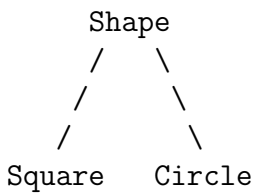
MODULE 4q - An abstract class

SQUARES AND CIRCLES

The Block.java program incorporated a ‘parent’ class Square and a ‘child’ class Cube. The relationship was established by the declaration of class Cube which included the clause extends Square

In this worksheet, class Square will itself be regarded as a child class whose parent class is Shape. Potentially, the idea is to develop a series of sister classes which might begin with Square and Circle and go on to include Triangle, Hexagon and so on.

An inheritance diagram describing the proposed relationships is:



In a fairly natural way this reflects the fact that squares and circles are both examples of shapes. As such, there are certain questions one can reasonably ask of both a circle and a square (such as, ‘what is the area?’ or ‘what is the perimeter?’) but some questions can be asked only of one or the other but not both (such as, ‘what is the length of a side?’ or ‘what is the radius?’).

An obvious way to exploit inheritance is to arrange for class Shape to include data fields and methods which are common to all Shapes and to let the child classes augment these data fields and methods by others which are specific to individual shapes.

A first suggestion as to what data fields and methods should be in the parent class Shape and what should be in the child classes Square and Circle might be as follows:

```
class Shape  -  data field  name
             -  methods     getName(), setName(), perimeter(), area()
```

```
class Square -  data field  side
             -  methods     getSide(), setSide()

class Circle -  data field  radius
             -  methods     getRadius(), setRadius()
```

The idea of the name data field is to give the user the opportunity of labelling individual Shapes, so a Square might be given the name Trafalgar and a Circle might be given the name Arctic. The name would normally be set once and for all via a constructor and could be read by means of the getName() method. The setName() method would provide a means of changing a name.

Incorporating a constructor, an outline draft of the parent class Shape might be:

```
class Shape
 { private String name;           // note that  name  is declared private

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

   public String getName()        // note that getName() is a String method
    { return this.name;           // which has no arguments
    }

   public void setName(String s)  // note that setName() is a void method
    { this.name = s;              // which has a String argument
    }

   public double perimeter()
    { ...                         // see text overleaf about this body
    }

   public double area()
    { ...                         // see text overleaf about this body
    }
 }
```

Using the principle of encapsulation, the data field name is declared private but the constructor and other methods which have to be accessed from outside the class are declared public.

Since the constructor Shape() is intended to do exactly what setName() does, the body of Shape() simply calls setName() as shown. Note the comments against the methods getName() and setName().

AN abstract CLASS

There is clearly a difficulty when it comes to providing bodies for perimeter() and area() since these methods have to be inherited by both Square and Circle.

One could, perhaps, deem a Square to be the 'default Shape' and provide perimeter() and area() methods specifically for a Square. These methods would have to be overridden in class Circle or any other child of Shape. This is not a sound approach. It would require data field side to be declared in class Shape rather than in class Square and would spoil the neutrality of Shape.

The problem that has arisen is a consequence of a Shape being something of an 'abstraction'. One can readily visualise a Square or a Circle but an unspecified Shape is rather more intangible. We may be sure that every Shape that we ever want to consider will have a perimeter and an area but we cannot say how these are to be calculated until a particular Shape is specified.

In Java, this train of thought gives rise to the 'abstract method'. In practical terms, an abstract method is one which has a heading line and no body. In a parent class, such a method lays down a marker specifying that a child class (or possibly a later generation class) MUST provide the method in full, complete with body, but in the parent itself the body is absent.

An abstract method is declared simply by incorporating the modifier abstract in the heading line and terminating the heading with a semi-colon. The abstract qualifier is omitted in the full declaration in the child class.

There is one further consideration. Any class which contains one or more abstract methods must itself be declared abstract. In the present case the appropriate abstract class Shape should be written:

```
abstract class Shape                    // note the abstract qualifier
 { 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();  // note abstract and the semi-colon

   public abstract double area();       // note abstract and the semi-colon
}
```

A Java rule prohibits the instantiation of an abstract class. This is hardly surprising. A child class that provides bodies for all the abstract methods can, of course, be instantiated.

Although one might imagine that an abstract class is defined as a class that contains at least one abstract method, one can in fact declare a class abstract without it containing any abstract methods. This is the standard way to prevent a class being instantiated.

A FIRST FULL PROGRAM

The program on the facing page incorporates the abstract class Shape just given and includes two appropriate child classes Square and Shape.

The method main() declares a Square jack and a Circle jill and prints them out. In the interests of saving keystrokes, the four methods getSide(), setSide(), getRadius(), and setRadius() have been omitted from the child classes Square and Circle.

Each child class contains a constructor and a toString() method. The constructors in both child classes have two arguments, one for the name and one for the side or radius as appropriate. Note that type double is used for both side and radius (rather than type int which

was used in the Block.java program).

The constructors both use super(s) to invoke the constructor of the parent class to set the name data field and each has a second statement which sets side or radius as appropriate.

