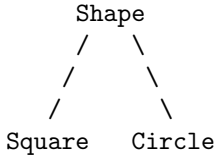
MODULE 5q - An interface

A SECOND (TRULY EVIL!) VARIATION OF ShapeA - INTRODUCTION

The versions of the ShapeA program introduced so far have involved a parent class Shape and two child classes Square and Circle:



In the most recent version of the ShapeA program, a polymorphic array was set up consisting of a mixture of elements, some of type Square and some of type Circle. The elements were sorted into ascending order of area.

To achieve this goal, the class definition of Square and the class definition of Circle each incorporated the following compare() method:

```
public boolean compare(Shape that)
 { return this.area() > that.area();
 }
```

It was noted that the duplication of the compare() method was an example of poor style. An altogether better approach will be described later in the course but, for the purposes of illustration, the next few variations are going to be of poorer style still!

It would, of course, be trivial to sort a mixture of Squares and Circles by perimeter instead of by area. The duplicated compare() method would simply be changed to:

```
public boolean compare(Shape that)
 { return this.perimeter() > that.perimeter();
 }
```

This is a straightforward change but now let's get awkward!

Suppose that when two Shapes are compared, instead of comparing their areas or their perimeters, one compares their sides (if both are Squares) or their radii (if both are Circles) or a side and a radius if there is one of each. Since a side and a radius both have the dimensions of length it is not unreasonable to compare one against the other.

Such a wish considerably complicates the compare() methods and the two will no longer be the same. This explains why the fix suggested earlier for avoiding duplication was not carried out. A first attempt at the compare() method in class Square might be:

```
public boolean compare(Shape that)
 { return this.side > that.side;
 }
```

The problem lies in the argument that whose type is given simply as the parent type Shape. This version of compare() would stand a chance if that happens to be a Square because that.side is meaningful. It is not meaningful if that is of type Circle when the method would more sensibly be:

```
public boolean compare(Shape that)
 { return this.side > that.radius;
 }
```

In fact, neither of these methods can ever work because that has type Shape and the compiler cannot find a data field side or a data field radius in class Shape.

One way to proceed (or dig oneself into a deeper hole!) is to use the operator instanceof as in:

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

The value of that instanceof Square is true if that happens to be of type Square and false otherwise. Even if true, one still cannot write that.side but has first to cast that to type Square by means of (Square)that as shown (and this has to be in brackets). Likewise one needs the cast (Circle)that if that is a Circle.

Note that there is no need to cast this because the method is being defined in class Square so this is guaranteed to be of type Square.

In a similar way the compare() method in class Circle might be:

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

A SECOND VARIATION - PROGRAM

The new version of the ShapeA program is shown below. All changes to the previous version of the ShapeA program are indicated by comments.

The revised compare() methods are incorporated into class Square and class Circle. The compare() method in class Square needs to access the data field radius in class Circle so this data field can no longer be private (likewise data field side needs to be accessible from class Circle). These changes from private to public constitute more bad news (but could be avoided by the use of public getSide() and getRadius() methods).

The principal reason that the programming style is so bad is most

easily appreciated if you consider adding a new child of class Shape, say a Triangle. This would require every compare() method in the existing child classes of Shape to be modified, a sure recipe for disaster!

Using the type of an item as a means of deciding which variant of an operation to perform is one of the classic examples of 'a bad thing in traditional programming which object oriented programming is supposed to allow us to avoid'.

Each object should know how to do the operation on itself and shouldn't have to worry about what the other objects do (or even whether they exist).

Set up this version now.

```
public class ShapeA
 { public static void main(String[] args)
    { Shape[] sa = {new Square("Trafalgar", 2d),
                    new Square("Leicester", 3d),
                    new Circle("Arctic", 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].compare(s[i]))
          { Shape t = s[i-1];
            s[i-1] = s[i];
            s[i] = t;
            i--;
          }
       }
    }
 }

abstract class Shape
 { private String name;

   public Shape(String s)
    { setName(s);
    }

   public String getName()
    { return this.name;
    }
```

```
   public void setName(String s)
    { this.name = s;
    }

   public abstract double perimeter();

   public abstract double area();

   public abstract boolean compare(Shape that);
 }


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

   public Square(String s, double side)
    { super(s);
      this.side = side;
    }

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

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

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

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

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

   public Circle(String s, double radius)
    { super(s);
      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)                 // new EVIL compare()
    { return this.radius > (that instanceof Circle
                            ? ((Circle)that).radius
                            : ((Square)that).side);
    }

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

WHY IS THE CODE SO BAD?

The two most recent versions of the ShapeA program have prompted four complaints:

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.

The latest version has attended to the second complaint and made things worse! As noted earlier, a better approach will be left pending.

Meanwhile the present version will be tested and then two further variations will be described to illustrate an alternative form of inheritance in Java.

TRY IT OUT

Compile and run the program. The output should be:

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

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

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

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

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

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

This output is much as before except that the order of the sorted elements will be different.

MULTIPLE INHERITANCE

Consider again abstract class Shape and the abstract methods in it:

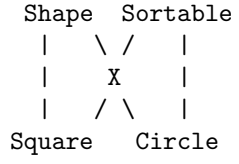
```
abstract class Shape
 { private String name;

