Object oriented programming for pedestrians.

Colin G. Johnson.

Object-oriented programming for pedestrians.

Java edition.

Colin G. Johnson.
Department of Computer Science,
University of Exeter, The Old Library,
Prince of Wales Road,
Exeter, EX4 4PT.
Telephone : 01392 264076
Fax : 01392 264067
Email : `colin@dcs.ex.ac.uk`

Version 1.0

September 29, 1998

1 Some initial waffle.

"For when you are put into the Vortex you are given just one momentary glimpse of the entire unimaginable infinity of creation, and somewhere in it a tiny little marker, a microscopic little dot, which says 'You are here.'"

— Douglas Adams

Welcome to *Object-Oriented Programming for Pedestrians*, my notes for the first few weeks of the programming course. There used to be a book called *Linear Algebra for Pedestrians* which I thought was a good title, so I've stolen the idea. The philosophy is somewhat akin to "do not run before you can walk"[1].

The aim of these notes is to explain the basics of computer programming, using the ideas of *object oriented programming* and a computer *language* called *Java*. There are plenty of books out there on these things, but this aims to provide a contrast to those books. Firstly it is short. It is possible to buy many 1000 or more page books on programming[2], but most of them are tedious, many of them are expensive, and those which aren't are too heavy to carry around, meaning that they get left at home when you really need them. These notes are not long. But they are hopefully long enough.

Secondly many textbooks assume that you know what you are doing or what you want to know already. These notes are more choosy and directed, and are not designed to be everything for everyone—they are designed to be a short introduction for intelligent people who don't know anything about computer programming, who are prepared to work hard and who don't want to be patronised. If you have some programming experience before then you can skim through bits of the notes and compare with your past experiences.

1.1 Some textbooks.

"of making many books there is no end"

— King James Bible, Ecclesiastes:12:12

Of course these notes don't cover everything. Some other books and things you might care to have a look at.

[1]as opposed to certain books which seems to attempt to teach you how to land before teaching you how to fall over.

[2]Sometimes called "ten-pound doorstops".

- *Java Gently*, Judy Bishop. This is probably the best of the books which aim to teach *Java* from scratch without any previous knowledge.
- *Teach Yourself Java*, Chris Wright. This is a slim, cheap book which starts from assuming nothing. Regrettably it rapidly gets onto some more advanced stuff about interface construction and things before doing too far into the details of the language. Nonetheless it is a useful book and covers some of the topics which will be covered in Media Computing next term.
- *Introduction to Programming Using Java*, David Arnow and Gerald Weiss. Yet another good book for beginners, containing some nice nontrivial examples.
- *Exploring Java*, Pat Niemeyer and Joshua Peck. This is a (very good) more advanced book which could be useful to those who already have programming experience.

There are also some World-Wide Web site of interest :

- *The Java tutorial*, `http://www.dcs.ex.ac.uk/support/software/java/tutorial/` is a very good set of notes on *Java*, better than many of the books that are available.
- *The Java API Index*, `http://www.dcs.ex.ac.uk/support/software/java/sgi/api/packages/` is a useful *reference* guide to many of the features of *Java*, but is very cryptic for beginners. Something to bookmark for future reference.
- *Java Programmers' Frequently Asked Questions*, Peter van der Linden, `http://www.best.com/˜pvdl/javafaq.txt` is a useful charivari of odd tips and tricks, ranging from the elementary to the advanced.
- *JavaWorld Book Catalog*, `http://www.javaworld.com/javaworld/books/jw-books-index.html` is an exhaustive (and somewhat exhausting) list of books about Java and related topics.
- *Java Learning Centre*, `http://www.firststep.com.au/jlc/` is a mixed collection of articles about learning *Java* and related topics.

I shall put additional comments about books and web sites and any other useful information on a world-wide web site, at

`http://www.dcs.ex.ac.uk/˜colin/courses/programming.html`

If you have any suggestions for additional things that could be added to the page, email me at `C.G.Johnson@ex.ac.uk`.

2 Computing as modelling the world.

Okay then, that's all the waffle out of the way. Where are we going? And what are we doing? A basic philosophy here is that *computer programming is creating an abstract model of the world*. Not the whole world. Not necessarily the "real" world (we could model the world of mathematics, or the world of the English language, or some other platonic reality). But we are attempting to create some abstract simulation of things that are happening in this "world".

2.1 Abstract? Model?

What do we mean by creating an abstract model of the world? Essentially we have a problem that we would like to solve, couched in real world terms. To solve this problem we must first *abstract*[3] the relevant features, and *show deliberate ignorance* of the rest. For example if we are creating a model of the national lottery draw, we can ignore features like "the weather". However we need to model ideas like "the numbers". This procedure could be called "design" or "specification" or "analysis". Once we have decided which things we need in this model we then write down a fairly formal description of the properties of the parts of the model, some description of the behaviour of the parts of the model, and some sequence of instructions which encapsulates what we do when we solve the problem.

The next stage is to translate that model of the world into instructions which can be understood by the computer. This involves taking our abstract design and formalizing it into statements in a *computer language*. In this course we shall use a language called *Java* which is fairly new and rather trendy and full of bad puns about coffee. Using this sequence of instructions, the *computer program*[4], we can solve the problem.

All this is summarized in a pretty picture, figure 1.

But hark! Our solution is still stuck in the abstract, translated version of the world. So to get a solution back we must provide some means for users of the program (that is, people who *use* the finished program to solve problems rather than the *programmer* who creates the model and translates it into the computer language) to interact with the program. This is provided for in the language by *input* and *output* functions, which enable aspects of the real world to be encoded and decoded.

Finally there is a stage of *feedback*. In testing the computer program there may be inconsistencies with the abstract model, in which case we must interpret these errors as mistakes in the translation process (the act of *programming* or *coding*) and correct them. Similarly once these have been ironed

[3]as in the verb "to abstract"

[4]**Pedantic aside number 1.** *Program* is the canonical spelling of the word in all variants of English, though occasionally you see *programme*, which looks wrong to me in this context.

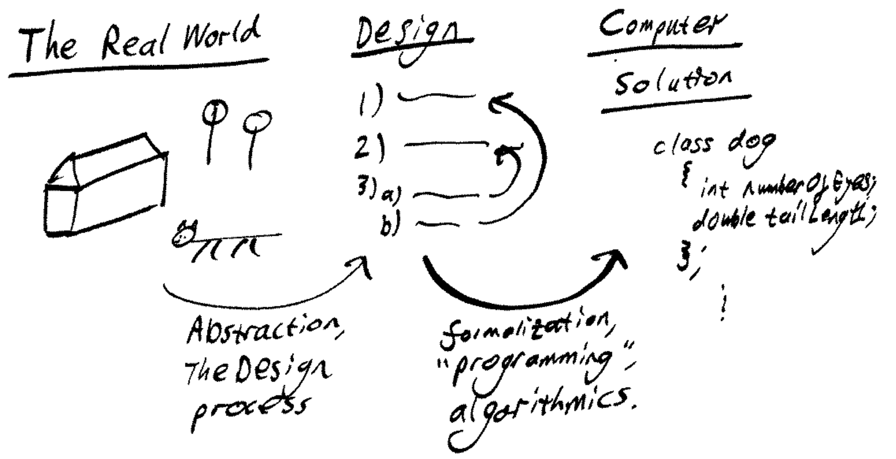


Figure 1: Modelling the world : version 1.

out there may be mistakes in the *model* itself, in which case we must remodel the system based on the discrepancies between the model and the real world. This is encapsulated in this poem :

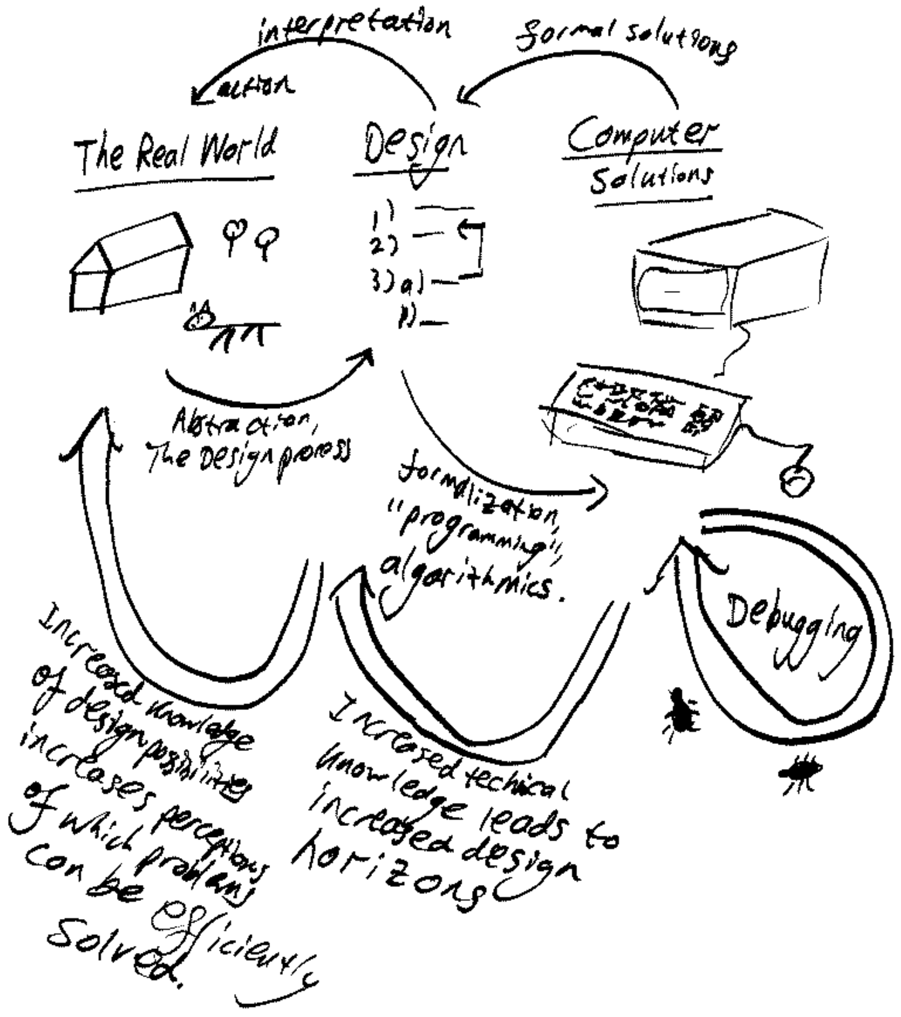
The road to wisdom
why—its plain and simple to express :
you err
and err
and err again
but less
and less
and less

—Piet Hein

All of these features are in the new, improved version of the diagram (figure 2). The single arrows represent the "normal" flow of activity, whereas the double arrows represent the "correction and debugging" stage, which also has positive connotations as a "learning" stage. More poetry:

Problems worthy
of attack
prove their worth
by hitting back.

—Piet Hein



interpretation
action
formal solutions
The Real World
Design
Computer Solutions
1)
2)
3) a)
b)
Abstraction, The Design process
formalization, "programming", algorithmics.
Debugging
Increased knowledge of design possibilities increases perceptions of which problems can be efficiently solved.
Increased techical knowledge leads to increased design horizons

Figure 2: Modelling the world : version 2.

2.2 An example.

Consider the computer system used in the library. Within that system there are a large number of things that need to be represented, including books, newspapers, students, and concepts within or interlinking those ideas, such as the idea of borrowing or the idea of a book being missing. The first stage in design is to decide how to structure that information—when someone borrows a book, do we transfer the information about that book to that person's "file", do we transfer information about that person onto the information about that book, or do we create a new set of data about links between people and books? All are valid ways of working, and only a mixture of experience, carefully thinking about the problem and trial-and-error can help you to do make the right choice.

Now what happens when someone tries to borrow a book. We can formalize this as follows :

```
Type in the borrower number
If the borrower has any fines, get them to pay those fines.
    If they can't, send them away.
If the borrower already has too many books out, get them to re-
  turn books first.
    If they can't, send them away.
Type in the book number.
If the book is a reference book, don't allow it to be taken out
    otherwise, assign the book to the borrower.
Demagnetize the security device on the book.
Give the book to the borrower.
```

This is a *sequence* of instructions, deliberately so, as most computer languages rely on the concept of a sequence of operations as the main way of describing what is going on.

2.3 Pencil-and-paper exercises.

"Why do we need to think? Can't we just sit here and go BLBLBLBLBLB with our lips for a bit?"

— Douglas Adams

1. Write out a careful sequence of instructions for boiling an egg.
2. Describe what (you think) happens when you go to a bank cash machine and ask for your bank balance, or ask to withdraw some money. Assume that you are using a machine at a branch other than your main branch.
3. Describe what you think happens inside the machine when you buy a train ticket from a ticket machine.

3 Object-oriented programming.

"Design and programming are human activities : forget that and all is lost."

— Bjarne Stroustrup.

Hey! The title says *object-oriented programming*. What's all that? Essentially object-oriented programming means that we create our model by firstly building a model of the types of things we want to use in developing our model (the *classes*) and then creating (*instantiating*) particular versions of those classes—these are called the *objects*.

