RHUL Department of Physics

PH2150 Scientific Computing Skills

Java Programmming

S. George and C. V. Quarman

Autumn Term 2011

RHUL Department of Physics
PH2150 Scientific Computing Skills
Java Programming
Version 2 revision 17.
September 23, 2011
First version September 2001.
By S. George and C. V. Quarman

Contents

Introduction

This five-week module is intended as an introduction to computer programming. The language you will use for this purpose is Java. This language, originally developed by Sun Microsystems and now owned by Oracle, has been one of the most widely used computer programming languages in the world for over a decade. It is best known for its use in association with the World Wide Web and the Internet, but over the next few weeks you will see that it can also be applied very effectively to scientific problems. Some big advantages of Java over other languages are that it is highly portable: Java programs run on most computer systems in use today; it is freely available; it is better integrated with graphics than other languages; it is relatively simple as it does not have the constraint of remaining backward compatible like older languages that have evolved; it is widely used, so Java programming is a marketable skill which you should find useful in many areas both inside and outside physics.

Java's portability means that you can use your programs on other types of computer without having to recompile them, and the program should behave in the same way regardless of the computer you are using. This is know as *platform independence*. It is for this reason Java has become widely used on the Internet.

These notes, example files, tips and up to date information about this course can be found at `http://www.pp.rhul.ac.uk/~george/PH2150`.

Objectives

This course gives you the opportunity to develop the following skills:

1. learn and practice the basic concepts involved in writing computer programs;
2. understand the main features of the Java language;
3. design and write simple programs in Java;
4. devise and use test procedures for your programs;
5. design and write Java programs to solve numerical problems;
6. gain the programming skills you need for the teamwork projects which follow this course.

Assessment

Assessment is based on the answers you give to the exercises included in the course. You should write down these answers along with any notes you wish to make in your lab book. Generally your answers should include printed listings of programs you write or modify and printed output from tests you have run to demonstrate that your program works.

Your answers will be assessed on the following criteria:

- quality and correctness of the answer — applies to both written answers and program listings and output;

- code presentation — your programs should be clearly laid out with appropriate use of variable/method names, comments and indentation;
- programming style — solutions should be simple, efficient and effective.
- testing — when appropriate, you need to demonstrate that your program works by testing its key features and comparing the resulting output with what you expected, e.g. a result calculated by hand. Make sure you write this down! You can't get marks for it, if it's only in your head.

Your final mark will be based on a subset of the exercises, but you must still attempt all of them. A few exercises may be worked through as examples in class if they are posing particular problems or raise interesting issues. The maximum marks for each exercise are shown beside the exercise in the script.

At the end of the course you must hand in for marking:

- your lab notebook, containing the answers, notes and printout described above;
- your programs: just the .java files, no .class files please, will be collected via turnitin, You will be given instructions for this later.
- include a table relating exercise numbers to program files names in your notebook.

You will be told where and when to hand in your book during the course.

Tips

- Write down your notes and answers **as you work**.
- **Do not** wait until the end to write everything up.
- Make sure that your **answers are clear** so that a reader can understand what you have done, otherwise it's hard to give you the marks you might deserve.
- **Read the questions carefully** to make sure you really do what you are asked.
- Identify each exercise by its number.
- Show your lab book to a demonstrator regularly so you can get some feedback on it.

1 Programming Basics

Computer programs and java

A computer program is fundamentally just a set of instructions for the computer to follow. *Running* a program means the computer executes the instructions that comprise the program.

Programming in Java involves the programmer writing a set of instructions in the Java language. The file this *Java code* is saved in, is known as a *source file* and has the name something.*java*. The source file must then be *compiled*[1]. The compiler reads and understands the program written in the source file, then translates it and writes it out into a second file (called something.*class*). This file contains the same program instructions written in a different form, to better suit the computer rather than you the programmer. Whenever the program is *run* the contents of this second file go through an *interpreter* which turns the contents into instructions that the computer's processor[2] can understand and carry out.

Basic program structure

We will start right away with a look at a Java program. For most of this course all your programs will have the same basic structure, which is reproduced below. This contains all the essential parts of code that must be included in any program.

line

```
1 public class Simple
2 {
3     public static void main(String[] argv)
4     {
5         //some program code goes here
6     }
7 }
```

So what does this all mean?