   public Shape(String s)
    { setName(s);
    }

   public String getName()
    { return this.name;
    }

   public void setName(String s)
    { this.name = s;
    }

   public abstract double perimeter();

   public abstract double area();

   public abstract boolean compare(Shape that);
 }
```

Informally, the abstract methods are simply the headings of methods which some program designer thinks should be included in child classes which represent geometric figures.

Both perimeter() and area() are obvious choices since they strongly

relate to Shapes. By contrast, compare() could apply much more generally since many other things apart from Shapes can be compared, for example Strings, goal averages and prices.

This leads to the idea that one might have TWO parent classes. In the present case, the abstract methods which strongly relate to Shapes would be in one and the abstract method which applies less specifically might be in the other. Keeping the name Shape for the first parent class and introducing the name Sortable for the other, the inheritance diagram might be depicted thus:



In outline the two parent classes might be written as:

```
abstract class Shape
 { .
   .

   public abstract double perimeter();

   public abstract double area();
 }

abstract class Sortable
 { .
   .

   public abstract boolean compare(Sortable s);
 }
```

Only the abstract methods in each class are shown and class Shape is as it was originally; it specifies methods which are fairly narrowly applicable to geometric figures. The idea of class Sortable is that given any two descendants it should be possible to compare them. In other words they are ‘Sortable’. Notice that the argument of compare() is now of type Sortable.

The separation achieved by having these two parent classes naturally leads to a concept known as ‘multiple inheritance’. In particular it seems that class Square and class Circle would inherit from both Shape and Sortable. One might hope Java allowed syntax such as:

```
class Square extends Shape, Sortable
 { ...
```

Unfortunately, this is not permitted. The designers of Java decided that general-purpose multiple inheritance is full of hazards and the disadvantages outweigh the advantages. Nevertheless, Java permits a limited form of multiple inheritance via interfaces...

AN interface - A RESTRICTED FORM OF abstract CLASS

An interface takes the idea of an abstract class one step further.
It is almost true to say that

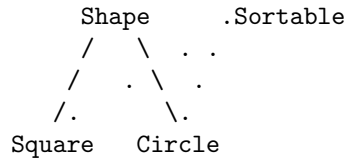
an interface is a restricted form of abstract class

in that it can contain abstract methods only. [In fact it may also contain data fields but these are implicitly both static and final; an interface certainly cannot contain any non-abstract methods.]

Given the restrictions, inheriting from an interface is nothing like as big a deal as inheriting from a class (whether abstract or not) and Java allows a child class to inherit from any number of interfaces. Such a child class may additionally inherit from a single class which may be abstract or non-abstract.

The abstract class Shape includes data fields and non-abstract methods so cannot be an interface. The abstract class Sortable doesn't need to contain anything other than the method heading for compare() so it can readily be converted into an interface.

An appropriate inheritance diagram to illustrate the proposed relationships is:



Dashes represent proper inheritance (as by a sub-class from a super class) and dots represent the lesser form of inheritance (by a sub-class from an interface).

Two principal changes will be incorporated in the next version of the program shown overleaf. First, the abstract method compare() will be removed from class Shape (thereby restoring Shape to its original form).

Secondly, the proposed abstract class Sortable (which is the new home of the compare() method) will be converted into an interface; this is achieved simply by replacing abstract class by interface in the heading line:

```
interface Sortable
 { public abstract boolean compare(Sortable s);
 }
```

The child classes Square and Circle will need to inherit from the abstract class Shape AND from the interface Sortable. This is achieved by using the qualifier implements in the heading line as in this example for class Square:

```
class Square extends Shape implements Sortable
 { ...
```

Note that a child class ‘extends’ a parent class but ‘implements’
an interface.

A THIRD VARIATION

The complete revised version of the ShapeA program is shown below.
All changes to the previous version are indicated by comments.

Note that individual Shapes like Squares and Circles inherit from both
Shape and Sortable and either Shape or Sortable can be used as a
generic type to describe a mixture of Squares and Circles.

When setting up the polymorphic array, Sortable is decidedly the more
appropriate type to describe the elements since it is the Sortable
interface that specifies the compare() method. This choice is also
consistent with the earlier decision to have the argument of the
compare() method of type Sortable.

In the program the type of the polymorphic array sa is Sortable
and the type of the formal argument of both printOut() and Sort()
is also Sortable and not Shape. Set up this version now.