3.1 Classes and objects.

"THINGS are all around us. Everywhere you go you see THINGS. THINGS are a necessary part of your life. And now in this special offer, we can give you THINGS at a fraction of the normal cost. Yes, cheap things are finally available direct to you, the housewife / barrister / chiropodist for only £9.99 per THING. Simply quote your credit-card number and say how many THINGS you need."

— *Matthew Jarron*

Here are some examples

- We are developing a program to calculate the accounts for a small business. The *classes* are the categories of things in the system—e.g. the concept of a salary, the concept of money itself, the concept of an end-of-month balance. We begin our program by defining what we mean by these ideas. When we come to create a particular record for you we create a set of *objects*—*your* salary, *your* balance, et cetera. Think of some more examples yourself.
- We are developing a program to allow the user to draw shapes on the screen. The classes are the categories of things in the system—circles, squares, polygons, curves et cetera. We can define certain things at the class level to do with global properties of, say, circles—a circle has a centre, a diameter, a colour, and we can do certain things to circles, such as change the diameter, find out what colour it is. However when we create a circle object we specify exact values of these things—we create a circle of diameter 3cm, centred at the origin, and shaded in a particular bright red colour.

Defining the classes we will use in our model is the first stage in designing a program. We Look at what categories of things are found in our real-world program and carefully define them. We then think about what particular examples of those things are found, and create, structure and manipulate those objects in order to solve our problem. In the next section there are a number of exercises with which you can develop this way of looking at the world.

3.2 Pencil-and-paper exercises.

Identify important classes and objects in the following systems.

1. A library computer system.
2. The computer system for a bank.
3. An arcade game.
4. A computer adventure game like *Myst*.
5. A word-processing program.
6. A program that schedules deliveries of goods to supermarkets.
7. A holiday booking system for a travel agent.
8. A spreadsheet program list *Excel*.
9. The database of students in the university.
10. A calculator program.
11. A world-wide web browser.

3.3 Choosing classes

It is noticeable from the two examples above that sometimes classes can relate to each other. For example in keeping a record for an item in the accounts system there could be a buying price and a selling price, both of which are of class *Price*. If we have a system for drawing shapes we might want to define some general properties at the "shape" level, e.g. colour and location, and then specify some more specific details at the level below, such as "diameter" for a circle of the angles in a triangle. This idea of a *hierarchy* of classes will be important later on.

There is a nice description of how to find classes in Bjarne Stroustrup's book *The C++ Programming Language*.

> The best place to start looking is the application itself—as opposed to looking in the computer scientist's bag of abstractions and concepts. Listen to someone who will become an expert user of the system once it has been built and to someone who is a somewhat dissatisfied user of the system being replaced. Note the vocabulary used.
>
> It is often said that the nouns will correspond to the classes and objects needed in the program; often that is indeed the case. However, this is by no means the end of the story: Verbs may

> denote operations on objects, traditional (global) functions that produce new values based on the value of their arguments, or even classes. …Verbs such as "iterate" or "commit" can be represented by an iterator object and an object representing a database commit operation, respectively. Even adjectives can often usefully be represented by classes. Consider, the adjectives "storable", "concurrent", "registered", "bounded." These may be classes intended to allow a designer or programmer to pick and choose among desirable attributes for later designed classes by specifying virtual base classes.
>
> The best tool for finding these initial key concepts/classes is a blackboard. The best method for their initial refinement is discussion with friends. Discussion is necessary to develop a viable initial vocabulary and conceptual framework. Few people can do that alone.

The whole of the chapter from which that excerpt is taken is valuable reading. I shall place a couple of copies in the common room.

4 Practicalities.

"In order to learn to program, you have to program."

— *Chris Wright.*

Right then time to look at the programming process from the point of view of sitting at a computer and attempting to do it, rather than thinking in abstract terms.

To solve a problem using the computer we need to write a list of instructions in a *computer language*—a way of giving instructions to the computer which is unambiguous and complete. The language we shall use here is called *Java* and has the advantage (amongst many) that programs written in the language can be used on many different kinds of computer. Thus you should be able to run the programs that you write on both the PCs and Unix machines in the department. If you have a computer at home you will probably be able to install and run *Java* there too—see section 16.

A program consists of a sequence of *class definitions* which give an abstract representation of the classes defined for the problem. There is also a special class, which we shall call the *application class*, which represents the concept of "the whole problem" and controls the flow of the solution procedure.

Simple problems require just the application class. Here is a very simple program.

```
import java.io.*;

class HelloWorldApp
{
   public static void main(String[] args)
      {
         System.out.println("Hello World!");
         // display the message
      }
}
```

This program prints the phrase "Hello World" onto the screen of the computer. In this section we shall first take the above program as given and learn how to get the computer to carry out the instructions. We shall then describe what is happening as the computer runs this program.

4.1 Running the simple program.

Firstly we need to get the program into the computer. I assume that you are familiar with the concept of a *text editor* from other parts of your course. Use the text editor to create a file called `HelloWorldApp.java`, and type the above program text into the file, exactly as above. Save the file.

Now return to the *Unix* command line and type the command

```
javac HelloWorldApp.java
```

This instructs the computer to *compile* the program—that is to convert the program from the structured, human-readable code above into a system of instructions (the *Java ByteCode*) which the computer can carry out quickly. If this doesn't work then carefully check that you have typed the program in as above, that you are still in the same directory where you began the program, and that you have typed the commands as indicated.

To actually get the computer to carry out the instructions, type

```
java HelloWorldApp
```

This should cause the computer to print "Hello World!" onto the screen. Again, if it doesn't work check the thinks mentioned above.

4.2 Anatomy of the simple program.

By now you should have managed to get the program to work. Here is a brief explanation of what is going on. The first thing to know is that the computer (unless instructed otherwise) reads the program from top to bottom, reading and understanding each "line" at a time. The first line, `import java.io.*;` "imports" another program which sets up the computer to do a particular thing. In this case the thing is input and output (hence `io`). The second instruction in the program reads `class HelloWorldApp`. This creates a *class*, an abstract representation of something. In this case the class is representing the *program itself*, rather than something from the real world.

The contents of the class are enclosed between pairs of curly brackets `{}`. This class can do one thing[5]—called `main`. The `main` method has a special status within the program, as the instructions within `main` are the first things to be carried out by the computer when it runs the program. Within this `main` structure there is one action—the key word here is `println`, which stand for "print line". This is a command to the computer to print the text that follows between the quotation marks. The next line of the program, after the symbol `//`, is a *comment*—something to help human readers of the program understand what is going on, rather than instructions to the program itself.

This simple program is not the best way to learn the concept of a class—the thing it represents (the program itself) is much more abstract than the things that a class normally represents. More typical examples will be given below.

4.3 Computer exercises.

1. Modify the simple program to display your name instead of "Hello World!".
2. Modify it so that it displays "Hello" on one line then "goodbye" on the next line.
3. After you have compiled the program make a listing of the files in the directory using the `ls` command on the *Unix* command line. What new file is created by the compilation process?

5 Program structure in Java.

Before we describe what other things can go into a program, we shall spend a few paragraphs discussing how a program is structured. Some of this won't fit into place until later on, but we shall keep it all in one place for reference.

We shall assume for now that our programs occupy a single file on the computer's disk. A typical program file contains a list of *class definitions*, which begin with the keyword `class` and a name, and are then contained within curly brackets. For example in the "hello world" program above,

[5]in the terminology introduced below, it has one *method*

there is one class, being the application class. Have a look through the notes at other complete programs to see more complex examples.

Sometimes the beginning of the program file contains a number of `import` statements, which define other classes which have been written by the creators of the *Java* language. An example of this is the `import java.io.*;` command at the beginning of the "hello world" program. Have a look at the *Java API Index* that you bookmarked in section 1.1 on the world-wide web.

5.1 Breaking the program into blocks.

"All good things in life have an end—except sausages, which have two."

— *Miles Reid.*

The beginning and end of a class are determined by pairs of *curly brackets*. These define a block of statements which are treated in some way or other as a unit. For example in the "hello world" program above, brackets are used to contain the whole of the *application class*, and they are used to contain the whole of the "main" section of the program.

Individual statements are separated by semicolons (;). Once a particular statement has been finished you place a semicolon to tell the computer to go onto the beginning of the next statement.

5.2 Comments.

We said earlier that the text after the message above. i.e. `//display the message` was a *comment* to aid humans who are reading and working on the program, rather than to give instructions to the computer. To create such a comment, type two backslashes `//`, then the computer will ignore anything between the backslashes and the end of that line.

Another way of creating comments is to enclose text between pairs of `/*` and `*/` symbols. Anything written between a pair of these, even if it is spread over several lines, is also ignored by the computer.

It is important to include comments in a program if you are going to understand what is meant by it when you return to working on it after even a short while. Useful things to comment include : the meaning of variable names, the input and output formats of methods, descriptions of what a particular few lines of code achieve, indications of how a particular part of the program might be changed in the future, references to algorithms taken from textbooks or other sources, the name and contact details for the programmer, copyright statements, and so on and so on.

Another useful facility is the *javadoc* program, which allows you to embed documentation for the program within the code itself, a concept called *literate programming*. By running the *javadoc* program on an appropriately commented *Java* file, you can generate HTML documentation for that code.

5.3 Beginnings.

"There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarrely inexplicable. There is another theory which states that this has already happened."

— *Douglas Adams*

It has been noted above that the computer starts at the first statement in the program and progresses down, line by line, through the program. But where does it start? Every program should have an *application class*, conventionally given a name ending in `App`, e.g. `HelloWorldApp`.

Once you have created the file, you type `javac` and the filename to *compile* the program. This creates an application file which you can run.

To run the program you type `java` at the *Unix* command line, followed by the name of the application class. So you typed `java HelloWorldApp` above to run the executable program called `HelloWorldApp`, which is stored in the file `HelloWorldApp.class`.

Within this class there should be a *method* (see section 11) called `main`. The program begins at the first statement in the block following the `main` declaration.

5.4 Pencil-and-paper exercise.

Look at the program in the next section. What would you type to compile and run it? Which would be the first statement executed by the program? What is the application class called?

5.5 Example.

Here is an example of a more complex program demonstrating the features above. The details won't make much sense yet, but look through it for the general structural features described above.

```
import java.io.*;

class Farm
{
  /* contains data for a farm */
  String farmerName;
  int numberOfCows;
  int numberOfPigs;
  int balesOfHay;

  public void newCowArrives()
  {
    /* called when a new cow is bought */
    numberOfCows = numberOfCows+1;
  }

  public void newPigArrives()
  {
    /* called when a new pig is bought */
    numberOfPigs = numberOfPigs+1;
  }

  public void makingHay(int howMuchHay)
  {
    // adds howMuchHay to stock
    balesOfHay = balesOfHay+howMuchHay;
  }

  public String toString()
  {
    // returns string summarizing farm
    return "Farmer "+farmerName+"'s farm has "+
       numberOfCows+" cows and "+numberOfPigs+
       " pigs and "+balesOfHay+" bales of hay";
  }
}
```

Type this into a file called `Farm.java` and compile it.

```
class FarmApp
{
  public static void main(String args[])
  {
    /* set up a farm */

    Farm homeFarm = new Farm();
    homeFarm.farmerName = "Giles";
    homeFarm.numberOfPigs = 14;
    homeFarm.numberOfCows = 12;
```

```
        homeFarm.balesOfHay = 80;

        /* print out information about the farm */

        System.out.println(homeFarm.toString());

        /* new animals are bought */

        System.out.println("A cow and a pig are bought");
        homeFarm.newCowArrives();
        homeFarm.newPigArrives();

        /* print out information about the farm */

        System.out.println(homeFarm.toString());

        /* more hay is made */

        System.out.println("More hay is gathered");
        homeFarm.makingHay(25);

        /* print out information about the farm */

        System.out.println(homeFarm.toString());
    }
}
```

Call this `FarmApp.java`, compile it, then get it to run. Try to work out what is (broadly) going on. Add some more comments as you discover what is going on. If you can work out how to, change the name of the farmer.

5.6 Pencil-and-paper exercises.

1. Look through the programs in the notes and in any *Java*-based programming textbook. Find examples of the use of curly brackets, comments, and semicolons in those programs.

6 Input and output.

In the previous section we discussed the general shape of a program in *Java*. This will be the first of several sections scattered through these notes where we explain one specific idea and how to implement it in *Java*.

We have already seen a simple statement for output. A line in the simple program above reads `System.out.println("Hello World!");`. Let us look at that more carefully. The first word, `System`, tells the computer to use the standard "system" output and input, which is in our case screen (a particular window on the screen, displaying text information) and keyboard. The second word, `out`, specifies that this is *output*—i.e. information is flowing from the computer to the user. The final part of the statement is `println`, which says what kind of output operation is to happen, which in this case is that a line will be printed, then at the end of printing that line the computer will move on to the next line.

The bit in the parentheses[6] after the instructions describes what these instructions are meant to act on. In this case the action of printing the line to the screen acts on the text "Hello World", which is enclosed in quotation marks which don't get printed. Finally the line ends with a semicolon which indicates that that statement is finished.

