MODULE 3q - An Extended Java Object

THE BOX PROGRAM RENAMED

Copy the file Box.java to Block.java and then make all the amendments indicated by comments in the program below. The name of the public class is changed from Box to Block in the first line and, secondly, the last System.out.printf statement has been removed. Finally, there is no static in class Square.

Set up this program now.

```
public class Block                                    // new name
 { public static void main(String[] args)
    { Square jack = new Square(6);
      System.out.printf("Details of jack...%n%s%n", jack.toString());
      Square jill = new Square(5);
      System.out.printf("Details of jill...%n%s%n", jill);
    }                                                 // printf() removed
 }

class Square
 { private int side;                                  // static removed

   public Square(int s)
    { this.side = s;
    }

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

   public String toString()
    { return String.format("Square: Side = %d%n" +
                           "        Area = %d%n", this.side, this.area());
    }
  }
```

Try the program out. It should behave exactly as an earlier version of Box.java did. The output should be:

```
Details of jack...
Square: Side = 6
        Area = 36

Details of jill...
Square: Side = 5
        Area = 25
```

SQUARES AND CUBES

The goal of this worksheet is to experiment with a second do-it-yourself type Cube which, of course, is a three-dimensional version of a Square.

It would not be difficult to declare class Cube thus:

```
class Cube
 { private int side;

   public Cube(int s)
    { this.side = s;
    }

   public int surface()
    { return 6*this.side*this.side;
    }

   public String toString()
    { return String.format("Cube:   Side    = %d%n" +
                           "        Surface = %d%n", this.side, this.surface());
    }
 }
```

Apart from the changes of name from Square to Cube and area to surface this is just about identical to class Square except that the new surface() method has a factor of 6 in it to reflect the fact that a Cube is made from six Squares and its total surface area is therefore six times that of one of the component Squares.

This relationship between a Cube and a Square leads to the idea of extending a class...

A FIRST VARIATION

The first variation of the Block program, shown below, incorporates a declaration of a new class Cube. Note that jack continues to be of type Square but jill is of type Cube as indicated. All changes to the previous version are indicated by comments. Set up this version now.

```
public class Block
 { public static void main(String[] args)
    { Square jack = new Square(6);
      System.out.printf("Details of jack...%n%s%n", jack.toString());
      Cube jill = new Cube(5);                          // type  Cube  now
      System.out.printf("Details of jill...%n%s%n", jill);
    }
 }

class Square
 { private int side;

   public Square(int s)
```

```
    { this.side = s;
    }

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

   public String toString()
    { return String.format("Square: Side = %d%n" +
                           "        Area = %d%n", this.side, this.area());
    }
 }

class Cube extends Square                             // note   extends Square
 { public Cube(int s)
    { super(s);                                       // super(s)  in constructor
    }

   public int surface()                               // multiplies inherited
    { return 6*this.area();                           // area() by a factor of 6
    }

   public String toString()                           // overrides inherited toString()
    { return String.format("Cube:   Surface = %d%n", this.surface());
    }
 }
```

Try the program out. The output should be:

```
Details of jack...
Square: Side = 6
        Area = 36

Details of jill...
Cube:   Surface = 150
```

INHERITANCE, OVERRIDING AND super

By writing `class Cube extends Square` in the heading line, class Cube is said to ‘inherit’ from class Square. In effect, class Cube contains all the data fields and methods of class Square as well as any data fields and methods declared in itself.

There are two exceptions: the first is that class Cube doesn’t inherit the constructor of class Square and the second is that any method in class Cube which has the same name as one in class Square will take precedence. Thus the `toString()` method in class Cube is said to ‘override’ the `toString()` method inherited from class Square.

A principal consequence is that class Cube inherits data field `side` and this can be used to specify the side of the Cube just as well as it can be used to specify the side of a Square provided a slightly different constructor is used. The obvious constructor for `Cube` is:

```
   public Cube(int s)
    { this.side = s;
    }
```

Unfortunately this doesn't work. Instead of assigning a value to this.side in the body of the constructor Cube the approved approach is to invoke the constructor of the class being inherited from. This suggests that Square(s) might be appropriate but the rules require using super(s) as a general-purpose way of invoking the constructor of the ‘parent’ class.

AN INHERITANCE DIAGRAM

The terms ‘inheritance’, ‘parent class’ and ‘child class’ are often used when discussing object oriented programming. Sometimes an inheritance diagram is drawn to describe the relationships:

```
            Square
             /  \
            /    \
           /      \
        Cube      ????
```

Rather as in a family tree, this reflects the fact that class Cube is a child class whose parent class is Square. The ???? represents a potential sister class for Cube. For example one might have a class Domino inheriting from Square on the grounds that a Domino is formed from two Squares.