```
public class ShapeA
 { public static void main(String[] args)             // type Sortable now
    { Sortable[] sa = {new Square("Trafalgar", 2d),
                       new Square("Leicester", 3d),
                       new Circle("Arctic", 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].compare(s[i]))
          { Sortable t = s[i-1];                      // type Sortable
            s[i-1] = s[i];
            s[i] = t;
            i--;
          }
       }
    }
 }
```

```
abstract class Shape
 { private String name;

   public Shape(String s)
    { setName(s);
    }

   public String getName()
    { return this.name;
    }

   public void setName(String s)
    { this.name = s;
    }

   public abstract double perimeter();

   public abstract double area();
                                                    // compare() removed
 }

interface Sortable                                  // new interface
 { public abstract boolean compare(Sortable that);  // type Sortable
 }

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

   public Square(String s, double side)
    { super(s);
      this.side = side;
    }

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

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

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

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

```
class Circle extends Shape implements Sortable        // implements Sortable
 { public double radius;                               // now public

   public Circle(String s, double radius)
    { super(s);
      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(Sortable that)              // type Sortable
    { return this.radius > (that instanceof Circle
                            ? ((Circle)that).radius
                            : ((Square)that).side);
    }

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

The program in general, and the compare() methods in particular, still reflect appalling practice. Improvements will be provided later!

TRY IT OUT

Compile and run this program. It ought to give the same output as the previous version.

A FOURTH VARIATION

In the program just tested, the child classes Square and Circle each inherit from a single parent class, Square, and a single interface, Sortable. As already noted, Java allows a child class to inherit from any number of interfaces but from at most one class.

The fourth variation of the ShapeA program is shown overleaf. All changes to the previous version are indicated by comments.

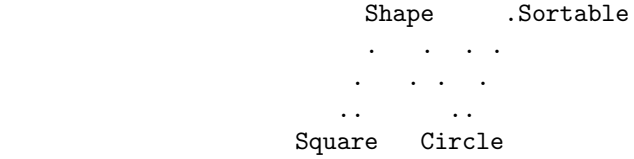
In this version, the data field name, the constructor, and the non-abstract methods have been removed from class Shape leaving it with just the two abstract methods perimeter() and area(). In this cut-down state, class Shape is ripe for conversion to an interface and

that is how it appears.

Given the removal of the data field name and the constructor from Shape, there can no longer be names (like Trafalgar) associated with individual Shapes. Additionally, the constructors in the child classes Square and Circle can no longer include the super(s) statements.

In consequence, there is only one argument in each of the constructors of Square and Circle and when the sa array is set up the elements are given as new Square(2d) rather than new Square("Trafalgar",2d) as before.

The child classes Square and Circle now inherit from TWO interfaces and an appropriate diagram is:



To indicate inheritance from two interfaces the heading line of class Square is modified from:

```
class Square extends Shape implements Sortable
```

to

```
class Square implements Shape, Sortable
```

Note the use of the comma. In a more ambitious case a child class may specify three, four or more interfaces after the implements qualifier.

Set up this version now.

```
public class ShapeA
 { public static void main(String[] args)             // type Sortable now
    { Sortable[] sa = {new Square(2d),                 // no names now
                       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].compare(s[i]))
         { Sortable t = s[i-1];
           s[i-1] = s[i];
           s[i] = t;
           i--;
         }
       }
    }
 }

interface Shape                                          // now an interface
 { public abstract double perimeter();                   // cut down to just
                                                         // abstract methods
   public abstract double area();
 }

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

class Square implements Shape, Sortable                  // implements Shape, Sortable
 { public double side;

   public Square(double side)                            // cut-down constructor
    { this.side = side;
    }

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

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

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

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

class Circle implements Shape, Sortable                  // implements Shape, Sortable
 { public double radius;

   public Circle(double radius)                          // cut-down constructor
    { 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(Sortable that)              // type Sortable
    { return this.radius > (that instanceof Circle
                            ? ((Circle)that).radius
                            : ((Square)that).side);
    }

   public String toString()
    { return String.format("  Circle -%n" +           //  cut-down first line
                           "  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 is shown overleaf.

Here is the output. The only difference from before is that there are no longer any names associated with the Shapes.

```
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
```