6.1 Computer exercises.

1. Replace the commented sections in the following program with appropriate *Java* statements, then insert appropriate comments to describe what the program is doing. Try to do this from memory at first, then look back at the notes to see if you have made any mistakes.

```
/* set up the program for input and output */

/* declare the application class */
{
  /* declare the main method */
    {
       /* print a line with you name on it */
       /* print a blank line */
       /* print something else */
    }
}
```

6.2 Command-line Input.

Now what about *input*—reading information of some kind into the computer program? This is more complicated, as we need to specify where the

[6]**Pedantic footnote number 2**. These are parentheses : (). These are brackets : []. These are curly brackets (sometimes called "braces") :{} (and to be idiosyncratic these " are "rabbit ears" and this ! is a "pling")

inputted information is to be stored. Think about typing into a program like a word processor or spreadsheet—each time you press a key the computer needs to keep track of what key was pressed. To handle this in a convenient, abstract way we use things called *variables*. Exact rules about variable names are given in section 8.1, for now we shall use short phrases like `luckyNumber` or `myBirthday` to represent things in the program. There is one variable in the `HelloWorld.java` program above, called `args` (for *arguments* (in the mathematical sense!)). We didn't make any use of it above, but we shall use it now.

The simplest kind of input is to type something in when you start the program running. Type in the following program (call the file `MyNameApp.java`), and compile as usual using the statement `javac MyNameApp.java`

```
import java.io.*;

class MyNameApp
{
   public static void main(String[] args)
      {
         System.out.println("My name is "+args[0]);
      }
}
```

Now run the program in the following way. Type `java MyNameApp Colin` (or whatever your name is). The computer should display `My name is Colin` (or whatever). If it doesn't work first time check through the file and the various programming stages until it is working okay. The output is partially hard-coded into the program, i.e. `"My name is "`, and partially read from the command line, which is represented inside the computer by `args[0]`, `args[1]`, et cetera, each representing a word. The + sign between these two strings of text tells the computer to run them together as a single piece of text.

6.3 Computer exercises.

1. Change `MyNameApp` so that when you type in something like `java MyNameApp Colin Johnson` it displays both names.

2. Fill in the blanks in the following program. Note that some of the statements take up multiple lines—remember that a statement is only ended when you write a semicolon.

```
/* set the computer up for input and output */;

/* declare a class called lengthApp */;
{
   public static void main(String args[])
     {
        int a,b,total;
        a = /* the first word inputted
            via the command line */.length();
        b = /* the second word inputted
               via the command line */.length();
        total = a+b;
        /* Print out the length of each word,
           followed by the total length */;
     }
}
```

   Now get the program to run. It should display the total length of both words inputted.

6.4 Inputting strings of text.

We would like to be able to do more complicated kinds of input. There are various ways of interacting with a computer, including typing in text, clicking on buttons, and selecting from a menu. For now we shall concentrate on typing text into the keyboard and processing that.

When carrying out input we have to indicate to the user that some kind of input is required, we have to have some signal to indicate that input has finished, and we need to deal with the many different input possibilities in a sensible way.

Let us begin with a comparatively simple input scenario. We would like to have the computer ask the user to type in their name, to read it in, and then repeat it to them. This program is called `InputNameApp.java`.

```
import java.io.*;
```

```
class InputNameApp
{
   public static void main(String[] args)
       throws java.io.IOException
     {
       /* set up the facility to read from the keyboard */

       BufferedReader keyboard
          = new BufferedReader(new InputStreamReader(System.in));
       String name;

       /* read in, then print out, the name */

       System.out.println("Please Type your name,"
                          +" then press the return key.");
       name = keyboard.readLine();
       System.out.println("Hello, "+name);
     }
}
```

Type in the program and get it running.

There are a number of things to note in the program. The first is the line after the line containing the word `main`, reading `throws java.io.IOException`. This is an error handling mechanism which is mandatory when using input statements—however this should be taken as read for now, details will be given later.

The first interesting statement is

```
       BufferedReader keyboard
         = new BufferedReader(new InputStreamReader(System.in));
```

This statement creates an object in the computer's memory called a `BufferedReader` called `keyboard`, which holds ("buffers") lines of text, which is gets from an object called an `InputStreamReader`, which takes its text from the keyboard, the `System.in`. In the next line, `String name;`, we create an object of type `String` which is called `name` and which is capable of holding a piece of text[7].

So far we have only set up the things needed to carry out input. The line which actually causes input to happen reads

```
       name = keyboard.readLine();
```

this passes a message "read in a line of text" to the keyboard, which waits for an input, terminated by a carriage-return, and then stores it in the `String` called `name`.

[7]Pieces of text are traditionally called "strings" because they are strings of letters and spaces and numbers and other characters

6.5 Outputting the contents of variables.

The final nontrivial line in the program above is the line which prints the name and the text "Hello". This is a `println` statement which prints two things. Firstly it prints a string, `"Hello, "`. This is followed by (indicated by the plus sign) the *contents* of the variable called `name`.

This is why we need the quotation marks. Within the parameter of a `println` method, we can print a mixture of fixed strings, which are begun and ended by pairs of quotation marks, and variables, which are indicated by giving their name. The style in which variables print out is defined by the class definition, and when you come to create your own variable types you can define how they will print (see section 11.7 below).

6.6 Computer exercises.

1. Write a program which accepts as input five words (on five separate lines) and then displays them in reverse order to the order in which they have been input.
2. Write a program which accepts as input someone's name, address and age, on separate lines, in that order, and which then prints a line with name and age on a single line, followed by address on the following line.

6.7 Dealing with string input.

We may want to do more with input strings than simply store them and regurgitate them whole. One such thing might be to split a line of text into individual words, e.g. in a spell-checking program. To do this we use an object called a `StringTokenizer`. To set the program up to use the `StringTokenizer` we need to include the line

```
import java.util.StringTokenizer;
```

at the top of the program file.

The `StringTokenizer` is initialized with an input string, which it then splits down into individual words. A word is defined as everything between two space characters, or similar *white space* characters such as tab, carriage-return and newline[8].

Here is an example (call this `stringChopper.java`). Type this in and get it to work.

[8]you can declare it so that it uses another character as the *delimiter* which separates the words.

```
import java.io.*;
import java.util.StringTokenizer;

class StringChopperApp
{
  public static void main(String args[])
        throws java.io.IOException
  {
        /* set up the computer to read from the keyboard */

        BufferedReader keyboard
           = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer splitString;
        String line;

        /* read in some text */

        System.out.println("Please type a line of text"+
           " at least five words long"+
           " then press the return key.");
        line = keyboard.readLine();
        splitString = new StringTokenizer(line);

        /* count the words and display some of them */

        System.out.println("There are "+
                            splitString.countTokens()+
   " words in your line.");

        System.out.println("First : "+splitString.nextToken());
        splitString.nextToken();
        System.out.println("Third : "+splitString.nextToken());
        splitString.nextToken();
        System.out.println("Fifth : "+splitString.nextToken());
  }
}
```

The program creates a `StringTokenizer` object called `splitString` which stores the split-down string. This object is declared in the line

```
        StringTokenizer splitString;
```

and after we have read in a line, we initialize this object using the `String` that you have typed in.

```
        splitString = new StringTokenizer(line);
```

Now there are a number of things we can do with `splitString`. The first is that we can send it a message asking it to tell us how many words (tokens) it contains. We then use `println` to print this out.

```
System.out.println("There are "+
                   splitString.countTokens()+
                   " words in your line.");
```

The other thing we can do is to print out some selected words from the string. When we initialize `splitString` it is "looking at" the first word in the string. We send it the message `nextToken()` which says "send us the word you're currently looking at and move onto the next word". In This case we do this five times. However it is only during the first, third and fifth times that we feed the input to the `println` statement—so only the first, third and fifth words are printed out.

6.8 Computer exercises.

1. Read the description of the `StringTokenizer` class at

   ```
   http://www.dcs.ex.ac.uk/support/
   software/java/sgi/api/
   java.util.StringTokenizer.html
   ```

2. Implement a program which accepts a (two word) name, (one word) town and an age on a single line and writes them each on a different line.

3. Implement a program which inputs a line of text and prints it out backwards (that is, the words are in reverse order, not spelled backwards). For now restrict yourself to a fixed sentence length, when you know more you can work with sentences of arbitrary length.

4. Implement a very simple spell-checking program which takes a single line of text, splits it into words, and checks each of these against a fixed list of (say four) words. If the word matches and of the four, then it is correct, otherwise it is wrong.

6.9 Summary.

This has been a long section which has covered a large amount of ground. This is to be expected—at the moment we are trying to simultaneously learn the basic principles of programming, to learn particular details of the programming language and to learn how these details fit together in order to construct programs which work. This means that a lot of things need to be introduced "at once". This is similar to learning to drive—it is impossible to learn to steer, then to learn to use the gears, et cetera—you have to do the whole thing at once, concentrating from time to time on various things. I think at this stage the easiest thing is to read through each section carefully, try out the exercises. However don't feel the need to master each topic before moving onto the next. Frequently seeing the same ideas in several different contexts can make understanding a lot easier.

So, to summarize. We can output strings of text using the `System.out.println` statement. This can either output strings directly, by enclosing them in double-inverted-commas, or it can print out the content of an object of type `String`. We have also looked at two forms of *input*. The first is command-line input, which uses the arguments to the `main` method. The second is to set up a `Reader` object, called a `BufferedReader`. Once we have our strings we can manipulate them using the `StringTokenizer` class.

6.10 Pencil-and-paper/computer exercise.

Find the errors in the following program. It should take a line of four words of text, and display the words one-by-one on separate lines. Correct the errors, get the program to run, and add appropriate comments to the program.

```
import java.io;
import java.util.StringTokenizer;

class InputOutput
{
  public static main()
       throws java.io.IOException
  {
       Sting line
       BufferedReader keyboard
          = new BufferedReader(new
            InputStreamReader(System.in));
       StringTokenizer SplitString();

       System.Out.PrintLn("Please type in a line"+
                          " of text four words long)
       line = keyboard.ReadLine();
       splitString = new StringTokenizer(lion);

       System.out.print("splitString.nextToken()");
       System.out.print("splitString.nextToken()");
       System.out.print("splitString.nextToken()");
    }
```

7 The design process.

Solutions to problems
are easy to find:
the problem's a great
contribution.
What's truly an art
is to wring from your mind
a problem to fit
the solution.

—*Piet Hein*

In this section we shall describe a simple program, go through a basic design process, and produce a program which solves the problem. The details of the program content will be explained later, but I think it is important to see something like this at the beginning. This is akin to learning a natural language, where we initially hear a phrase sounding like *kelluretill?*, and associate it with a particular concept, and only later break it down into the individual words *"Quelle heure est il?"* which we can then use separately to form new sentences. This section leads us through the programming process.

Let us begin with a real-world problem, this time the world in question is that of arithmetic, and the problem is to print out the prime numbers[9] between 2 and 100. We can do this trivially by calculating them by hand and writing a program to print them out as above, but the point is to use the computer to *compute* this.

So the first stage is the *design* stage. We may know already, or find out from a book, or work out by thinking about it [10] that a number between 2 and 100 is prime if it *cannot* be divided by 2,3,5 or 7. So our design looks like this :

```
Take the numbers in turn from 1 to 100.
If the number is divisible by 2, 3, 5 or 7
    or if the number is 2, 3, 5 or 7
    go onto the next number
else
    print the current number
    and then go onto the next number.
```

This kind of design is called *pseudocode*, as it looks somewhat similar to the "code" (lines of computer language) used when writing a computer program.

The second stage is to formalize this using the *Java* language. I have **emboldened** in the pseudocode above all of the important things which translate fairly directly into *Java* statements. Here is the translation into *Java*. You are not expected to understand this straight away!

```
import java.io.*;

class FindPrimesApp
{
```

[9]The prime numbers are the numbers with only two divisors, i.e. $2, 3, 5, 7, 11, \ldots$

[10]Proof sketch : if a number can be divided by a non-prime number (like 14 divides 42), then the non-prime number can be divided by prime numbers (in this case 7 and 2). So it suffices to check for divisibility by primes. However for numbers in the range 2–100 it is necessary to check only those primes less than $10 (= \sqrt{100})$, as if it can be divided by a number greater than 10 then the other divisor must be less than 10, otherwise the number would be greater than 100. ■

```
  public static void main(String args[])
  {
    int number;
    int total = 0;
    for ( number=2 ; number < 101 ; number = number + 1)
      {
         if (   (     ( (number%2)==0 )
                   || ( (number%3)==0 )
                   || ( (number%5)==0 )
                   || ( (number%7)==0 )
                )
             &&
                (
                      (number != 2)
                   && (number != 3)
                   && (number != 5)
                   && (number != 7)
                )
            )
          {
             /* number isn't prime */
          }
        else
          {
             /* number is prime */
             System.out.println(number+" is prime");
          }
      }
  }
}
```

Try to work out what aspects of the program correspond to the various steps in the English language description above. To help : `&&` means "and", `||` means "or" and `%` means "remainder". What lines correspond fairly accurately between the two? Which lines in the computer language have no analogy in the English language description?