`public class Simple`
: Java programs are made up of *classes*, so the first line says that we are going to define a a *class* and we have chosen to call it Simple, though of course you could call it another name. (The convention is for class names to start with a capital letter.) The class Simple is everything contained within the outermost set of curly brackets {}. The purpose of the word `public` is something we will come to later; in the meantime you should always use it.

`public static void main(String[] argv)`
: Classes contain *methods*, and this line (3) is the start of the *method* called *main*. When the computer runs any Java program it starts by looking for the *main method* and carries out whatever the instructions are within it. All programs must contain one (and only one) *main method*.

[1] Don't worry if you don't know what this means yet; you will be shown how to compile a program during the first session.

[2] The processor is the part of the computer which does all the work, e.g. the Intel Pentium processor in your PC.

lines 4-6

Instructions for what the *main method* should do are in the form of some Java code which goes in the main method — i.e. within the corresponding set of curly brackets, `{}`. The `//some...` part on line 5 is called a *comment*, a note to yourself or others reading the file which is ignored by the computer. There are other ways to write comments which will be covered soon.

Also note how within the class all the other lines have been indented by four spaces, and within the method everything is indented by a further four spaces. The spaces don't have any effect in the program — they are just used to make the structure of the program clearer to see at a glance. You may also use blank lines occasionally for clarity as well.

Running a program on a lab PC

In this section you will learn how to run your first Java program.

Here is perhaps the simplest program you can write in Java.

```
/**
 *  Purpose: To write the word 'Hello' to the screen.
 */

public class Hello
{
    public static void main(String[] argv)
    {
        System.out.println("Hello");      // outputs Hello to the screen
    }
}
```

The program simply writes 'Hello' to your screen. You will see that it has the same basic structure as was shown before. The semi-colon, `;`, is used in Java to indicate the end of a line of code, or one instruction. Don't worry about understanding the `System.out.println` just yet, the program is just for use in the exercise that follows. The purpose of this exercise is to show you how to run Java programs on the computers you are using. You will now be taken through typing in the program and running it on the lab PCs.

Exercise 1.1 ________________________________ (1 mark)

You will be shown in the lab session how to use your computer to write, compile and run Java programs.

Following the demonstration, type in the example above, save it, compile it and run it. Read through the advice below as you work.

In appendix A you will find some notes on how to use Java on your PC.

Writing

Type in the program **exactly** as shown in the example above. Take care to copy upper and lower case correctly from the example, as Java is *case sensitive*: the program will not work if you get this wrong.

Saving

The code should always be saved in a file that has the same name as the *public class* and with the extension .java so you should save the above code under the name 'Hello.java'. This is your source file. Note that the filename is case sensistive.

Compilation

The compilation process produces a file called Hello.class, which then is used when you run the program.

If you get error messages when you compile the program, check very carefully what you have typed, then try again or ask a demonstrator. Beware that there is a distinction made between upper and lower case letters, so if for example you type `Class` rather than `class`, you will get error messages.

Especially when you are creating longer programs you are bound to make mistakes so you will get often errors when you first compile a program. Deal with one error at a time starting from the first that the *compiler* found. Sometimes an error at the beginning of a program can cause many errors later on and if you correct the first problem and compile again you may find some of the other errors have vanished. The error message will contain the line number and some description of the error which you can use to help find what went wrong. Appendix B contains more help on errors, you should refer to this when you come across problems.

Running

You should now see the word 'Hello' printed on the screen.

If you don't then you may get a message about an *exception* written to your screen though it is unlikely to happen with a program as small as we are dealing with here. *Exceptions* arise from problems that occur during the running of the program that were not detected by the compiler. You will be warned about the most common exceptions as we go along.

If you have an *exception* now, ask for help. You will learn how to solve these problems yourself as you gain experience.

