INTRODUCTION TO COMPUTER PROGRAMMING

Richard Pierse

Class 7: Object-Oriented Programming

Introduction

One of the key issues in programming is the reusability of code. Suppose that you have written a program to perform a certain task. Later, you find that you need to write a program to perform a new task, related to but slightly different from the original task. Clearly, it would be efficient if the code written to solve the first problem could be easily modified to solve the new one. Whether this is possible will depend on the way that the code for the first program was originally designed. Object-Oriented Programming (***OOP***) is a way of designing programs that makes them very easy to reuse. *Java* is an object-oriented language so that *Java* programmers are forced to use *OOP* concepts in designing their programs.

Object-oriented design is based on a few very abstract concepts: ***objects***, ***classes*** and ***inheritance***. Initially, these ideas may seem hard to relate to the practical business of computer programming. However, once you get used to them, you will find that *OOP* concepts will help you to write well-structured, reusable code.

Objects

Look around you and you will see lots of examples of real-world objects: your computer, your desk, your class mates. These real world objects share two characteristics: *states* and *behaviours*. States of an object are pieces of information associated with them. For a computer, states include its make (IBM, Dell, Viglen), CPU speed, colour, whether switched on/off etc. For your class mates, states include their age, height, hair colour, whether standing up or sitting down etc. The behaviour of an object is what an object does or what can be done to an object. Behaviours associated with a computer include switching on/off, pressing a key, inserting a CD etc. Behaviours of your class mates include breathing, talking, eating, sleeping etc. An object is completely defined by its states and behaviour.

Software Objects

Software objects are modelled after real-world objects in that they have both states and behaviour. A software object maintains its states in ***variables***. A software object implements its behaviour with ***methods***: functions (subroutines) associated with an object that do something to the object. A software object is completely defined by its variables and its methods.

Classes

In the real-world, there are often many objects of the same kind; for example, your computer is one of many computers in this class room. In object-oriented terminology, your computer is a particular example of the ***class*** of objects known as computers. All objects in the class of computer objects will share the same behaviour but may have different states, for example your computer may be switched off while another computer is switched on.

Similarly, a software object is a particular example (or ***instance***) of a ***class*** of objects. The class is a blueprint or prototype that defines the variables and the methods that are common to all objects in that class. Once the class has been defined, many objects of the same class can be created, each sharing the same behaviour but differing in their states.

Inheritance

There will be relationships between some classes of objects. For example, the class of computers is a sub-class of a more general class of electrical appliances. All electrical appliances (including computers) will share some common states (colour, whether on or off) and some behaviours (turning on or off) but will have other states and behaviours that are appliance-specific. The common states and methods are said to be ***inherited*** by all sub-classes.

A software class may be defined as a sub-class (or ***extension***) of a more general class. In this case it will ***inherit*** all the states and methods of its mother class. However, it may add new states and methods and may over-ride some of its inherited methods to provide specialised implementations of those methods.

A *Java* Example

Here is an example of some *Java* code to define a class called `Box`. This class has three variables: `length`, `width`, and `colour` and three methods: `area()`, `setcolour()` and `setsize()`.

```
public class Box
{
     float length,width;
     int colour;
     public int area() {return width*length;}
     public void setcolour (int mycolour) {
          colour=mycolour;
     }
     public void setsize {int w, int l) {
          width=w; length=l;
     }
}
```

The first line of this code declares the name of the class. This is a new class that does not inherit from any other class. The keyword **`public`** means that the class is visible and therefore accessible to other classes. Note that all three methods in this class are also declared public. It is possible to hide both classes and methods from public view so that they are not accessible. The keyword **`private`** means that a method is not accessible outside its class. (Clearly, if a class is declared as **`private`** it will never be accessible at all). The keyword **`protected`** means that a class or method is only accessible within its class or by classes derived from it (i.e. sub-classes).

The three methods `area()`, `setcolour()` and `setsize()` are functions that operate on objects of the class `Box`. The first method, `area`, takes no arguments but returns an integer value (the area of the box) so is declared as type **`int`** . The second method, `setcolour`, takes one argument but has no returns and so is declared as **`void`** . The third method, `setsize`, is also declared as type **`void`** since it takes two arguments but does not return any value.

Defining Inherited Classes

Suppose now that we want to create a new class, `Square`, for square boxes for which `length = width`. This class is clearly a sub-class of `Box` and will inherit most of its variables and methods. The class definition is simply:

```
public class Square extends Box
{
    public void setsize {int s) {width=length=s;}
}
```

The keyword **`extends`** means that class `Square` inherits all its variables and methods from class `Box`. The only variables or methods which need to be defined are any new ones that aren't already in the class `Box` plus any that we need to modify in the new class. In this case, we want to modify the method `setsize()` so that it only takes a single argument and sets both variables `width` and `length` to the same value.