7.1 Computer exercises.

1. Type the above program into the computer, compile and run it. If it doesn't work, then compare carefully what you have typed with what is printed in the notes.
2. If (and only if!) you are feeling confident (i.e. if you already know some programming, or are coming back to this exercise later on this the term), try modifying the program to
   (a) Print out the *non*-prime numbers between 2 and 100.
   (b) Print out the prime numbers between 2 and 144.

7.2 Pencil-and-paper exercises.

Write out (in English) a step-by-step procedure for the following things

1. Deciding whether one word is contained within another.
2. Deciding whether two words are anagrams of each other.
3. Sorting a list of numbers into numerical order.
4. Sorting a list of words into alphabetical order.
5. Checking whether you have won a prize on the lottery.
6. Writing out all of the combinations of a set of letters.
7. Writing a decimal number in binary.
8. Finding the prime factors of a number.

Sometimes there will be more than one way of doing these things. If there is, which way would be better to use on the computer, and why? Later on, once you have learned some more of the *Java* language, you can come back and translate these procedures into the *Java* language.

8 Variables.

We have talked above about *variables*, that is words like `name` or `number` which stand for particular things in the computer's model of the world. They are *variables* because then can change their value as the program progresses, as opposed to *constants*, like `2` and `"This string is fixed"` which cannot change in that way.

All variables in *Java* have a *name*, which must be unique within the program[11]. They also must have a *type*, which says what kind of information is represented by that variable. These type can be built-in to the main language, like `int` (integers) or `String`, or they can be of types defined by the user.

The basic syntax for creating an object of a particular type consists of the name of the type followed by the name of the variable object you would like to create. You can create more than one variable of the same type by separating the name by commas. Examples

```
  int firstNumber, secondNumber;
  String myName;
```

These two lines respectively create labels for integer variables called `firstNumber` and `secondNumber`, and then a string variable called `myName`. However at this stage these "containers" are still empty.

To actually give a value to them we use an assignment statement, which consists of the variable name on the left hand side (l.h.s.), followed by an equals (=) sign, and a value on the right hand side (r.h.s.). Examples.

```
firstNumber = 4;
secondNumber = 5;
myName = "Zaphod Beeblebrox";
```

Note that we cannot assign a number to the string variable :

```
myName = 6; //this line is wrong
```

is wrong, and will hopefully produce an error message at the compilation stage.

We can also assign a variable of one type to a variable of another type. This *copies* the current value from the r.h.s. variable into the l.h.s. variable, the previous value of the l.h.s. variable is lost, and the value of the r.h.s variable remains as before.

```
  firstNumber = secondNumber;
```

At the end of this, `firstNumber` and `secondNumber` will both equal 5.

Here is a simple program which demonstrates some of these ideas.

```
import java.io.*;

class VariablesApp
{
   public static void main(String[] args)
```

[11]This is actually more complicated. The same name can be used in different classes, or locally within different methods within a class. However at this stage it is wisest to use distinct variable names for everything in your program.

```
      {
         int firstNumber,secondNumber,thirdNumber;
         String myName;
         firstNumber = 4;
         secondNumber = 5;
         myName = "Zaphod Beeblebrox";
         System.out.println(firstNumber+" and "+secondNumber);
         System.out.println("My name is "+myName};
         thirdNumber = firstNumber;
         System.out.println(thirdNumber);
         thirdNumber = secondNumber+3;
         System.out.println(thirdNumber);
         myName = myName + ", but not really");
         System.out.println("My name is "+myName};
      }
}
```

8.1 Variable-naming rules.

So far we have constructed variable names by stringing a few words together. This is the most common kind of variable name. There is a set of rules on what can be used as a variable name :

- It must be a list of one or more characters chosen from the uppercase letters, the lowercase letters, the digits, the `$` symbol and the _ symbol.
- The first character must not be a digit.
- There must not be any "white space" in the name.
- The name should not be the same as a *Java* keyword. These are : `abstract, boolean, break, byte, case, cast, catch, char, class, const, continue, default, do, double, else, extends, false, final, finally, float, for future, generic, goto, if, implements, import, inner, instanceof, int, interface, long, native, new, null, operator, outer, package, private, protected, public, rest, return, short, static, sure, switch, synchronized, this, throw, throws, transient, true, try, var, void, volatile, while.`
- The name must not be the same as another variable in the same program[12].

[12] Strictly within the same *scope*—but you can worry about this later.

Note that uppercase and lowercase letters are treated differently. So `cheesesandwich` and `cheeseSandwich` are different variables. Actually using two names which differ apart from capitalization is a route to terminal confusion, and should be avoided.

8.2 Pencil-and-paper exercise.

Which of the following are valid variable names in *Java*?

```
3VOL
SLARTIBARTFAST
h
h3
numberOfCows
colin'sBankAccount
_moose
Carrot$And$Stick
H
Value4
My_little_pony
new_value
cout
new
$aardvark
```

8.3 Variable-naming conventions.

There are a number of conventions for the use of these names. Ordinary variables are conventionally notated as a list of words, the first all in lower case and the remainder capitalized.

```
int luckyNumber;
String nameOfTheLuckyPerson;
```

Variables which you do not intend to change during the course of the program are given names in uppercase, with the different words separated by underscores.

```
double PI = 3.141593;
String AUTHOR_OF_THIS_BOOK = "Colin Johnson";
```

Class names (see 9) are traditionally a list of words, all of which begin with a capital letter.

```
class Dog
{
  . . .
};
class LibraryBook
{
  . . .
}
```

Note that the names of primitive data types defined in the *Java* language, like `int` and `float`, and statement names like `if` and `for`, are always in all-lowercase.

8.4 Pencil-and-paper exercises.

Which of the following declarations are correct, in the sense that they would be correctly compiled? Which obey the naming conventions outlined in the previous section? Correct the errors.

```
int aNumber;
String MyString;
boolean true;
boolean TRUE_FALSE;
int FEIGENBAUMS_NUMBER = 4.669;
double accountBalance;
char singleLetter = "a";
int balesOfHay;
double accountBalance = 0.0;
int anotherNumber
boolean coinFLIP;
```

8.5 Types.

As we said above, every variable is of a *type*, which describes the kind of data that can be held in that variable. When you get down to writing substantial programming, many of your variables will be of complicated types, such as a variable describing a student's academic record. However all of these are built up from certain primitive data types which encapsulate the concepts of numbers, text, and true/false decisions.

Here are the most useful primitive data types.

int The `int` type holds one integer. It is restricted to numbers between -2147483648 and +2147483647[13].

[13]Question : why these seemingly arbitrary numbers?

double The `double` type holds one decimal number, with up to fifteen significant figures[14]

String Which holds a string of text. To represent strings absolutely, we enclosed them between double quotation marks : "this is a string". Strictly `String` is not a primitive data type, it is an object which is automaticlaly defined in all *Java* applications, and it has a couple of properties (such as the definition of the + operator for string concatenation) which make it unusual. However this subtlety can be put to one side for now, but you will need to remember it for later.

char A variable of `char` type holds one character. To represent a character absolutely we enclose it in single (left) quotation marks : so 'a', 'b', '4', '%' et cetera.

```
char singleLetter;
singleLetter = 'a';
singleLetter = 'd';
```

boolean A `boolean` variable is either `true` or `false`.

```
boolean yesOrNo;
yesOrNo = true;
yesOrNo = false;
```

8.6 Typecasting.

Sometimes we want to convert variables to other types. In some circumstances this happens automatically, for example if we multiply an `int` varibale by a `double` variable the answer is a `double` variable, the `int` being temporarily converted into a `double` whilst the calculation is taking place. The `*` operator is used to indicate multiplication, as the set of symbols historically used in computer languages do not contain a $\times$ symbol.

```
int n;
double x,y;
n = 12;
x = 15.2;
y = x*n;
```

[14]So why not *decimal* or *real* or something more intuitive? This is a historical legacy from C and C++, where the most common type of decimal representation was a `float` of 7 significant figures, and the `double` type allowed double the number of significant figures. Now that computers are faster and have more memory, the `double` is standard, but the name sticks.

Sometimes this is more complicated. Assume that we want to do the calculation the other way—we want to convert the `double` to an `int` in order to produce an `int` result (clearly there would be some rounding needed). To do this we use a *typecast*, consisting of the name of the desired type in brackets before the variable name.

```
double x;
int n,m;
n = 13;
x = 16.4;
m = n * (int)x;
```

This last line reads something like "m equals n times the integer form of x". The conversion from double to integer works by rounding down to the nearest integer below the float, as illustrated in the following program.

```
import java.io.*;

class CastApp
{
   public static void main(String[] args)
      {
        double x,y;
        int n,m;
        x = 16.4;
        y = 16.9;
        m = (int)x;
        n = (int)y;
        System.out.println(m+"  "+n);
      }
}
```

Another kind of conversion that we can do is between `String` or `char` representations and numerical representations. The simplest of these is to cast a `char` into an `int`. This gives the number the computer uses internall to represent that character—its ASCII code. Here is a program which illustrates this.

```
import java.io.*;

class CastTwoApp
{
   public static void main(String[] args)
      {
        int n,m;
        char letterOne,letterTwo;
        letterOne = 'A';
        letterTwo = 'C';
        m = (int)letterOne;
        n = (int)letterTwo;
        System.out.println(m+"  "+n);
    }
}
```

Another related thing that we want to be able to do is to input integers into programs. Imagine writing a calculator program, or any program which accepts numerical data as an input. Our input routines from earlier read in text which it stored in objects of type `String`. How do we convert these `String`s to numbers with which we can do arithmetic? Fortunately there is a built in routine to do this for us, called `ParseInt`. This is a method of the `Integer` class, which is another way of representing integers different from the `int` built in type[15]. The basic syntax is like this.

```
String str1;
int n,m;
str1 = "46";
n = Integer.parseInt(str);
m = 54;
System.out.println(n+"  "+m+"  "+(n+m));
```

Embed all that in a program and get it to run. Now use the `readLine` method and a string buffer to input integers from the keyboard and do arithmetic with them.

8.7 Computer exercises.

1. Write a program based on `CastTwoApp` above which converts letter grades (say on a scale of A-G) into number grades.
2. Write a simple addition program, which accepts two numbers as input and returns their sum.

[15]I am glossing over a lot here. Strictly `int` isn't an class in the *Java* language, but `Integer` is. So if we want to act on integers using more than the basic arithmetic operators we need to do an (explicit or implicit) convertion from `int` to `Integer` and back again. This is perhaps one of the few candidates for a flaw in the design of the *Java* programming language, though I understand why it is like it is. Read more about this in one of the textbooks sometime, as it is a difficult feature of the *Java* language to understand and it is worth learning about.

9 Classes and objects.

We have already touched upon the concepts of classes and objects above. In this section we shall go into detail about these concepts, both from a theoretical and a practical point of view.

9.1 Classes.

"The history of all hitherto existing society is the history of class struggles."

— Marx and Engels, The Communist Manifesto

A *class* is an abstract *category* or *type* of things in the real world. So for example "book", "student", "vehicle", "bus", "car", "window", "square", "circle", "shape" are all examples of classes. However a particular book, student, window or whatever are *not* classes.

A class has certain *properties*, which vary from object to object within that class. So for example the `student` class will have properties like "name", "address", "exam results" et cetera. Remember what was said earlier about *showing deliberate ignorance* of irrelevant features of a model—different applications will represent the same thing in different ways. For example a "student" class used to create a database for the Student Health Centre will store details of the student's height, whilst a database of students stored for academic purposes will not include that information.

Here is a fragment of *Java* code which describes a simple bank account.

```
class BankAccount
{
   String customerName;
   String address;
   int telephone;
   boolean student;
   double balance;
}
```

This is just a fragment of code—don't try to run it yet. To remind you, class definitions tend to be placed at the top of a program file, with the application class (like `HelloWorldApp`) at the bottom of the file.

The first line uses the keyword `class` to say that what follows is a class definition, and immediately following that is a name for the class, `BankAccount`[16] The remaining part, that between the curly brackets, says what kinds of data are stored *when* we create an object of type `BankAccount.` At this stage *no* memory is allocated to store names and addresses et cetera, all that has been created is the concept of a bank account. These are in the form of variable declarations as described in the previous section.

[16]The naming rules for classes are the same as those for variables. However it is conventional to begin class names with a capital letter.

9.2 Objects.

"Prais'd be the fathomless universe, for life and joy, for objects and knowledge curious."

— *Walt Whitman*

So how do we create an object—a particular *instance*[17] of a class? The syntax is similar to creating a variable of predefined type like `int`. We type the name of the class followed by the name of the object that we wish to create. So to create a bank account object called `colinsBankAccount`[18] we type

```
BankAccount colinsBankAccount = new BankAccount();
```

which then allocates memory sufficient for one name, one address, et cetera.