Comments

Comments in a computer program are notes you make to yourself. They will be ignored by the compiler and do not affect the way the program runs. For this reason they must be marked. There are three ways to do this depending on the circumstances. The first looks like this:

```
/**
 *   Purpose: describe what the program does,
 *   add any extra lines needed using an asterisk
 */
```

This form should be used at the beginning of all your files and give any necessary information about your code, including its purpose, your name, the date and the exercise number where applicable. Especially with longer files you may find it helpful to include a modification history here as well, stating what changes were made and when.

The other two types of comment are:

```
// comments go here, rest of line is ignored

/* comments here may use
   more than one line
   and finish with */
```

When `//` appears on a line, everything to the right up to the end of the line is ignored by the compiler. Java code can appear to the left of `//`. Longer comments may take the second form shown. Anything between `/*...*/` is considered to be a comment by the compiler and ignored. Unlike `//`, the comment can span several lines.

As `//` comments may be placed within `/*...*/`, you may try putting the latter around a block of code in your program that you think is causing an error or exception, as long as the block of code only includes `//` comments[3]. Two sets of `/*...*/` comments can not be nested within each other.

It is important that you add sufficient comments to your programs so that it is clear what they do and how they work. When someone else reads your program they cannot read your mind, so they will need more help to understand it than you do. You will see more clearly how comments are put to use as you progress through the course.

[3] Appendix B, *Dealing with Errors*, explains this and other techniques for debugging.

Variables — declaring and assigning values

Before discussing input and output, including `System.out.println(...)` which you used in Hello.java, it will be helpful to cover the topic of variables. You will have come across the idea of variables in mathematics, and the concept of a variable in programming is similar.

A variable is associated with space in the computer's memory that can hold a value (often a number). Each variable has a name, which you choose, and subsequently use to refer to it.

You must start by *declaring* the variable, this gives the variable a name and reserves some memory space to store whatever value the variable takes. It also tells the compiler what you intend the variable to represent. This is known as the variable's *type*. For example a variable which always takes integer values, can be *declared* as *type* `int` in Java. For example the line of code

```
int i;
```

declares an integer variable (i.e. a variable of type `int`) which we have chosen to call `i`. As an `int`, it can store any integer value between -2147483648 and 2147483647.

Java provides *types* to represent several kinds of number, e.g. integer and floating point, non-numerical things like text, and other more abstract things. These are listed on page 9. You can give a variable a longer name if you like, and it is usually a good idea to choose a word that explains what the variable is for. The convention is for variables to be named using lower case letters, or if the name consists of more than one word, that a capital be used at the start of each word other than the first. You may also use numbers or an underscore _ in your variable names, but not at the beginning of the name. Examples of some well chosen and valid variable names might be `total`, `maxValue`, `answer1`.

Exercise 1.2 (1 mark)

Now try the simple program below. As it contains `public class VarTry` you must type it in to a file called 'VarTry.java'.

Compile and run VarTry in the same way that you did with the Hello program to check it prints out the number nine. Print the output and put it in your lab book. Read the explanation that follows and try to understand how it works.

```
/**
 * Name:
 * Date:
 * Exercise: 1.2
 * Purpose:  To demonstrate the declaration and simple uses of variables.
 */

public class VarTry
{
    public static void main(String[] argv)
    {
        int i;                     // declares integer variable named i
        i = 9;                     // gives i the value 9
        System.out.println(i);     // print the value of i to the screen
    }
}
```

You will see here how each line of code must end with a semicolon, `;` , (if you need to write a line of code that is longer than the width of the window you can continue it on the next line by omitting the semicolon until the end of the statement).

The line `i = 9` gives the value 9 to variable `i`. This is known as *assigning* a value to a variable — `i` is *assigned* the value 9. It means that the space in the computer's memory associated with `i` now holds this value. The first time a variable is assigned a value, it is said to be *initialised*. The `=` symbol is known as the *assignment operator*.

It is also possible to declare a variable and assign it a value in the same line, so instead of `int i` and then `i = 9` you can write `int i = 9` all in one go. If you have more than one variable of the same type you can also declare them together e.g.

```
int i, j;          // or
int i=1, j, k=2;
```

which can be a useful way of saving space. Where necessary you should add comments explaining the meaning of the variables, both so it is clear to you if you come to look at your program at a later date, and to those marking your programs.