It is important to note that a parent class may have any number of children but a child class has exactly one parent.

A SECOND VARIATION

The method surface() returns the surface area of a Cube and achieves this by simply invoking the area() method inherited from Square. The area() method supplies the area of one Square face of the Cube and, when this is multiplied by 6, you get the total surface area of the Cube.

It might be thought that the surface() method could equally return 6*this.side*this.side and this is the only modification in the second variation of the Block program shown below. Set up this version now.

```
public class Block
 { public static void main(String[] args)
    { Square jack = new Square(6);
      System.out.printf("Details of jack...%n%s%n", jack.toString());
      Cube jill = new Cube(5);
      System.out.printf("Details of jill...%n%s%n", jill);
    }
 }
```

```
class Square
 { private int side;

   public Square(int s)
    { this.side = s;
    }

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

   public String toString()
    { return String.format("Square: Side = %d%n" +
                           "        Area = %d%n", this.side, this.area());
    }
 }

class Cube extends Square
 { public Cube(int s)
    { super(s);
    }

   public int surface()
    { return 6*this.side*this.side;                    // THE ONLY CHANGED LINE
    }

   public String toString()
    { return String.format("Cube:   Surface = %d%n", this.surface());
    }
 }
```

TRY IT OUT

Try compiling this program. You will get an error message complaining:

```
   side has private access in Square
```

Although side really is 'in class Cube' it is there by inheritance, and Java still recognises that its origin is in a different class AND that it has the visibility modifier private in that different class.

A THIRD VARIATION

Change the declaration of the data field side from

```
   private int side;
```

to

```
   public int side;
```

and try compiling again. There should be no error messages and, when the program is run, the output should be as before.

BAD PRACTICE

Although the program now works again, it is generally bad practice to make a data field public in these circumstances. The principle of encapsulation is compromised. The fourth variation will show the approved way of attending to the problem.

Before looking at the next variation, note that the consequences of side being private explain why the toString() method of Cube didn't begin:

```
      return String.format("Cube:   Side    = %d%n" +
```

When private, the inherited data field side would not be accessible.

A FOURTH VARIATION

The approved way of determining the value of a private data field is to get at it via a public method. In the fourth variation shown below, the data field side is private once more but there is a new and public method getSide() in class Cube which returns the value of side. This method is invoked in both the surface() method and the toString() method of Cube.

All changes to the previous version are indicated by comments. Set up this version now.

```
public class Block
 { public static void main(String[] args)
    { Square jack = new Square(6);
      System.out.printf("Details of jack...%n%s%n", jack.toString());
      Cube jill = new Cube(5);
      System.out.printf("Details of jill...%n%s%n", jill);
    }
 }

class Square
 { private int side;                                  // back to private

   public Square(int s)
    { this.side = s;
    }

   public int getSide()
    { return this.side;                               // new method
    }

   public int area()
    { return this.side*this.side;
    }
   public String toString()
    { return String.format("Square: Side = %d%n" +
                           "        Area = %d%n", this.side, this.area());
    }
 }

class Cube extends Square
 { public Cube(int s)
    { super(s);
    }

   public int surface()
    { return 6*this.getSide()*this.getSide();         // changed again
    }

   public String toString()                           // modified toString()
    { return String.format("Cube:   Side    = %d%n" +
                           "        Surface = %d%n", this.getSide(), this.surface());
    }
 }
```

TRY IT OUT

Compile and run this program. The output should be:

```
Details of jack...
Square: Side = 6
        Area = 36

Details of jill...
Cube:   Side =    5
        Surface = 150
```

Note that despite side being declared private there is no difficulty about referring to this.side in class Square because the data field side is declared in this class and is being referred to within it.

Note also that the approved way of changing the value of side from outside class Square would also be to go via a public method, perhaps called setSide() as in:

```
   public void setSide(int s)
    { this.side = s:
    }
```

This, of course, is effectively duplicating the work of the constructor but one cannot use a constructor except at the time of instantiation.

JAVA NAMING CONVENTIONS - YET MORE

Note that getSide and setSide follow the Java naming convention.

As method names they begin with lower-case letters but the new word Side in the middle merits an upper-case S. The data field side continues of course to merit a lower-case s.