Now we have our object what can we do with it? To access the information in the object we use a dot `.` symbol. This works as follows. If we now want to say that my bank balance is £851.62, we issue the following statement.

```
colinsBankAccount.balance = 851.62;
```

using the assignment operator = introduced above. Now if we want to print this balance we can use the following statement.

```
System.out.println("Colin's bank balance is "
                    +colinsBankAccount.balance);
```

So in general if *obj* is an object and *data* is the name of one ofthe pieces of data in that object, we refer to that piece of data outside the class defintion as *obj.data*.

Let us bundle all this together into a program that you can type in and run. There are two files here. The first is called `BankAccount.java` and defines the account class itself.

```
import java.io.*;

class BankAccount
{
   String customerName = "Blank";
   String address = "Blank";
   int telephone = 0;
   boolean student = false;
   double balance = 0.0;
}
```

[17]The process of creating an instant of a class is called *instantiation*.

[18]to fit within the variable naming rules and conventions we have to miss out the possessive apostrophe and decapitalize the name!

The second is called `FirstBankApp.java` and contains the main application.

```
class FirstBankApp
{
   public static void main(String[] args)
      {
         System.out.println("Bank account program.");
         BankAccount colinsBankAccount = new BankAccount();
         colinsBankAccount.balance = 851.62;
         System.out.println("Colin's bank balance is "
                           +colinsBankAccount.balance);
      }
}
```

An important thing to note here is the syntax for creating an object. Look at the line

```
         BankAccount colinsBankAccount = new BankAccount();
```

In this the first two words are the *declaration*, they ask for a label called `colinsBankAccount` to be created, and for it to be ready to point to an object of class `BankAccount`. The stuff after the equals sign actually creates the area in the computer's memory which is going to store the details of the bank account itself. The keyword `new` essentially means "create space in the memory for an object of that type", and the thing following it is what we will later call a *constructor* method.

9.3 Computer Exercises.

1. Type in the above program at get it to run.
2. Modify the program so that you assign something to all the components of the class.
3. Modify the program so that the user can *type* in the various components of the class.

9.4 Another example.

Here is another example—a short program which creates a new class. The new class is call LotteryResult, and represents the balls drawn in the national lottery. The main program accepts input of a particular set of results, and prints them out again. This is all we can do for now—at the moment no *computing* can go on within the class until we have introduced the concept of a *method* below. We shall use the idea of an *array* here—that is a list of things of the same type, indexed by a number. So instead of having to create six values called `ballOne`, `ballTwo`, ..., `ballSix`, we say

```
  int[] balls = new int[6];
```

which creates a list of objects `balls[0]`, ..., `balls[5]` to represent the balls. More details in section 13 below.

From now on I shall leave you to work out how to divide up the code into files. Remember that each *class* has its own file, named after the class.

```
import java.io.*;

class LotteryResult
{
  int[] balls = new int[6];
  int bonusBall;

  public String toString()
  {
    return "The balls are "+balls[0]+", "+
      balls[1]+", "+balls[2]+", "+balls[3]+
      ", "+balls[4]+" and "+balls[5]+
      " and the bonus ball is "+bonusBall;
  }
}

class LotteryOneApp
{
  public static void main(String[] args)
       throws java.io.IOException
  {
    System.out.println("Welcome to the national lottery.");

    /* set up the variables and keyboard reader */

    LotteryResult thisWeek = new LotteryResult();

    BufferedReader keyboard
      = new BufferedReader(new InputStreamReader(System.in));

    String number;

    /* input this weeks numbers */

    System.out.println("Please type the numbers one by one,"
       +" pressing the return key each time.");
    int i,n;
    for (i=0;i<=5;i++)
      {
        number = keyboard.readLine();
        n = Integer.parseInt(number);
        System.out.println("Number is "+number);
        thisWeek.balls[i] = n;
      }
    System.out.println("Please type the bonus number,"
       +" then press the return key.");
    number = keyboard.readLine();
    n = Integer.parseInt(number);
    System.out.println("Bonus number is "+n);
    thisWeek.bonusBall = n;

    /* print out the numbers */

    System.out.println("\n *** This week’s results ***");
    System.out.println(thisWeek.toString());
  }
}
```

Look carefully through the program, and try to work out what is happening. There are one or two unfamiliar constructs–can you work out what they do either by observing their behaviour in the program, or by loooking them up in a textbook?

Actually the direct mention of the `toString()` method in the final non-trivial line is not really needed. It can instead be written in this way,

```
    System.out.println(thisWeek);
```

as the `println` method automatically searches for a `toString` method. This also applies with primitive data types like `int`s and `double`s.

9.5 Computer exercises.

1. Modify the lottery program using a string buffer so that you can type all of the numbers on one line.
2. Create a simple program which creates a class for library books, and allows you to input and change data like the lottery program above.

9.6 Pencil-and-paper exercise.

1. (THIS QUESTION IS TAKEN FROM LAST YEAR'S COURSEWORK). We are creating a computer system for the university library.
    (a) Identify four classes of objects which might be used in creating the system.
    (b) Take one of these classes and detail the information which would need to be stored about that object in the system.
    (c) A user comes in to borrow a book. Which information would be changed by this?
    (d) Name two other examples of tasks which would need to be processed by the system.

10 Control-flow abstractions.

So far we have dealt with programs where the flow of control has gone from the first statement in a method to the end of the method. However, this limits us to a very small set of programs—those where a single sequence of actions is sufficient to solve a problem. Frequently we need a program which can make a choice depending upon the users input, or which can carry out one of a set of actions depending on previous calculations, or which repeats the same computation or similar computations many times.

To handle these things in a sensible way there are a number of *control-flow abstractions* which represent in a flexible and abstract way these different ideas. Here are the common ideas. There are other interesting things like `do` statements and enumeration methods and threads which will be described in any comprehensive *Java* book.

10.1 `if`.

"'If's an illusion, I don't believe in if any more."

— *Roger Whittaker*

Probably the simplest control-flow statement is the `if` statement, which tests for some condition and then carries out a specified block of statements if the condition is satisfied, and carries on if the condition is not satisfied. Here is an example.

```
import java.io.*;
```

```
class IfExampleApp
{
   public static void main(String[] args)
      throws java.io.IOException
     {
       BufferedReader keyboard
         = new BufferedReader(new
           InputStreamReader(System.in));
       String line;

       System.out.println("Please type the password, "
                          +"then press the return key.");
       line = keyboard.readLine();

       if (line.equals("elephant"))
         {
            System.out.println("Password correct");
         }
       System.out.println("Thank you for using this program.");
     }
}
```

The syntax in general looks like this : there is the keyword `if` followed by the conditional expression in parentheses. Following this there is one or more statements enclosed in curly brackets. If the conditional expression is true, then the statements between the curly brackets are carried out, and control then transferred to the next statement after the brackets. If the conditional is false, then control passes directly to the first statement after the end of the curly brackets.

Sometimes we would like to choose between two different alternatives. In this case we can use a structure of the form `if ...else ...`, where the block between `if` and `else` is what happens if the condition is satisfied, and the bit after the else what happens if the condition is not satisfied. Here is a variant of the above program which demonstrates this.

```
import java.io.*;

class IfElseExampleApp
{
   public static void main(String[] args)
      throws java.io.IOException
     {
       BufferedReader keyboard
          = new BufferedReader(new
            InputStreamReader(System.in));
       String line;

       System.out.println("Please type the password, "
                          +"then press the return key.");
       line = keyboard.readLine();

       if (line.equals("moose"))
         {
            System.out.println("Password correct");
         }
       else
         {
            System.out.println("Password incorrect");
         }
       System.out.println("Thank you for using this program.");
     }
}
```

Another interesting use of the `if` statement is in checking for errors in input. Here is an example.

```
import java.io.*;
import java.util.StringTokenizer;

class CountWordsApp
{
  public static void main(String args[])
       throws java.io.IOException
  {
       BufferedReader keyboard
          = new BufferedReader(new InputStreamReader(System.in));
       StringTokenizer splitString;
       String line;

       System.out.println("Please type three words, "+
          "then press the return key.");
       line = keyboard.readLine();
       splitString = new StringTokenizer(line);

       if (splitString.countTokens() != 3)
         {
   System.out.println("Sorry, wrong number of words.");
         }
       else
         {
                System.out.println("Correct number of words.");
            }
        System.out.println("Thank you for using this program.");
    }
}
```

10.2 Computer exercise.

1. Modify the password-checking program above so that it gives the user three chances at getting the password correct.

10.3 `for`.

"'The first ten million years were the worst,' said Marvin, 'and the second ten million, they were the worst too. The third ten million I didn't enjoy at all. After that I went into a bit of a decline.'"

— *Douglas Adams*

Another key control-flow concept is *repetition*. One of the most powerful things we can do with computers is repetitive calculation—we can add up a huge list of numbers, we can simulate hundreds of different variants on weather patterns or financial predictions, we can look through thousands of files searching for a particular key phrase. The main statement for carrying out repetition in *Java* is called `for`.

Here is an example. Type this program in and get it to run.

```
import java.io.*;

class ForExampleApp
{
  public static void main(String[] args)
  {
    int i;
    for (i=0;i<20;i++)
      {
         System.out.println(i);
      }
  }
}
```

What is going on here? The `for` statement takes in three arguments, which control the number of times the block following the statement is repeated. The first argument (`i=0`) is carried out before the first repetition. This is some kind of initialisation statement. In this case it sets a counter, called `i`, to zero. The second statement (`i<20`) is a *finishing condition*. Before the computer carries out a repetition of the loop it first checks the finishing condition, and if the condition is false (in this case, when `i` becomes greater than or equal to twenty), the computer stops repeating and transfers control directly to the end of the block. The final argument (`i++`) is executed each time a repetition of the loop has been completed. In this example it increments the counter `i` by one.

Note that you can access the variable `i` from within the block. We use that to display the number. We can also change the value of `i` if we want. What happens if you insert the line

```
i = i+2;
```

immediately after the line `System.out.println(i);`? Try to work this out theoretically first, then type it in to try it out. However it is usually possible to incorporate thing slike this into the main `for` statement, and this should be does if possible as it makes for cleeaner code.

10.4 Computer exercises.

1. Modify the above program so that it counts down from 24 to 0.
2. Modify the above program so that it displays the odd numbers between 20 and 120.
3. Experiment with missing out arguments from the `for` statement.
4. You may be familiar with the classroom game of *fizz-buzz*, where the class sits in a circle and counts numbers in turn, replacing the number with "fizz" if it is divisible by three, "buzz" if it is divisible by five, and "fizz-buzz" if it is divisible by both. Modify the program so that it displays these strings whenever such numbers come up.
5. Modify the fizz-buzz program so that the user can choose to replace 3 and 5 with two numbers of their choice.
6. (THIS IS TAKEN FROM LAST YEAR'S CLASS TEST.) A `for` statement has three parameters in the brackets which follow the keyword `for`.
   (a) Describe what each of these parameters does.
   (b) What is the effect of leaving out the first parameter?
   (c) The second?
   (d) The third?
   (e) What happens if all of them are left blank?
   (f) Explain what is meant by an *infinite loop*, and write a few lines of *Java* giving an example.

10.5 `while`.

The `while` statement is another looping construct. It has a simpler syntax `while (` *condition* `)` { *actions* } . The condition is a single argument, like `line != "elephant"`. Here is another password-checking program which uses this structure.

```
import java.io.*;

class WhileExampleApp
{
  public static void main(String[] args)
      throws java.io.IOException
  {
    BufferedReader keyboard
      = new BufferedReader(new InputStreamReader(System.in));
    String line;

    System.out.println("Please type the password,"+
                       " then press the return key.");
    line = keyboard.readLine();

    while (!line.equals("aardvark"))
    {
      System.out.println("Try again.");
      line = keyboard.readLine();
    }
    System.out.println("Correct!");
  }
}
```

10.6 Computer exercises.

1. Reimplement the word-counter program above using a while loop, so that it gives you as many chances as you need to get the right number of words. Now modify the program so that the user can choose how many words to input.
2. Imagine a game like squash or badminton, where two players score points (though not in strict alternation). Write a program which allows either player to type in when points are scored, keeps a track of the total, and stops when either player has reached a winning total. Do the same for darts scoring, where you have to count downwards, finish exactly, and finish on a double (and get the computer to tell you which double you need, et cetera).

10.7 `switch`

"The world is everything that is the case."

— *Ludwig Wittgenstein.*

Variations on the concept of the *menu* are ubiquitous in computer applications. Another common thing is a long sequence of `if` constructs. and many other situations, are appropriately dealt with by the `switch` statement in *Java*.