You will of course want to use many quantities which are not integers, and there are several different variable *types* which cover these possibilities. For real numbers there are two possibilities — `float` and `double`. As `double` uses twice as much memory as `float` to store values, it is more accurate. For real numbers you will probably mostly want to use `double`. All the variable *types* are described below.

byte Integer variable allocated only 8 bits[4] (i.e 1 byte) of memory. May store values from -128 to 127.

short Short integer, allocated only 16 bits of memory. May take values from -32768 to 32767.

int Most commonly used form of integer variable, 32 bits. May hold any value in the range -2147483648 to 2147483647.

long Used if working with particularly large integers, 64 bits. Up to nineteen digits and a sign.

float *Floating point* real number, which must be written in the format `3.45`, or `3.0e-5` — that is they must include a decimal place and may include the letter `e`. What follows `e` is the power of ten, so the two examples mean 3.45 and 3.0×10^{-5} respectively. A `float` variable is 32 bits and holds a value between $\pm 1.4 \times 10^{-45}$ and $\pm 3.4 \times 10^{38}$, to eight significant figures.

double A 16 significant figure, floating point, real number, 64 bits. Values are written as for `float`, e.g. `5.8e59`, and may take value between $\pm 4.9 \times 10^{-324}$ and $\pm 1.8 \times 10^{308}$.

boolean May take one of only two possible values, *true* and *false*. This is a *logical truth* variable (1 bit).

char A single character (this may be an upper or lower case letter, number or other keyboard symbols like `:`, `#` or `!` for example.)

You can also create a `String` of characters. A `String` is like a word or a line of text and can include spaces, upper and lower case letters, numbers and other keyboard symbols. Strings are created in a similar way to variables, and the double quote symbol `"` is used to mark the start and end of the text. You may for example write

```
String name = "Bob1";
```

This creates a `String` called `name` which stores 'Bob1'.

You should be aware that there are some words which you may not use as names for variables (or methods or classes for that matter) as they have a special meaning in Java. These are: `abstract, boolean, break, byte, byvalue, 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, strictfp, super, switch, synchronized, this, throw, throws, transient, true, try, var, void, volatile, while, widefp.`

[4]A *bit* is the smallest unit of computer memory. It may be thought of as either a 1 or a 0 in some binary code. 8 bits make up 1 *byte* of memory.

Exercise 1.3 (6 marks)

With reference to the section above, say what type of variable would be suitable for the following, and suggest a name for it too.

It may help to think of possible values and the range which would be valid before you choose a type.

Example: the temperature of a room in C. *Room temperature is likely to vary between 15 and 35 C but it can take any value inbetween including non-integers, e.g 20.37 C, so the type must be **float** or **double** depending on the accuracy required. A suitable variable name, following the conventions described above, would be **roomTemp**.*

(i) the number of pages in a book;

(ii) the number of atoms in a book;

(iii) the length of a side of a triangle in metres;

(iv) your name;

(v) whether or not a nucleus has decayed;

(vi) the probability that it could have decayed.

Simple output

In the Hello program you used the line `System.out.println("Hello")` to write 'Hello' to the screen. This is the standard way of outputting to the screen and it works for numbers and variables (see previous section) as well. If you want to print several things at the same time you can use the `+` symbol as follows.

```
System.out.println("Hello" + 1.234);
System.out.println("Hello " + "number " + 1.234);
System.out.println("The value of variable x is: " + x);
```

The first two commands result in the outputs 'Hello1.234' and 'Hello number 1.234' respectively. The third command will print to the screen 'The value of variable x is: ' followed by whatever number, character, or string, is stored in the `x`.

You can add separate Strings together using `+`, e.g. `String name = "Bob" + " Smith"`. Or `String id = "abc" + 1.4` creates a string called `id` which stores 'abc1.4'. Here 1.4 is a number but it is automatically converted into a string before being added to `"abc"`. We explore strings a little more in the next section, which is on output, where you will see that it is equally possible to have instead of a number, a variable, where the value of the variable is added to the string in the same way that the number is above.

Exercise 1.4 (4 marks)

Try programming output yourself.