By modifying the Block.java program, or otherwise, set up a new file ShapeA.java containing the code on the facing page.

```
public class ShapeA
 { public static void main(String[] args)
    { Square jack = new Square("Trafalgar", 2d);
      Circle jill = new Circle("Arctic", 1.5d);
      System.out.println("Shape jack\n" + jack);
      System.out.println("Shape jill\n" + jill);
    }
 }

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();  // note abstract and the semi-colon

   public abstract double area();       // note abstract and the semi-colon
 }

class Square extends Shape
 { private double side;

   public Square(String s, double side)
```

```
    { super(s);
      this.side = side;
    }

   public double perimeter()            // no abstract
    { return 4d*this.side;
    }

   public double area()                 // no abstract
    { return this.side*this.side;
    }

   public String toString()
    { return "  Square - " + this.getName() + "\n" +
             "  Side is " + this.side + "\n" +
             "  Perimeter is " + this.perimeter() + "\n" +
             "  Area is " + this.area() + "\n";
    }
 }

class Circle extends Shape
 { private double radius;

   public Circle(String s, double radius)
    { super(s);
      this.radius = radius;
    }

   public double perimeter()            // no abstract
    { return 2d*Math.PI*this.radius;
    }

   public double area()                 // no abstract
    { return Math.PI*this.radius*this.radius;
    }

   public String toString()
    { return "  Circle - " + this.getName() + "\n" +
             "  Radius is " + this.radius + "\n" +
             "  Circumference is " + this.perimeter() + "\n" +
             "  Area is " + this.area() + "\n";
    }
 }
```

TRY IT OUT

Compile and run the program. The output should be:

```
Shape jack
  Square - Trafalgar
  Side is 2.0
  Perimeter is 8.0
  Area is 4.0

Shape jill
  Circle - Arctic
  Radius is 1.5
  Circumference is 9.42477796076938
  Area is 7.0685834705770345
```

A FIRST (SOMEWHAT SINFUL!) VARIATION

One of the advantages of having the parent class Shape is that one can have a Shape array. In this, some elements may be of class Square and others may be of class Circle. When an array has elements of more than one type it is said to be polymorphic (literally 'many shaped').

It is perfectly possible to sort the elements of a polymorphic array into order but one needs to decide (in the present case) what makes a particular Square larger than a particular Circle. Let's suppose that the Shape with the greater area is deemed to be the larger.

One way of proceeding would be to add a third abstract method compare() to the abstract class Shape. This is a way of insisting that each child class incorporates a boolean compare() method. This would return true if a particular instantiation of the class had a larger area than some other instantiation.

Modify the first part of the ShapeA program so that it appears thus:

```
public class ShapeA
 { public static void main(String[] args)
    { Shape[] sa = {new Square("Trafalgar", 2d),      // a polymorphic array
                    new Square("Leicester", 3d),
                    new Circle("Arctic", 1.5d)};
      printOut(sa);
      sort(sa);                                       // sort this array
      printOut(sa);
    }

   private static void printOut(Shape[] s)           // new method
    { for (int i=0; i<s.length; i++)
         System.out.println("sa[" + i + "]: " + s[i]);
    }

   private static void sort(Shape[] s)               // new method
    { 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);       // new abstract method
 }
```

Early in the method main() a polymorphic array of three elements is set up. This contains two elements of type Square and one of type Circle.

The sort() method used to sort the elements into ascending order of area incorporates a pairwise comparison in the condition of the while-statement. First time in, when i=1, this amounts to:

```
          s[0].compare(s[1])
```

This invokes the compare method in s[0] (a reference to a Square) and passes to it the actual argument s[1]. In this case s[1] refers to another Square but it could equally refer to a Circle because the corresponding formal argument that being of the parent type Shape can equally accept a Square or a Circle.

A suitable method compare() complete with body would be:

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

This uses the heading insisted upon by the third abstract method in (the augmented) class Shape. This could be used in both class Square and class Circle.

The reason the program is not very respectable is given later. For the moment, simply complete the program by incorporating the compare() method into the two child classes as shown.

```
class Square extends Shape
 { private 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(Shape that)       // BAD duplication (see below) //
    { return this.area() > that.area();                                     //
    }                                                                       //
                                                                            //
   public String toString()                                                 //
    { return "  Square - " + this.getName() + "\n" +                        //
             "  Side is " + this.side + "\n" +                              //
             "  Perimeter is " + this.perimeter() + "\n" +                  //
             "  Area is " + this.area() + "\n";                             //
    }                                                                       //
 }                                                                          //
                                                                            //
class Circle extends Shape                                                  //
 { private double radius;                                                   //
                                                                            //
   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)       // BAD duplication (see above) //
    { return this.area() > that.area();
    }

   public String toString()
    { return "  Circle - " + this.getName() + "\n" +
             "  Radius is " + this.radius + "\n" +
             "  Circumference is " + this.perimeter() + "\n" +
             "  Area is " + this.area() + "\n";
    }
 }
```

WHY IS IT SINFUL?

There is no need for you to include the comments but note that they draw attention to the fact that the identical method compare() is being used in class Square and in class Circle. Such duplication of code should always ring warning bells and often means unsound practice.

A simple fix would be to incorporate the duplicated method in class Shape, of course as a non-abstract method, and it would then be inherited by both Square and Circle avoiding any need to write the method twice.

This fix will not be applied yet for reasons which will become clear later.

Another, lesser, quibble concerns the choice of the name ‘compare’ for a method. It is not clear whether this should return true if some particular instantiation is larger than some other or smaller than it. It would be better to choose a name like greaterThan to avoid confusion.

There will be more to say about poor style later.

TRY IT OUT

Compile and run the program. The output should be:

```
sa[0]:   Square - Trafalgar
  Side is 2.0
  Perimeter is 8.0
  Area is 4.0

sa[1]:   Square - Leicester
  Side is 3.0
  Perimeter is 12.0
  Area is 9.0

sa[2]:   Circle - Arctic
  Radius is 1.5
  Circumference is 9.42477796076938
  Area is 7.0685834705770345

sa[0]:   Square - Trafalgar
  Side is 2.0
  Perimeter is 8.0
  Area is 4.0

sa[1]:   Circle - Arctic
  Radius is 1.5
  Circumference is 9.42477796076938
  Area is 7.0685834705770345

sa[2]:   Square - Leicester
  Side is 3.0
  Perimeter is 12.0
  Area is 9.0
```