MODULE 3p - A Java Object

A PROGRAM WITH TWO CLASSES

Key the following source code into the file Box.java:

```
public class Box
 { public static void main(String[] args)
    { Square jack;
      jack = new Square();
      jack.side = 6;
      System.out.printf("Area of jack is %d%n", jack.area());
    }
 }

class Square
 { public int side;

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

This program contains TWO classes: Box contains the all important method main() but no data fields and no other methods. The other class Square contains a single data field side and a single method area() (and notice that both are public but NEITHER is static).

In the method main(), a single local variable jack is declared but it is not of type int or of type String but of type Square and this introduces a whole new way of thinking...

DO-IT-YOURSELF TYPES

Square is a do-it-yourself type and in Java the names of such types are always the names of classes. If you want to set up such a type you must declare a class whose name is that of the desired type and within the declaration you must specify your requirements. In the present case we want to specify a Square...

A Square is completely specified just by saying what the length of a side is and in class Square the data field side will hold the value representing the length of the side in, say, inches. For reasons to be explained later, side must be public but NOT static.

Additionally in class Square there is a method area() which can be used to work out the area of the Square from the length of a side. This too must be public but not static.

A NEW SQUARE - A HANDLE

The declaration Square jack sets up a variable jack but jack does NOT have a value until it is assigned a new Square() and this assignment is sometimes said to 'give jack a handle on a Square'.

Having got a handle on a Square, jack can then refer to the side of its Square by using the construction jack.side and jack can refer to the method area() by using the construction jack.area()

In the method main(), the second assignment sets jack.side to 6 (which is to say the data field side is assigned the value 6).

The System.out.printf statement then writes out the area of the square by invoking the method area() via jack.area()

Notice that the brackets are needed because area() is a method but it has no arguments so there is nothing inside the brackets. Of course, area() needs to know the value of the data field side but, WITHIN the method, one can't use jack.side because jack is in a different class and would be unknown in class Square. Accordingly one writes this.side instead.

JAVA NAMING CONVENTIONS - A REMINDER

It is a standard convention in Java that class names begin with an upper-case letter. There are four class names in the program: Box, Square, String and System.

It is also a convention that data field names (e.g. side), method names (e.g. main() and area()) and local variable names (e.g. jack) begin with lower-case letters.

TRY IT OUT

Compile and run the program. The output should be:

```
Area of jack is 36
```

Give an ls command and notice that there are TWO .class files.

A FIRST VARIATION

In the initial version of the program the user took three steps to set up a new Square for jack...

1. Declare the variable as `Square jack;`
2. Assign a new Square as `jack = new Square();`
3. Specify the size as `jack.side = 6;`

In the first variation of the program below, the first two steps are combined into a single statement by arranging for the declaration and

the assignment to be merged.

Modify the original version of the Box program so that it is as shown below. The declaration of jack is merged with the assignment of a new Square to this variable. Also declare a second variable jill in the same way but give Square jill a different size. Set up this version now.

```
public class Box
 { public static void main(String[] args)
    { Square jack = new Square();
      jack.side = 6;
      System.out.printf("Area of jack is %d%n", jack.area());
      Square jill = new Square();
      jill.side = 5;
      System.out.printf("Area of jill is %d%n", jill.area());
    }
 }

class Square
 { public int side;

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

The method main() now declares two local variables, both of type Square.

CLASSES AND OBJECTS - INSTANTIATION

In some ways a class is a kind of once-off DESIGN or outline. This is true of class Square whose design appears in the second half of the program.

The keyword new brings the design to life and results in what is called an 'instantiation' of the class, usually referred to as an object.

The current version of the program has one class Square (and there can never be more than one) but two Square objects: jack and jill.

There can be any (reasonable) number of objects instantiated from a single class and they must be thought of as each quite separate from the other. This is important because jack.side must be a different variable from jill.side or it would be impossible to have different sizes of Square.

TRY IT OUT

Compile and run this program. It ought to give the result:

```
Area of jack is 36
Area of jill is 25
```

A SECOND VARIATION

The initial version of the program took three steps to set up a new Square for jack...

```
   1. Declare the variable as    Square jack;
   2. Assign a new Square as     jack = new Square();
   3. Specify the size as        jack.side = 6;
```

In the first variation, steps 1 and 2 were merged and in the second variation below the third step is merged in too by arranging for the size of the Square to be specified inside the brackets as new Square(6) and for this to work there has to be a special extra method called a 'constructor' incorporated into the class definition.