- Make a copy[5] of your program VarTry.java, call it VarTry2.java, and change the class name to match the new file name.
- Alter VarTry2 so that you also declare a `double` variable, `x`, assign it a value and print this to the screen.
- Compile and run your program to check the output is what you expect.
- Next modify your program to create a string and print that to the screen as well. Compile and run your program to check the output is what you expect.
- Finally modify your program again to declare a second `double` variable `y`, assign it a value, and add the statement `System.out.println(x + y);` to your code. Again, compile and run your program to check the output is what you expect.
- Write in your lab notebook what the program outputs to the screen.
- Alter your program to `System.out.println(x + " " + y)` to see the two values printed to your screen separately.
- Your program listing, sample output and your answer to all these questions should all go in your lab notebook.

If you wish to print several things to the screen on the same line without using `+` all the time, you can use System.out.print() which works in the same way as the System.out.println() you've just been using, but will not move to the next line each time.

Arithmetic operators

Now that you are able to *initialise*[6] and output data, its time to do something with it in between. For numbers or numerical variables there are the operators `+`, `-`, `/`, and `*` which add, subtract (or reverse sign), multiply and divide respectively, just as in mathematics. There is also % which gives the remainder when the first number is divided by the second, e.g. the value of `4 % 3` is 1.

There is also the *assignment* operator, =, which you used earlier. This does not act in the same way as an equals sign in maths, instead it puts the value which is to the right of the operator into the variable which is to the left. E.g. `x = 1` gives x the value 1, `x = y` gives x the value of y, `x = 2*y + 1` gives x the value of $2y + 1$, and `x = x + 1` adds 1 to the value of x.

[5] It is a good idea to save versions of your programs as they develop so that if the changes you make cause new errors you have a working previous version of your program to refer to. Remember to change the name of the public class so it matches the filename.

[6] Initialise — assign a value to a variable for the first time

Expressions within round brackets are evaluated first, then division, multiplication, addition and subtraction operations in that order — so you can add brackets where necessary to influence the order in which the expressions are evaluated. E.g. `2 + 4/2.0` returns the value 4 whereas `(2 + 4) / 2.0` the value 3. You will also need to use brackets sometimes in situations such as `x* (-y)`. Leaving the brackets out here would not only make the meaning unclear but also cause two operators to be placed next to each other which is not allowed.

Exercise 1.5 ——— (5 marks)

Copy the following source file, Division.java, from the course web site:
`http://www.pp.rhul.ac.uk/~george/PH2150/downloads/Division.java`

(i) Add three lines to this source file so that the program also prints out the difference of the two variables `i` and `j`, the product of `i` and `j`, and the ratio `i/j`. Run, and record in you lab book the results *exactly*. Comment on whether these are the answers you would expect?

(ii) Alter the third line that you added so it now gives `i` divided by `j` as a whole number and a remainder. Also print out the value of `((double)i)/j`. Again record the exact output of your program in your lab book.

You do not have to type in this program.

```
/**
 * Name:
 * Date:
 * Exercise: 1.5
 * Purpose:  To demonstrate use of arithmetical operators and
 *           a common problem encountered with integer division.
 */

public class Division
{
    public static void main(String[] args)
    {
        // declare and initialise variables
        int i = 5, j = 3;

        // print the numbers and the sum of i and j to screen
        System.out.println("The values are: i is " + i +
                           ", j is " + j);
        System.out.println("The sum of these integers is " + (i + j) );
    }
}
```

What this exercise aims to demonstrate a potential difficulty with dividing one integer by another. If both the numbers involved are integers, an integer result is expected so the answer

is rounded down to a whole number — thus 5/3 gives 1. If using numbers in your program you could instead write `5.0/3` or `5/3.0` to get a decimal answer. The default type for a decimal number in your code is double, and when one number in the division is double, the result is type double as well. Obviously if you are using a variable just adding `.0` is not much use, so instead there is a way of converting the value of an `int` variable into a `double` before the operation is carried out. This is the `(double)i` that you have just made use of in exercise 1.5.

2 Input, mathematical functions and conditional branching

Keyboard input

It would obviously be much more useful to create a program which could add together any two numbers rather than just two specific numbers given in the program. Recompiling every time you wish to try a program with new values is not satisfactory. The way to enter numbers via the keyboard into your program while it is running is therefore given below. Input is slightly more complicated than output, so you are not expected to understand the code at this stage.