The `switch` statement consists of the keyword `switch` followed by a variable name in parentheses. The body of the statement is contained in curly brackets after this, consisting of several statements of the form `case` : *a number* : ...`break;`. If the variable is equal to the number given, then everything between that `case` statement and the `break` statement is carried out. If none of the variables are matched, then a `default` statement will executed if there is one. Following this, control is transferred to the statement after the end of the curly brackets defining the body of the `switch` statement. Here is an example.

```
import java.io.*;

class SwitchExampleApp
{
   public static void main(String[] args)
       throws java.io.IOException
      {
        System.out.println("Welcome to Milliways, "+
            "the restaurant at the end of the universe.");

        /* set up the computer for keyboard input */

        BufferedReader keyboard
           = new BufferedReader(new InputStreamReader(System.in));
        String choice;
        int n;

        /* print out the menu */

        System.out.println("\n\t****** Menu ******");
        System.out.println("\n1. Pan-galactic gargle-blaster.");
        System.out.println("2. Carrot burgers.");
        System.out.println("3. Pie and chips.");
        System.out.println();

        /* input the user’s choice */

        choice = keyboard.readLine();
        n = Integer.parseInt(choice);

        System.out.println();

        /* print out appropriate response */

        switch (n)
        {
          case 1:
            System.out.println("Pan-galactic gargle-blaster, "+
                               "at once!");
            break;
          case 2:
            System.out.println("Carrot burgers. "+
                               "Certainly, my friend!");
            break;
          case 3:
            System.out.println("Pie and chips is off, "+
                               "I'm afraid.");
            break;
          default:
            System.out.println("There isn't a choice "+n+
                               " on the menu.");
            break;
        }
      System.out.println("Thank you for choosing "+
                         to eat at Milliways.");
    }
}
```

Type it in, get it to run, and carefully work out what is going on. Try adding other items to the menu. Change it so that the menu is alphabetical. Add sub-choices like different sauces that you can have. Attempt to simulate the near-infinity of choices-within-choices on the menu at the *Imperial* if you like!

A caveat : only `char` and `int` variables can be used as the parameter in the switch statement[19]. This is somewhat regrettable, and stems from there being no standard way of defining "equality" in a *Java* class.

[19]plus one or two others we haven't talked about yet, but they are essentially variants on `int`.

10.8 Computer exercises.

1. Write a program which allows the user to type in any letter, and which replies with a word beginning with that letter ("A is for apple, B is for Belgium," et cetera).

2. Write a program which creates an instance of the BankAccount class from above, and which then presents the user with a number of possible actions : paying money in, taking money out, inputting a name, changing address, displaying the record, and any other thing that you can think of.

3. Do a similar exercise but this time for a student's academic record. You will have to write the class definition yourself this time. Make the menu choice alphabetic. Make some of the menus "two level"—i.e. there are menus within menus.

10.9 Pencil-and-paper exercises.

1. (THIS IS TAKEN FROM LAST YEAR'S CLASS TEST.) Write a fragment of *Java* which inputs a character and outputs the word "alphabet" if the character is a lower-case letter and "number" if it is a number. [OPTIONAL HINT : You may assume that the computer is using ASCII encoding .]

2. I have subtly slipped in codes like `\t` and `\n` into some of the strings above. What do they do? What does the statement

   ```
   System.out.println();
   ```

   do?

10.10 Computer exercises.

These exercises draw upon and consolidate all of the ideas in this section.

1. (THIS WAS PART OF LAST YEAR'S COURSEWORK.) Write a program which accepts as input three characters, two of which are single figure numbers and one of which is a letter chosen from the set $\{a, s, m, d\}$. According to the letter inputted the two numbers are added, subtracted, multiplied or divided. Extend the program so that there is error checking on the input. Extend the program so that several calculations can be done, the program stopping when the letter e is input. Incorporate a check for dividing by zero.
2. Write a program which accepts two strings (of fixed length at first, then of arbitrary length) and determines whether they are anagrams of each other. There are at least three ways to solve this problem.
3. Take some of the programs above and rewrite all of the `while` statements as `for` statements, and vice versa. Similarly write all of the `switch` statements using `if` statements. Can you always do this? If so, why are the different statements provided?
4. Write a simple "hangman" game. At the beginning, a word is chosen (fix the length of the word in advance), either by the computer choosing from a preprepared list or by a player typing in a word. Then loop through, printing the current guessed letters and the letters that have already been chosen. Continue until either a certain number of letters have been tried or the word has been guessed.

11 Methods.

"Though there be madness, yet there be method in it"

— William Shakespeare

In this section we introduce the concept of a *method*. Methods define how the information in a class and be accessed or processed.

11.1 Access to object elements.

In the classes declared in section 9 the variables were declared as `public`. Thus elements of objects of that class can be accessed from other objects (including the application class) directly. Sometimes this is not what we want, for a number of reasons :

- Earlier we talked about *modelling the world* using objects. If an object is to represent a real-world thing, then we would not only like to represent the *attributes* of that thing but also to represent the possible *actions* and *behaviours* of that thing.

- When many people are working on a project, we would like to create classes whose *external interface* (that is, external to other parts of the program) is consistent, allowing the writer of that individual class to change the internal details whilst leaving the external interface the same. Thus when one worker changes the internal details of how the class is implemented, the other workers can keep on using the class as before.

- We want to write classes that are *reusable*. One conceptual problem wih creating reusable software is that people have often found it easier to write their own software from scratch than try to understand the fine details of someone else's software. If we can encapsulate the behaviours of the class as a set of interfaces, then all the reuser of the software has to do is to understand these external interfaces (look at the descriptions in the *Java API Index* web site that I told you to bookmark earlier for examples). This is probably the most important productivity benefit from object-oriented design, and after years of rather vacuous talk about this, it is now beginning to come true.

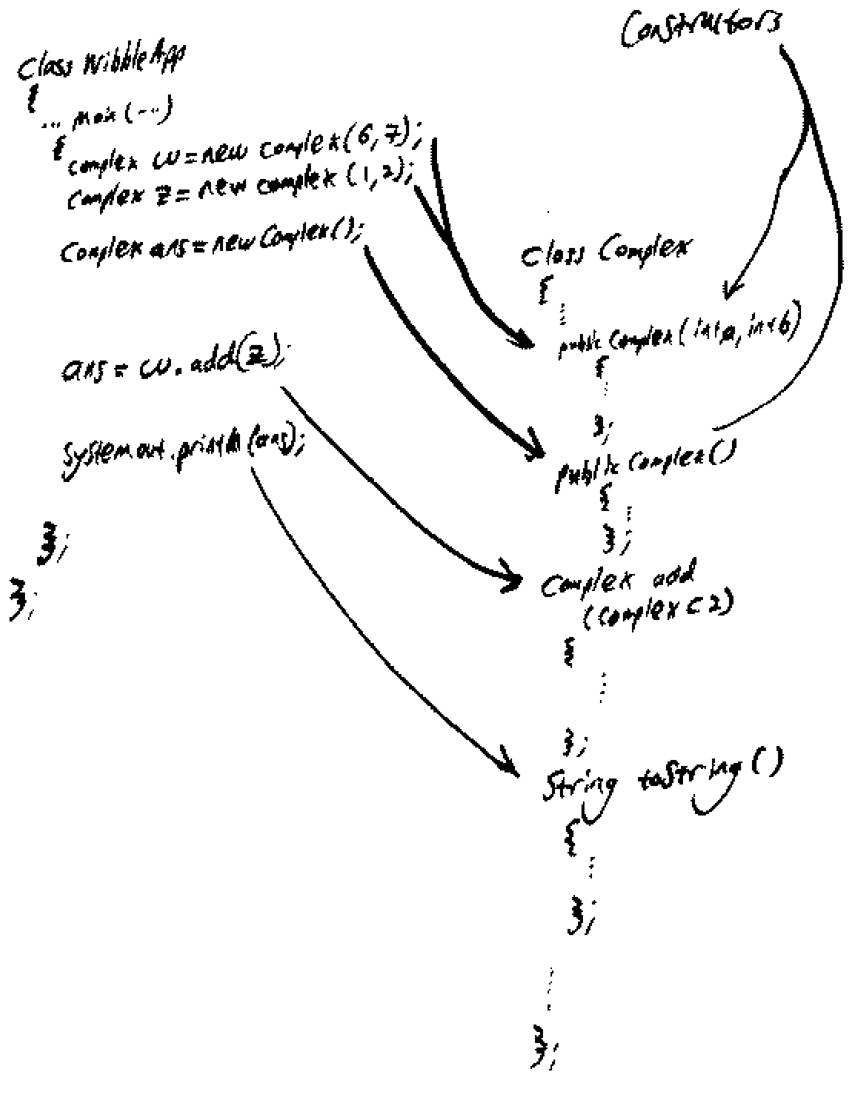
The key idea here is that we are not just encapsulating *data* when we define a class, but we are defining *interfaces / actions / behaviours*. We define these through the use of *methods*, which are defined within a class and which describe these properties.

The diagram in figure 3 summarizes some of the ideas here in pictorial form.

11.2 Examples.

Consider a class used to contain the details of how to access sites on the Internet. If you are writing a program that uses this class, you would like to deal with that class at a particular level of behaviour—you want to be able to say what site to visit, what information to bring back. You don't want to have to specify the fine details of the communication protocols, you would like that to be *encapsulated* within the class definition.

Another example is the various routines in the `java.io` library. You want to print out a string of text, or read in a number, or download some data to a file. The classes within `java.io` encapsulate these concepts (like `println`) so that you don't have to worry about the details of how characters are drawn on the screen, when to wrap sentences, when to scroll the screen, et cetera.



```
class WibbleApp
{
 ... main (...)
 {
  complex w=new complex(6,7);
  complex z=new complex(1,2);
  Complex ans=new Complex();

  ans=w.add(z);
  System.out.println(ans);
 };
};
```

```
class Complex
{
 :
 public Complex(int a, int b)
 {
  :
 };
 public Complex()
 {
  :
 };
 Complex add
   (complex c2)
 {
  :
 };
 String toString()
 {
  :
 };
 :
};
```

Figure 3: Methods.

11.3 On to practicalities.

This section so far has been rather dry and theoretical. That is to be expected—we have just explained several of the key ideas underlying the object-oriented programming philosophy. Now for a practical example.

We have already seen one method in disguise—the `main` method which is part of the application class. Let us create another example. First, we return to the `BankAccount` class defined earlier. Here is the class definition.

```
class BankAccount
{
   String customerName = "Blank";
   String address = "Blank";
   int telephone = 0;
   boolean student = false;
   double balance = 0.0;
}
```

Now we will add to this class a number of methods. Let us begin by adding two very simple methods, which will allow us to set the `boolean` variable `student`, which contains `true` if the customer is a student and `false` otherwise.

```
class BankAccount
{
   String customerName = "Blank";
   String address = "Blank";
   int telephone = 0;
   boolean student = false;
   double balance = 0.0;

   public void setAStudent()
   {
      student = true;
   }

   public void setNotAStudent()
   {
      student = false;
   }
}
```

Notice that we are still within the class defintion—we have not done anything to any particular objects of class `BankAccount` by adding these methods.

So what we have done here is to define two kinds of message that we can pass to an object of class `BankAccount`. The first passes the information that the account belongs to a student, the second that the account does *not* belong to a student. The general form of a method is therefore (for now) the words `public` and `void` followed by the name of the method, followed by a pair of empty parentheses, and followed by the content of the message contained betwixt a pair of curly brackets.

The two words in front of the defintion say what kind of method it is. Firstly `public` means that the method can be accesses from objects other that objects of that class, so they can be accessed from the `main` program for example. The second word, `void`, says that communication is one-way for this method—no information is returned to the thing that sent the message.

Here is a simple `main` program which demonstrates how these methods are used. We access the method in the same way we access class variables—by using the dot between the object name and the method name.

```
class SecondBankApp
{
   public static void main(String[] args)
     {
        System.out.println("Bank account program, version two.");
        BankAccount accountNumber52342 = new BankAccount();
        BankAccount accountNumber34251 = new BankAccount();
        accountNumber52342.balance = 851.62;
        accountNumber34251.balance = 326.94;
        accountNumber52342.setNotAStudent();
        accountNumber34251.setAStudent();
        if (accountNumber34251.student==true)
        {
           System.out.println("Account 34251 "+
                  " belongs to a student");
        }
        else
        {
           System.out.println("Account 34251 "+
                " does not belong to a student");
        }
     }
}
```

Put together the new definition of the class with the new application class and run the program.

11.4 Parameter passing.

"Communication across the revolutionary divide is inevitably partial."

— *Thomas S. Kuhn*

So far our methods have just consisted of a "do this" statement. It would be useful if we could pass more complicated information as part of our message. Say for example we have a program that is used by the teller in a bank, and that you have a method that represents paying some money into the bank. Clearly we cannot have a different methods for each amount you might pay in : `payInFiftyPounds`,`payInTwentyPounds`, `payInSixtyFivePoundsAndThreePence`. Instead we *pass parameters* to the object through the method.

