  Northing is 2.00
  Easting is 4.00
  Modulus is 4.47
  Argument is 26.57

sa[2]:   Tide -
  Northing is 4.00
```

```
Easting is 2.00
Modulus is 4.47
Argument is 63.43
```

SUGGESTIONS FOR FURTHER EXPERIMENTS

Add another sub-class Wind (which can be much like Tide) and mix the types in the sac array.

Add other child classes of SortableForLengths. For example class Vehicles with sub-classes Bicycle and Car, both of which can rank on a data field length.

Invent similar child classes for SortableForComplex.