The beginning of your program will require a few extra pieces of code, so that it looks like this:

```
/**
 * Purpose:  Shows things you must add to allow input from the keyboard.
 *           These include "import java.io.*" and "throws IOException" in
 *           addition to the lines indicated in the main method.
 */

import java.io.*;

public class InputExample
{
    public static void main(String[] argv) throws IOException
    {
    // code needed for keyboard input
    BufferedReader br = new BufferedReader(
                          new InputStreamReader(System.in));
    String temp;

    // rest of your program code here, as usual

    }
}
```

Then whenever you wish to read something from the keyboard, simply write

```
    temp = br.readLine();
```

You will probably find it helpful though to add a line like `System.out.println("Enter Input")` or similar whenever you use this so that it is obvious the program is waiting for some input. When running the program, you will know to type in something when you see this prompt. Press the return key to finish and let the program continue with your input.

The input is stored in the string `temp` so if you want to use the input as anything other than a string it needs to be converted into the right variable *type*. Java provides a way to do this as shown in the examples below.[7]

[7] **NB** the *type* conversion examples above work for JDK 1.3, but older versions of Java, e.g. J++, do not support this syntax. Instead you have to use something like this:
`x = Double.valueOf(temp).doubleValue();`


```
x = Double.parseDouble(temp);            // string temp converted to double x
y = Float.parseFloat(temp);              // string temp converted to float y
i = Long.parseLong(temp);                // string temp converted to long i
j = Integer.parseInt(temp);              // string temp converted to int j
k = Short.parseShort(temp);              // string temp converted to short k
m = Byte.parseByte(temp);                // string temp converted to byte m
a = Boolean.valueOf(temp).booleanValue(); // string temp converted to boolean a
```

Note that `x`, `y`, etc. must first been declared as their respective variable *types* before you assign the result of the string conversion. There will be a chance to practice this input method in the next exercise.

This is the standard way to input data from the keyboard. It looks quite complicated, but you should just copy the relevant code whenever you want to input data from the keyboard and for the moment trust that it works. By way of a brief explanation: `import` tells the compiler that you will be using some software from outside your program. The name `java.io.*` tells it what this is and implies to the compiler where the necessary files can be found. If there is the possibility of an error occuring in the program, it can be handled using a Java feature called *exception handling*. The BufferedReader and InputStreamReader used to get the keyboard input can throw up *exceptions* or errors, in this case called an `IOException`. This **must** be handled in your program, and the simplest way to do this is to declare that `main throws IOException`. Later in the course you will come to understand these concepts better.

There will be a chance to practice this input method in the next exercise.

Mathematical library

It may be necessary in a program to use common mathematical functions such as the sine, square root, or log of a number. There are some standard methods Java provides to do this simply. To calculate $y = x^2$ it is easiest to use the expression `y = x*x`, however for higher powers of x this is awkward, so in general $y = x^n$ is calculated using `y = Math.pow(x,n)`.

The values or variables placed in the round brackets, `()`, are known as *arguments*.

These useful methods providing mathematical functions exist:

```
Math.E            // e as a double
Math.PI           // pi as a double
Math.sin(x)       // sine of x
Math.cos(x)       // cosine of x
Math.tan(x)       // tangent of x
Math.asin(x)      // arcsine of x
Math.acos(x)      // arcsine of x
Math.atan(x       // arctangent of x
Math.exp(x)       // exponential of x
Math.log(x)       // natural logarithm of x
Math.pow(x,n)     // x raised to the power n (both double)
Math.sqrt(x)      // square root of x
Math.random()     // random double between 0.0 and 1.0 (uniform distribution)
                          // requires no argument
```

```
Math.abs(x)         // the absolute value of x, works for int, long, float and
                           // double values, returning value of the same type
```

Except `Math.abs()` these mathematical functions all give result values of type `double`. They also all take a `double` variables or values as their *arguments* — both `x` and `n` in the above are `double` variables. Note that `Math.pow()` requires two *arguments*, whereas `Math.random()` requires none, although the brackets must still be included. Using `Math.abs()` is slightly different — using it on a `float` variable will give a `float` value, on an `int` variable will return an `int` value, and so on.