Here is the `BankAccount` class with a general `payIn` method.

```
class BankAccount
{
   String customerName = "Blank";
   String address = "Blank";
   int telephone = 0;
   boolean student = false;
   double balance = 0.0;

   public void setAStudent()
   {
      student = true;
   }

   public void setNotAStudent()
   {
      student = false;
   }

   public void payIn(double amount)
   {
      balance = balance + amount;
   }
}
```

The general structure of a method with a parameter is very similar to the earlier version. The only difference is that between the parentheses we have a variable type followed by a variable name. The variable name can them be used in the class defintion to stand for the piece of information that has been passed through to the object. Multiple parameters can be passed by having a list of parameters separated by commas, as in

```
  public void twoNames(String nameOne, String nameTwo);
```

Here is a main program which uses the above class definition.

```
class ThirdBankApp
{
   public static void main(String[] args)
      {
         System.out.println("Bank account program, version three.");
         BankAccount accountNumber52342 = new BankAccount();

         accountNumber52342.balance = 851.62;

System.out.println("Balance : "+accountNumber52342.balance);

accountNumber52342.payIn(32.45);
  //paying in a fixed amount

System.out.println("Balance : "+accountNumber52342.balance);

double aCheque = 45.53;
accountNumber52342.payIn(aCheque);
  //paying in via another variable

System.out.println("Balance : "+accountNumber52342.balance);
      }
}
```

Type the class defintion and the main application into files and get the program to compile and run. Modify it so that the user can *type in* the amount they want to pay in.

11.5 Computer exercises.

Notice that the final balance comes out at 929.6 rather than 929.60 as we would like. Find out from a textbook how to do *output formatting* and correct the program so that it displays the pence correctly.

11.6 Return types.

Look at the `BankAccount` class again. In order to access certain pieces of information within the account, we need to know the internal names of the classes. This sits uncomfortably with the idea(l)s of *encapsulation*—we would like to provide more programmer-friendly ways to access the information. We would also like to be able to send messages to the class asking for more complicated information than just a copy of the data. For example we would like to ask an object of the `BankAccount` class to calculate the difference between balance and overdraft limit and return this currently available cash total to the main program. Another example is sending a message to an object of the class asking that it return a string summarizing the content of the account.

The word `void` means that there is nothing to be returned from the method. To indicate that something *is* returned, we name the class of object that we would like the method to return. Here are a couple of examples.

```
class BankAccount
{
   String customerName = "Blank";
   String address = "Blank";
   int telephone = 0;
   boolean student = false;
   double balance = 0.0;
   double overdraftLimit = 0.0;

   public double returnBalance()
   {
     return balance;
   }

   public double cashAvailable()
   {
     double answer = 0.0;
     answer = balance + overdraftLimit;
     return answer;
   }
}
```

Notice that to return the answer we use the keyword `return` followed by the name of the variable we want to return. This also ends the method—any instructions in the method description after the `return` statement will be ignored. If the return type is `void` then there is no need for a `return` statement.

We can grab the value returned from the method by putting the method invocation on the r.h.s. of an assignment statement, or we can use the information directly e.g. in a print statement. Here is a main program which demonstrates these ideas.

```
class FourthBankApp
{
   public static void main(String[] args)
     {
        System.out.println("Bank account program, version four.");
        BankAccount accountNumber52342 = new BankAccount();

        accountNumber52342.balance = 851.62;
        accountNumber52342.overdraftLimit = 200.00;

        double bal;
        bal = accountNumber52342.returnBalance();

        System.out.println("Balance : "+bal);
        System.out.println("Cash available : "
                           +accountNumber52342.cashAvailable());

        System.out.println("Increase overdraft limit "
                           +"by 200 pounds.");
        accountNumber52342.overdraftLimit = 400.00;
        System.out.println("Cash available now : "
                   +accountNumber52342.cashAvailable());
    }
}
```

11.7 the `toString` method.

A particular example of a method which returns something is the `toString()` method, which returns a string which is some way “summarizes” the content of the object. We have come across this *in passim* as part of the lottery program in section 9.4. Go back and see if you can figure out how it works. There is another example in the farm program in section 5.5.

A `toString()` method is a `public` method with `String` return type. It has a special role within `println` and similar statements, in that if you ask for an object of a class to be printed (by naming an object of that class between the parentheses of a print statement), then the `toString()` method returns the string that will be printed. Work out what is going on in the two examples above, then do the exercise.

11.8 Computer exercise.

Write a `toString()` method for the `BankAccount` class, such that if you type something like

```
BankAccount accountNumber52342
  = new BankAccount();
. . .
System.out.println(accountNumber52342);
```

a summary of the account details is printed. Tinker with it so that the format looks good. Create a main program that shows how this works.

11.9 Pencil-and-paper exercises.

These are general exercises for this whole section.

1. (THIS QUESTION IS TAKEN FROM LAST YEAR'S COURSEWORK). A university is creating a computer system to hold student records and timetable details.
   (a) One example of a class is `Student`. Identify four other classes, and name some of the information and methods found in those classes.
   (b) Take the `Student` class and write in detail (in the English language) the information and methods required in that class. Include at least six pieces of information, in each case saying what type of data that information is, and at least eight methods, identifying the input and output types of each.
   (c) Write out a *Java* class definition for the `Student` class, incorporating the information and methods described above, and including at least one constructor method. There is no need to write out the functions corresponding to the methods.
   (d) Choose three methods from the class and write out *Java* function definitions for these methods. The three methods should be chosen to show three different aspects of the *Java* language—for example three print functions would not be an acceptable solution.

11.10 Computer exercises.

These are general exercises for this whole section.

1. Add some more methods to the `BankAccount` class to make is more realistic. Write an application class that represents a cash machine, and an application class which represents the computer system used by the cashier in a bank.
2. Return to the lottery example in section 9.4 and add some appropriate methods and a main program which demonstrates their use. Demonstrate all of the features discussed in this section.

12 Expressions and Operators.

We have been informally using a large number of little *expressions* in various contexts throughout the examples so far. For example we have used *assignment* expressions, where we copy something from one object to another, and we have used *relational expressions*, for example comparing two numbers or strings to check whether they are equal.

In context it is clear what is going on, but when it comes to creating your own programs which are different to the above you may need a broader range of these expressions. Thus the various kinds of expressions are summarized here in a fairly telegraphic fashion.

12.1 Assignment.

We have often come across the assignment operator = in a variety of contexts. Its job is to copy the results of what happens on the r.h.s. (which can be quite complicated) onto the l.h.s., which must be the name of a variable which has been created and initialized. Here are some examples.

```
int i,j;
String name;
i = 3;
j = 0;
j = i+2;
name = "Arthur Dent";
name = accountNumber.accountName;
name = "My name is"+person.returnName();
```

There are some more complicated assignment operators, such as `+=`, which takes the l.h.s. and adds it to the r.h.s., assigning that value to the l.h.s., i.e. `i += 2` is the same as `i=i+2`. Similar things are defined for all the arithmetic operators, and are a handy shorthand from time to time.

12.2 Relationals.

In the control-flow abstractions we were often testing whether things were equal, greater than, not equal to one another et cetera. Here is a list.

- == Equals
- != Does not equal
- < Is less than
- > Is greater than
- <= Is less than or equal to
- >= Is greater than or equal to

Notice that the relation of two things being equal is == and **not** =. Writing something like

```
if (i=2) // this is wrong
{
  . . .
}
```

is a double error—it not only doesn't do what you want (i.e. checking whether i equals 2), but it *sets* i equal to 2, dumping you into a deep quagmire. Trying to find bugs like this is nontrivial. The correct version is

```
if (i==2)
{
  . . .
}
```

These relational operators only apply to the built in numerical types like `int` and `double` and to the `char` type. To compare two `String`s use

```
String stringOne,stringTwo;
. . .
while (stringOne.equals(stringTwo))
{
  . . .
}
```

You can and should write your own `equals` methods for classes that you design yourself.

These relations can be connected together with *boolean* operators `&&` (and) and `||` (or). So for example

```
if ( (x==2) && (y!=4) )
```

reading "if x equals 2 and y does not equal 4 ...". There is also a boolean not, `!`, which can go at the beginning of a long sequence of expressions.

```
if ( !( ((x==3) && (y!=4)) || z<=3) )
{
   . . .
}
```

12.3 Arithmetic.

Again we have seen plenty of arithmetic already. Here is a summary

- + addition
- – subtraction
- `*` multiplication
- / division
- `%` Remainder (integers only!)

There are also `++` and `--`, which crop up in `for` loops a lot. `i++` means "add one to i" and `j--` means "subtract one from j". They are convenient abbreviations. Also a minus sign before a number has its usual meaning of the number being negative.

There are no power or factorial operators in *Java*.

12.4 Computer exercise.

Write your own power and factorial methods. If you make them methods of the application class then they can be invoked in `main` without an object name.

12.5 String arithmetic.

"String arithmetic" is a strange phrase, with somewhat dubious etymology, but it conveys the idea elegantly. We can act on numbers using arithmetic expressions for addition, division, et cetera. String arithmetic refers to things like connecting two strings together, splitting a string apart, et cetera.

The only actual *operator* which acts on strings is the *concatenation operator*, +. This takes the contents of the first string and puts it onto the front of the second string. So

```
String phraseOne,phraseTwo,wordOne;
phraseOne = "Hello "+"Dolly";
wordOne = " Dolly";
phraseTwo = "Hello"+wordOne+wordOne;
```

puts the string `"Hello Dolly"` into `phraseOne` and the string `"Hello Dolly Dolly"` into `phraseTwo`. Write a program which demonstrates this.

There is also a corresponding += operator. Other string arithmetic operations are defined by means of methods in the `String` class, such as `length()` which calculates the length of a string. You can read about this in the *Java* API index mentioned in section —refse:books. An examples was briefly mentioned above, in the exercises in section 6.3.

12.6 Precedence.

Many of you will be familiar with precedence rules in algebra and arithmetic. We learn that multiplication is conventionally performed before addition, such that $1+4\times 5 = 21$ and not 25. There exists a similar precedence structure in *Java* operators. You can use parentheses to make something clear, like $(1 + 4) \times 5$, and this carried through into *Java* too. I advise the generous use of parentheses rather than trying to rely on remembering lists of precedence.

In any case here is the list, for reference. Things at the top are performed first. Things within a row are performed from left to right (apart from the second line). So $5 * 4/3$ is "five times four" followed by "20 times three". This rarely matters, though occasionally it might : for example you might mutliply several large numbers together, causing an overflow, then the next operation might be to do some divisions. By rearranging the order so that the multiplications and divisions alternative, you can avoid the overflow error.

- `.`, `[]`, i.e. referencing and array access.
- +, in the sense of string concatenation
- `++`, `--`, `!`, `-` and typecasting.
- `*`, `/`, `%`
- `+`, `−`, where + is addition.
- `<`, `<=`, `>`, `>=`
- `==`, `!=`
- `&&`

- `||`
- `=, +=, -=, *=, /=, %=`

12.7 Pencil-and-paper / computer exercise.

What would you expect to be printed by the following statements

```
System.out.println(
   "Eight plus four equals "+8+4);
System.out.println(
   "Eight plus four equals "+(8+4));
```

Write a program incorporating these statements to check that your expectations are fulfilled.

13 Hip hip array!

This final main section deals with the beginnings of the idea of a *data structure*. We have seen above how to create different kinds of data, by using classes to represent different kinds of things found in the world, and using objects to represent particular examples of those things. However our examples have all be deliberately small scale—creating one or two bank accounts or student records or whatever.

A *data structure* is a way of organizing large sets of data in a sensible way. The example here is the *array*, which consists of a linear list of data indexed by integers. Imagine a line of pidgeon holes, each of which is able to hold one example of a particular kind of data (see figure 4), each of which is numbers with a number, starting at zero.

We can create such an array in *Java* to hold any kinds of data. This could be an existing data type, such as `int` or `String`, or it could be a class that you have created like `StudentRecord`. Creating an array is similar to creating a normal variable. For example to create an array capable of storing twenty `int` values we write a line like this

```
int[] myArray = new int[20];
```

To create a different kind of array we replace `int` both times with the type of object to be contained in the array, and replace `20` with the number of elements contained in the array.

Here is a short program to create and display an array.

```
import java.io.*;

class ArrayExampleOneApp
```

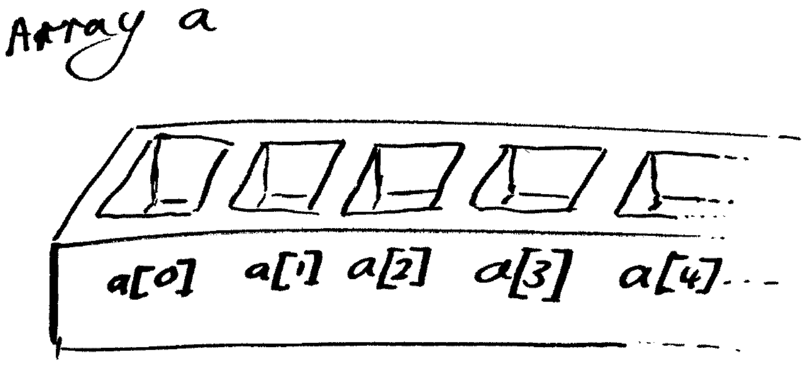


Figure 4: An array as a list of pidgeon-holes.

```
{
   public static void main(String[] args)
     {
       /* create a new array */

       int[] myArray = new int[20];
       int i;

       /* fill it up with square numbers */

       for (i=0;i<=19;i++)
         {
           myArray[i] = i*i;
         }

       /* print it in reverse order */

       for (i=19;i>=0;i--)
         {
           System.out.println("myArray["+i+"] contains "+
                                 myArray[i]);
         }

       /* add up a running total */

       int runningTotal=0;
       System.out.println();
       for (i=0;i<=19;i++)
         {
           runningTotal = runningTotal + myArray[i];
           System.out.println("Running total is "+runningTotal);
         }
     }
}
```

Type this program in and get it to run. Modify it so that different things are stored in the array, like the cube of the number or the characters 'a','b','c' et cetera.