A FIFTH VARIATION - OVERLOADING

The constructors in class Square and class Cube enable the user to specify any int value for the side of a Square or a Cube. Suppose it turns out that the most commonly used value for side is 1 (giving rise to a so-called ‘unit square’ or ‘unit cube’). It would be useful to be able to set up such Squares and Cubes by writing

```
   new Square()          and          new Cube()
```

where there are no actual arguments. For these operations to work the constructors would have to be

```
   public Square()        and         public Cube()
    { this.side = 1;                   { super(1);
    }                                  }
```

where there are no formal arguments. Notice that super(1) calls the earlier version of the constructor in class Square and it would probably be better now to use plain super() to invoke the new argumentless constructor in class Square.

A potential worry is whether the new versions of the constructors can coexist with the earlier, more general purpose, constructors. It turns out that Java allows such coexistence, which is generally known as ‘overloading’.

Moreover, Java allows overloading of methods as well as constructors. Thus, any class may contain several constructors or several methods of the same name provided only that their argument lists are different. The difference has to be more than a simple change of identifier, thus the following:

```
   public Square(int s)      and      public Square(int edge)
```

are not deemed to be different, In each case there is a single int argument and the fact that one is called s and the other is called edge doesn’t make them distinguishable.

The fifth variation of the Block program is shown overleaf. All changes to the previous version are indicated by comments. This variation incorporates the earlier constructors as well as new versions which, when called, result in the instantiation of unit Squares and unit Cubes respectively. These might be regarded as defaults.

In method main(), jack and jill are set to a unit Square and a unit Cube respectively. The earlier constructors, though present, are not used directly. Note that super() has been used in the new constructor in class Cube though super(1) would achieve the same effect.

Set up this version now.

```
public class Block
 { public static void main(String[] args)
    { Square jack = new Square();                     // no argument
      System.out.printf("Details of jack...%n%s%n", jack.toString());
      Cube jill = new Cube();                         // no argument
      System.out.printf("Details of jill...%n%s%n", jill);
    }
 }

class Square
 { private int side;

   public Square()                                    // new constructor
    { this.side = 1;
    }

   public Square(int s)                               // old constructor
    { this.side = s;
    }

   public int getSide()
    { return this.side;
    }

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

   public String toString()
    { return String.format("Square: Side = %d%n" +
                           "        Area = %d%n", this.side, this.area());
    }
 }

class Cube extends Square
 { public Cube()                                      // new constructor
    { super();                                        // no argument in super()
    }

   public Cube(int s)                                 // old constructor
    { super(s);
    }

   public int surface()
    { return 6*this.getSide()*this.getSide();
    }

   public String toString()
    { return String.format("Cube:   Side    = %d%n" +
                           "        Surface = %d%n", this.getSide(), this.surface());
    }
 }
```

TRY IT OUT

Compile and run this program. The output should be:

```
Details of jack...
Square: Side = 1
        Area = 1

Details of jill...
Cube:   Side =    1
        Surface = 6
```

EXERCISES

Verify that the earlier constructors still function properly by setting up four local variables in method main() thus:

```
    { Square jack = new Square();                    // unit Square
      Cube jill = new Cube();                        // unit Cube
      Square jacky = new Square(6);                  // Square with side 6
      Cube jilly = new Cube(5);                      // Cube with side 5
```

Here jack and jill exploit the new constructors and jacky and jilly exploit the earlier versions. Add appropriate printf() statements to write out the four objects.

Next, add an extra method volume() to class Cube. The volume can be calculated by multiplying the area of one face by the side but the getSide() method must be used. The new method will be:

```
   public int volume()
    { return this.getSide()*this.area();
    }
```

Additionally modify the toString() method so that the volume is written out too. Try the program out:

```
   public String toString()
    { return String.format("Cube:   Side    = %d%n" +
                           "        Surface = %d%n" +
                           "        Volume  = %d%n",
                           this.getSide(), this.surface(), this.volume());
    }
```

Add a new method perimeter() to class Square and arrange that this method returns the perimeter of the Square. Adjust the toString() method in class Square so that when jack's details are written out they include the Perimeter as well as the Side and Area.

Next add a new method seam() to class Cube and arrange that this method returns the total length of all the sides of the Cube. Adjust the toString() method in class Cube so that when jill's details are written out they include the Seam as well as the Side, Area and Volume.

OTHER TASKS

By this stage of the course you should be able to attempt the following problems in the Problems sheet:

9. Determining a Square Root by Iteration

10. The Recurring Fraction Problem

3. [REVISITED] All Prime Numbers less than 600

   Solve problem 3 using a boolean array instead of an int array.
   It makes much more sense to use type boolean.