Note that all the trigonometric functions work in radians.

Exercise 2.1 ———————————————— (6 marks)

Write a program that reads in a value for an angle from the keyboard, computes and writes out the sine and cosine and, as a check, the value of $sin^2x + cos^2x$ for that angle. The value of $sin^2x + cos^2x$, of course, should be very close to 1. In your laboratory notebook, record the results of entering the angles 3.5 and 2.3e-3. You should use variables of type double and include all significant figures in your written answer. Stick a print out of your program in your lab notebook.

The `if` statement and comparison operators

We use `if` in a program so that a piece of code may only be carried out under certain circumstances, which we can choose, by specifying some conditions that must be satisfied. It is one way in which the *flow*, the order in which lines of code are executed when the program runs, can be controlled, (you may find it helpful to think of `if` and other forms of *flow control* by imagining a flow chart). A basic example of the code for an `if` statement is as follows:

```
if ( boo )
{
    x = y;
}
```

In this short piece of code, x is only set equal to y if whatever condition we have chosen to put in the round brackets () is true. This is where the boolean variable type briefly mentioned earlier may come in useful. If the value of a boolean variable `boo` is true, he code in the curly brackets `{}` (in the above case `x = y`) is executed. If `boo` is false, the code in curly brackets `{}` will **not** be executed.

Notice how within the curly brackets `{}` code is indented. This is simply to make it clearer that the code is contained within an `if` statement.

More usefully for the scientist, we can write things like

```
if (w < z)
{
   x = y;
}
```


As you might expect `if (w < z)` means 'if w is less than z', and in the above code `x = y` only happens if it is true that w is less than z. It is also possible to use the same variables in the condition, i.e. between the round brackets `()`, as between the curly brackets `{}` if you need to.

Less than, `<`, is an example of a *comparison operator* — it compares two variables, asking whether one is less than the other, returning a `boolean` result — *true* if the condition is satisfied and *false* if it is not. The other *comparison operators* are in the table below. Note the difference of `w == z` i.e. 'does w equal z?' to `x = y` i.e. 'set the value of x equal to y' which is the assignment operator we met earlier.

`<` Less than
  E.g. `w < z` is true if the value of `w` is less than the value of `z`.

`>` Greater than
  E.g. `w > z` is true if `w` is greater than `z`.

`==` Equal to
  E.g. `w == z` is true if the values of `w` and `z` are equal.

`<=` Less than or equal to
  E.g. `w <= z` is true if `w` is less than or equal to `z`.

`>=` Greater than or equal to
  E.g. `w >= z` is true if `w` is greater than or equal to `z`.

`!=` Not equal to
  E.g. `w != z` is true if `w` and `z` have different values.

Here's a simple example where `if` is used to compute the modulus, i.e absolute value, of any `double` value entered.

```
/**
 * Purpose:  An example to demonstrate use of 'if'
 */

import java.io.*;

public class IfExample
{
    public static void main(String[] args) throws IOException
    {
        //Read in a value from the keyboard
        BufferedReader br = new BufferedReader(
                              new InputStreamReader(System.in));
        System.out.println("Enter positive or negative number...");
        String temp = br.readLine();

        double x = Double.parseDouble(temp);

        if ( x < 0 )
        {
            x = -x;
        }

        System.out.println("The modulus of this number is " + x);
    }
}
```

Else and else if

There is also the option of adding `else` after an if — the `else` section should contain parts to be executed if the condition is *not* true. A trivial example of this is given below, where a value for `boo` is displayed on the screen.

```
    boolean boo;

    // some code goes here which gives boo the value true or false

    if ( boo )
    {
        System.out.println("if: boo is true");
    }
    else
    {
        System.out.println("else: boo is false");
    }

    // print value of boo to the screen directly
    System.out.println("boo is " + boo);
```