Class Constructors

Each class has a special method called a ***constructor*** that gets called every time a new object of that class is created. The purpose of the constructor is to do any initialisation that needs to be done. For example, in our `Box` class, the class variables `length`, `width`, and `colour` are not initialised. We could initialise them, each time a new `Box` object is created, by including the following constructor within the class definition:

```
    public Box() {
        length=10; width=20; colour=7;
    }
```

The class constructor always has the same name as the class and has no declared type. The constructor can take arguments that are passed to it when a new object is created, as in

```
    public Box (int l, int w, int col) {
        length=l; width=w; colour=col;
    }
```

If a class constructor is not explicitly defined, a null one is used.

Java Objects

Once a class has been defined, objects of that class can be created using the **`new`** command. For example, the line

```
Box box1,box2; Square sq3;
```

declares two objects `box1` and `box2` of the class `Box` and one object `sq3` of the class `Square`. Then the lines

```
box1 = new Box();
box2 = new Box();
sq3  = new Square();
```

actually create these three new objects. Once the objects have been created, their (public) methods and variables can be accessed using the dot operator. For example:

```
box1.setsize(15,20);
```

invokes the `setsize()` method to set the size of `box1` and

```
area3=sq3.area();
```

returns the area of the object `sq3` in the variable `area3`.

Similarly,

```
box2.length=50;
```

sets the class variable `length` for object `box2` to the value `50`.

Example Program - Redux

Consider again the example program from the end of last week's class, which we are now in a better position to understand:

```
public class MyProg
{
     public MyProg() {}
     public static void main (String args[]) {
          int i;
          for(i=10;i>=1;--i) {
               System.out.println(i);
          }
          System.out.println(”Blast Off!”);
          System.exit(0);
     }
}
```

This program defines a new class called `MyProg` with two public methods: `MyProg()` and `main()`. This class is the application class. The first method is the class constructor (which actually does nothing here). The second is the special method

```
     public static void main (String args[])
```

This is declared as type **`void`** because it returns no value. It takes, as single argument, a **`String`** array `args[]` (used to hold any command line parameters passed to the application).

Note that `main()` is declared with the keyword **`static`**. This keyword, when applied to a method, means that the method can be called independently of an instance of a class. When the *Java* virtual machine loads an application, it will expect to find a `main()` method in the application class, which it will then execute.

In the example, the method `main()` uses two methods from the class `System`: `System.exit()` and `System.out.println()`. This is a pre-defined *Java* class. Note that we do not have to create an object of this class before calling these methods since all methods of the class `System` are **`static`**.

Using the *Java Swing Library*

The real power of *OOP* comes from the ability to use and extend pre-defined classes. Here is an example of an application using the classes defined in the Java Swing library.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class SwingApplication {
    static String Prefix = "Number of button clicks: ";
    int numClicks = 0;
    public Component createComponents() {
        final JLabel label = new JLabel(Prefix + "0    ");

        JButton button = new JButton("I'm a button!");
        button.setMnemonic(KeyEvent.VK_I);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                numClicks++;
                label.setText(Prefix + numClicks);
            }
        });
        label.setLabelFor(button);

        JPanel pane = new JPanel();
        pane.setBorder(BorderFactory.createEmptyBorder(
                             30,30,10,30) );
        pane.setLayout(new GridLayout(0,1));
        pane.add(button);
        pane.add(label);
        return pane;
    }
    public static void main(String[] args) {
          JFrame frame; SwingApplication app;
          Component contents;
          frame = new JFrame("SwingApplication");
          app = new SwingApplication();
          contents = app.createComponents();
          frame.getContentPane().add(contents,
                         BorderLayout.CENTER);

        // Finish setting up the frame, and show it.
          frame.addWindowListener(new WindowAdapter() {
              public void windowClosing(WindowEvent e)
               { System.exit(0);}
          });
        frame.pack();
        frame.setVisible(true);
    }
}
```

Making Sense of the Program

- The `main()` method creates the objects `frame` and `app`

- `frame` is an instance of the swing library class `JFrame` which defines a frame class for a user window with controls.

- `app` is an instance of the program's own application class `SwingApplication`. This is needed so that `main` can call the method `createComponents()`.

- `createComponents()` itself creates instances of the swing classes `JButton`, `JLabel` and `JPanel`, which are the Windows controls (a button with label and the panel in which to place it) that will appear in the window. These controls are then initialised using methods defined for the swing classes.

- Finally, the `JFrame` window is displayed.

When the program runs, it displays a button which counts the number of button clicks.

The program can be downloaded from my web page.