Modify the first variation to the following form and try it out:

```
public class Box
 { public static void main(String[] args)
    { Square jack = new Square(6);
      System.out.printf("Area of jack is %d%n", jack.area());
      Square jill = new Square(5);
      System.out.printf("Area of jill is %d%n", jill.area());
    }
 }

class Square
 { private int side;          // note  private  instead of  public

   public Square(int s)       // this new special method is
    { this.side = s;          // called a constructor.  It has
    }                         // the same name as the class itself.

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

CONSTRUCTORS

Most classes are intended to be instantiated as objects by some other class. Thus, here, class Square is instantiated as two objects jack and jill by declarations in the method main() in class Box.

Most classes incorporate data fields which need to be initialised and it is convenient to specify the initial values at the time of instantiation.

This goal is achieved by including a constructor in the class and arranging for the arguments of the constructor to be handed the initial values and for the body of the constructor to assign these values to the appropriate data fields.

A constructor is written just like a method except that there is no return type. You should never write public int Square(int s) or public void Square(int s) but just plain public Square(int s) as in the program.

VISIBILITY MODIFIERS - public AND private - ENCAPSULATION

The two keywords public and private are called visibility modifiers and control the accessibility of data fields and methods...

Data fields (and methods) with a private modifier cannot be accessed from outside the class in which they are declared. Roughly speaking the goal is to have all data fields (and any methods not accessed from outside the class) private. This is known as 'encapsulation'.

The idea is that a user should be able to set or read a typical data field only under the control of public methods declared elsewhere in the class. In the present case the constructor is used for setting the value of side (it therefore has to be declared public since it has to be accessed from outside the class). Likewise method area() has to be declared public since it too is accessed from outside the class.

MOST CLASSES CONTAIN CONSTRUCTORS

The reason for the absence of constructors in most examples so far is that most programs have consisted of a single class, the one which incorporates the method main(). Data fields and methods in this class are not accessed from outside the class (except that the method main() itself is the entry point of the Java Virtual Machine).

Accordingly, the class which incorporates the method main() will usually have all its data fields and methods declared private except the method main() itself which must be declared public.

A THIRD VARIATION

In the third variation below, an extra method called toString() is added to class Square and this method provides details of the Square as a String which can be used in a System.out.printf statement.

Modify the Box program so that the two System.out.printf statements are as in the following version and add the toString() method to the declaration of class Square:

```
public class Box
 { 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);
    }
 }
```

```
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());
    }
 }
```

THE toString() METHOD

The method name `toString()` is special in Java and it is policy that most class declarations should incorporate a method with this name with the express purpose of giving details in the form of a String.

Any instantiation of the class, such as jack or jill in the present case, can gain access to the details just by invoking `jack.toString()` or `jill.toString()` respectively.

To make matters even more convenient, Java arranges that if an object identifier is used in a String context WITHOUT explicitly invoking the toString() method then that method will be invoked implicitly.

Thus it is unnecessary to write `jill.toString()` and one can write just plain `jill` as in the second System.out.printf statement.

Try the program out. The results should be:

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

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

JAVA NAMING CONVENTIONS - MORE

It is another Java convention that if a data field name, a method name, or a local variable name consists of two or more words these words are run together but it is arranged that the second and subsequent words begin with an upper-case letter. The name toString follows this convention.

The same convention applies to class names except that the very

first letter is also upper-case, as in ComeIn and ComeAgain.

FINAL VARIATIONS - static or no static

The final experiments on the Box program are intended to illustrate
the effect of the static modifier.

First, simply add an extra System.out.printf statement to the end of
method main() to check that jack is not changed by the instantiation
of jill:

```
public class Box
 { 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);
      System.out.printf("Details of jack...%n%s%n", jack);    // new statement
    }
 }

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());
    }
 }
```

Try the program out. Unsurprisingly, when jack's details are
written out for the second time they are unchanged.

Next modify the declaration of the data field side so that it
is static:

```
public class Box
 { 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);
      System.out.printf("Details of jack...%n%s%n", jack);
    }
 }
```

```
class Square
 { private static int side;                              // static modifier added

   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 and notice that when jack's details are written out for the second time they are the SAME as jill's.

When a data field is declared static it belongs firmly to the class in which it is declared. A fresh version is NOT set up each time a new object is created from the class so there is only one side in the above program. This is set to 6 when jack is created and is set to 5 when jill is created and is not changed again.

INSTANCE VARIABLES AND CLASS VARIABLES

In the first two versions of the current program, the data field side was declared:

```
   public int side;
```

A data field which is declared WITHOUT a static modifier is called an 'instance variable'. It (usefully) exists only after instantiation of the containing class. The variable is then accessible from outside the class via jack.side or jill.side (provided the visibility modifier is public) and from inside the class via this.side (whether the visibility modifier is public or private).

Suppose, instead, the declaration had been:

```
   public static int side;
```

A data field which is declared WITH a static modifier is called a 'class variable'. It exists whether or not the containing class is instantiated and exists only once no matter how many times the class is instantiated.

Had the visibility modifier of side in the latest program been public, the variable would have been accessible from outside the class via jack.side or jill.side OR (using the class name) by Square.side (but in each case the visibility modifier has to be public).

From inside the class, the variable would be accessible via this.side OR (again using the class name) Square.side.

INSTANCE METHODS AND CLASS METHODS

In a similar way, there are instance methods and class methods, methods declared without and with the static modifier respectively.

An instance method (usefully) exists only after instantiation of the containing class. The methods area() and toString() (and also the constructor) are all instance methods. They are accessed via jack.area() or jack.toString() (though the toString() is usually implicit). These methods could not be accessed via Square.area() or Square.toString()

A class method exists whether or not the containing class is instantiated and exists only once no matter how many times the class is instantiated. The method main() is always a class method. In some sense it is accessed (by the Java Virtual Machine) via Box.main() (note the use of the class name Box).

OTHER TASKS

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

5. The Date of Easter Problem

6. The Friday 13th Problem

7. The Forward and Backward Count Problem

8. The Accumulating Rounding Errors Problem