Note that we can use a reference like `myArray[3]` or `myArray[i]` anywhere that we can use a normal integer variable. So we can print it out, use it in arithmetic, et cetera. Note also that we start counting from **zero**, so if we are creating an array with 20 members, then the elements are numbered **from 0 to 19**. The following is a very common error.

```
   char[] alphaArray[50];
   int i;
   for (i=1;i<=50;i++)   //this is wrong
   {
     alphaArray[i] = 'a';
   }
```

The correct version is

```
   char[] alphaArray[50];
   int i;
   for (i=0;i<=49;i++)
   {
     alphaArray[i] = 'a';
   }
```

We can create arrays of classes we define ourselves too. Here is the revenge of the bank account class from earlier (call this `arrayExampleTwo.java`).

```
import java.io.*;

class BankAccount
{
  String customerName = "Blank";
  String address = "Blank";
  int telephone = 0;
  boolean student = false;
  double balance = 0.0;

  public String toString()
  {
    String studentStatus;
    if (student==true)
      {
        studentStatus = "Account holder is a student";
      }
    else
      {
        studentStatus = "Account holder is not a student";
      }
    return "Bank account Summary.\n"+
            "Name : "+customerName+"\n"+
            "Address : "+address+"\n"+
            "Telephone Number : "+telephone+"\n"+
            "Account balance : "+balance+"\n"+
            studentStatus;
  }
}

class ArrayExampleTwoApp
{
  public static void main(String[] args)
       throws java.io.IOException
  {
    BufferedReader keyboard
      = new BufferedReader(new InputStreamReader(System.in));
    String inputString;
    int i,n;
    double d;

    BankAccount[] accounts = new BankAccount[5];

    for (i=0;i<=4;i++)
      {
        accounts[i] = new BankAccount();
      };

    for (i=0;i<=4;i++)
      {
        System.out.println("Account number "+i);

        System.out.println("Please type your name, "+
                           "then press the return key");
        inputString = keyboard.readLine();
        accounts[i].customerName = inputString;
        System.out.println("Please type your address, "+
                           "then press the return key");
        inputString = keyboard.readLine();
        accounts[i].address = inputString;

        System.out.println("Please type your telephone number, "+
                           "then press the return key");
        inputString = keyboard.readLine();
        n = Integer.parseInt(inputString);
        accounts[i].telephone = n;

        System.out.println("Please type the amount of money "+
                            "that you wish to pay in, "+
                            "then press the return key");
        inputString = keyboard.readLine();
        d = Double.valueOf(inputString).doubleValue();
        accounts[i].balance = d;

        System.out.println("Are you a student? Please type "+
                     "yes or no, then press the return key");
        inputString = keyboard.readLine();
        if (inputString.equals("yes"))
          {
            accounts[i].student = true;
          }
        else
          {
            accounts[i].student = false;
          }
        System.out.println("\nThank you. Next please!\n");
      }

    System.out.println("\t *** Account summary ***\n");
    for (i=0;i<=4;i++)
      {
        System.out.println(accounts[i]);
        System.out.println();
      }
  }
}
```

This example, though long, is fairly straightforward. Go through it carefully, making certain that you understand what is going on. Notice particularly the lines

```
    for (i=0;i<=4;i++)
```

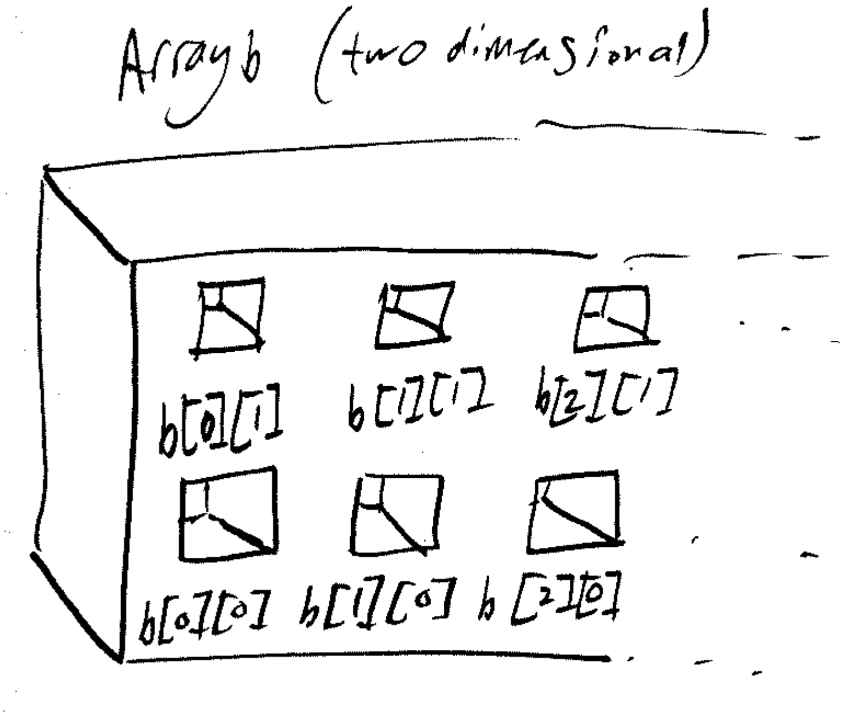


Figure 5: Two-dimensional arrays.

```
    {
      accounts[i] = new BankAccount();
    };
```

We need to create a `new` instance of *every* element in the array.
Also note the syntax for converting a `String` to a `double`.

```
      d = Double.valueOf(inputString).doubleValue();
```

13.1 Two-dimensional arrays.

The arrays that we have created above have been a list of variables. However it is also possible to create "two dimensional" arrays, which we can imagine as a grid of pidgeon-holes (figure 5). In a two-dimensional array each variable is referred to by two integers, so we can use it to hold grids of information like a chessboard, with each square specifying a piece or a blank symbol or the brightness of each pixel on the computer screen.

Here is a program containing a two-dimensional array. It fills the array with the number $i \times j$ at position (i, j).

```
import java.io.*;
```

```
class ArrayExampleThreeApp
{
   public static void main(String[] args)
      {
        int[][] myArray = new int[10][12];
        int i,j;

        for (i=0;i<=9;i++)
          {
            for (j=0;j<=11;j++)
              {
                 myArray[i][j] = i*j;
              }
          }

        for (i=0;i<=9;i++)
          {
            for (j=0;j<=11;j++)
              {
                 System.out.print(myArray[i][j]+" ");
              }
            System.out.print("\n");
          }

      }
}
```

Note that the syntax is basically the same as before, except that we have two pairs of brackets where previously we had one. The "nested for loop" construct is used to iterate through the whole array, with one `for` loop contained within another. We can create three, four, ..., n dimensional arrays in an analogous way.

13.2 Pencil-and-paper exercises.

Exercises for the whole of this section.

(THESE EXERCISES ARE ADAPTED FROM LAST YEAR'S CLASS TEST.)

1. Write a few lines of *Java* which declares one array that can hold data for 12 employees each having a name and age.
2. Give an example of a collection of real-world data which would be well represented by an array, and an example of one which would be badly represented by an array, explaining briefly why.

13.3 Computer exercises.

Exercises for the whole of this section.

1. Create a two-dimensional array of `char` (or a type of your own devising) which holds the positions of chess pieces. Set up the array so that the pieces are in their starting positions. Create a program which allows the pieces to be moved by typing in starting and finishing destinations—this could be very simple, making no checks as to whether the move is legal, or in could incorporate all sorts of checks. Include a routine which prints out a rough text-based picture of the board.
2. Create a simple word-search puzzle. The user types in a (say 5×5) grid of letters, and the computer returns a list of the words (taken from a fixed list) which it finds in the grid. Once you have learned about file-handling you can perhaps use the list of words in `/usr/share/lib/dict/words` for this kind of game.
3. Write a simple "battleships" game. The computer generates at random (you will have to discover some routines for random numbers from the Java API Index) the positions of a number of "ships", and the user guesses where the ships are, and the computer reports whether you have hit a ship or not. A simpler version could use a standard set of ship positions.
4. Similarly write a "noughts and crossses" game.

14 A substantial exercise.

This exercise was last years final programming assignment. Do you think that you would be able to carry it out? What things do you think that you would still need to find out about in order to carry out this exercise?

Question 1.

THIS QUESTION TESTS YOUR KNOWLEDGE IN UNDERSTANDING THE KEY CONCEPTS IN OBJECT-ORIENTED DESIGN.

A local shop requires a computer program to assist in stock keeping. The shop sells 20 different types of goods. Each article has a name, an identifier code, a retail price and its current stock quantity. It is required that when an item is sold by the shop assistant, it should automatically update the stock of that item and when the current stock falls below 30 pieces for any item a message should appear on the screen to signal the necessity of placing an order. The manager should also be able to view different types of information, such as: list of articles sold in the shop, list of articles out of stock, current stock quantity for all articles, current stock quantity for the specified article, value of current stock, number of items sold per article, number of purchases since opening the shop. The program should be operated either by the shop assistant or the manager. the type of user should determine the type of functions allowed to be performed. The shop assistant should be able to sell articles. The manager should be able to place order for articles, view stock information and update stock quantity.

1. Based on the description above, try to identify candidate classes for the system. List the classes in English, not in *Java*, with their attributes and operations. [MARKS : 10]
2. Explain why each of your classes would make a good class in the program. [MARKS : 6]
3. Describe the relationships between the classes. [MARKS : 4]

Question 2.

THIS QUESTION TESTS YOUR ABILITY TO INDEPENDENTLY DEVELOP A *Java* PROGRAM.

- Based on your identified candidate classes, design the final classes. Your answer should consist of the complete *Java* declarations of the classes, with sufficient comments to make it clear how each class works. [MARKS : 10]
- Implement your program. The program will be assessed by the following criteria:
  - To what extent the program solves the problem [MARKS : 10]
  - Modularity (a good use of classes and functions) [MARKS : 6]
  - Proper syntax [MARKS : 2]
  - Clear layout and readability [MARKS : 4]
  - Good, meaningful comments [MARKS : 2]
  - A critical evaluation of your work (errors/problems left in the program, possible improvements) [MARKS : 6]

  Your answer should consist of a few paragraphs describing how your program works and evaluating how well it performs the task, a complete listing of the source code, and a sufficient range of sample runs to demonstrate the full capability of your program.

[TOTAL MARKS : 60]

15 Final waffle.

FORD Six pints of bitter. And quickly please, the world's about to end.
BARMAN Oh yes, sir? Nice weather for it.

—*Douglas Adams.*

In this section you will find all the things which didn't fit into any of the sections above!

15.1 For future presentation.

The second section of the course will cover more advanced topics, including :

- More about input and output, including file handling and string operations.
- More complex data structures, e.g. linked lists, queues, trees, stacks.

- More details about class structure, including constructor and finalize methods.
- Inheritance of classes.
- Exception handling (`try` and `catch` statements).

Java will be revisited next term as part of the Media Computing course, where we shall look at how we can build good user interfaces and how we can exploit the power of connecting together computers across a network. The language will also be used during Project I, and the lectures leading up to the project will introduce some appropriate advanced topics in *Java* and in programming more generally.

15.2 The legalistic[20] bit.

15.3 Colophon.

This document has been written using Leslie Lamport's excellent LaTeX document processing system, using Donald Knuth's TeX as the underlying typesetting engine. This is a wonderful example of free software, and I urge you to take the time to learn this sometime. Have a look at

`http://www.tex.ac.uk/`

for some details. The main text font is 12-point Palatino, and the typewriter font is 12-point `Courier`.

16 Appendix: Portability, and obtaining the JDK.

The set of files needed to compile and run *Java* programs are referred to as the *Java Developers Kit*. You can obtain copies of this free of charge by downloading from the internet. The address for this is `http://www.javasoft.com/products/JDK/`. Versions are available for most home computers, including Linux, Windows and Apple machines.

[20]Originally this read "the legal bit", but all of the guide is *legal*—hence "legalistic".

Programs written in *Java* on one machine are portable to other machines (provided that you are not trying to run a program written in a later edition of *Java* on a machine with only an earlier edition installed). Moreover this compatibility extends down to the level of the executable file—a program compiled on an Apple Mac will run on a Unix machine, for example. This is because the *Java* compiler only compiles down to the level of the *Java ByteCode*, which is translated at runtime into the particular machine-specific instructions. The book *The Java Virtual Machine*, by Jon Meyer and Troy Downing, contains many details about this should you get interested in this sort of thing in the future. Not a book for beginners, though!