There is also an optional `else if` that can be added. This works like an `else` but you get to specify another test condition as well. Here is an example:

```
/**
 * Purpose: An example to demonstrate the use of 'if'
 */
import java.io.*;
public class IfExample2
{
    public static void main (String[] args) throws IOException
    {
        BufferedReader br = new BufferedReader(
                              new InputStreamReader (System.in));
        System.out.print ("Enter a number between 0 and 10 inclusive: ");
        String temp = br.readLine();
        double x = Double.parseDouble(temp);
        // check user input
        if (x > 10)
        {
            System.out.println("The number you entered is too high");
        }
        else if (x < 0)
        {
            System.out.println("The number you entered is too low");
        }
        else
        {
            System.out.println("The number you entered is " + x);
        }
    }
}
```

This program checks whether input is in the range 0-10. If the input is larger than 10 the first condition is *true* and the `if` block of code is executed. If x is less than 10 the second test is performed — `x<0`. If the second condition is true the `elseif` block of code is executed. Otherwise, if both conditions were *false* the final `else` block of code is executed.

You may use as many `else if`'s as you like after an `if`, allowing tests of many different conditions such as `x<0`. If an `else` is used it always comes at the end, being executed only if all the conditions preceding it are found to be *false*.

Exercise 2.2 (10 marks)

Create a program to solve quadratic equations making use of the well-known formula for solving quadratics. Assuming the form of the quadratic is $ax^2 + bx + c$, read in the three numbers a, b, c from the keyboard. If the equation has complex roots, your program should output a message to the screen to this effect and not try to calculate them, otherwise the one or two roots of the equation should be printed to the screen. Include in your lab book a printout of your program and sample output to show that it works when tested for complex roots, a single root, and two roots, by using it to solve the following equations:

(i) $x^2 - 4x + 4$

(ii) $x^2 - 4x + 8$

(iii) $2x^2 + 13x + 21$

Logical operators

It is also possible to test whether more than one condition is satisfied at the same time. E.g.

```
if ((w < z) && (a == b))
{
    x = y;
}
```

This uses what is known as the AND *logical operator*, which is represented in the code above by the symbol `&&`. It does exactly what the name suggests — the code will set `x` equal to `y` only if **both** `w` is less than `z` **and** `a` is equal to `b`. Look carefully at the use of round brackets in the example above. These are important as the conditions within the innermost brackets are evaluated first. When these are found to be either *true* or *false*, the `&&` operator asks whether both sets of brackets are *true*.

There are also other logical operators, including OR and NOT which again do rather what their names suggest.

`&&` AND operator
E.g. `(boo1 && boo2)` returns *true* only if both `boo1` *and* `boo2` are *true*. Returns *false* otherwise.

`||` OR operator
E.g. `(boo1 || boo2)` returns *true* if either `boo1`, `boo2`, or both are *true*. Returns *false* otherwise.

`!` NOT operator
E.g. `!boo` returns false if `boo` is *true* and vice versa.

Exercise 2.3 (4 marks)

Look at the code given below. What is the output from this program when:

(i) a = 1, b = 2, c = 3, and d = 4?

(ii) a = 2, b = 4, c = 2, and d = 1?

(iii) a = 7, b = 4, c = 2, and d = 1?

(iv) a = 3, b = 1, c = 3, and d = 1?

Work it out by hand, then write a program to check your answers if you wish.

```
if ((a>b) && (c!=d))
{
    System.out.println("First if block of code executed");
}
else
{
    System.out.println("First else block of code executed");
}

if ((a==c) || (b<=d))
{
    System.out.println("Second if block of code executed");
}
else
{
    System.out.println("Second else block of code executed");
}
```

3 Iteration

One of the most common and powerful ways to control the flow of execution through a program is by using iteration, also known as *loops*. A *loop* is just a way of telling the computer to repeat some code many times. The number of iterations can be fixed, or depend on the outcome of calculations made within the *loop*.

There are several types of *loop* available in Java. The first to be introduced is called a `for` loop.

For loops

Here is an example of a `for` loop:

```
for (int i=0; i<10; i++)
{
    System.out.println(i);
}
```

The first line, starting with the `for` statement, defines the loop. The code to be executed inside the loop is contained within the curly brackets, {}. Code within these brackets is indented to make it easier to see that it is inside the loop.

There are three expressions within the round brackets after the `for` statement. The first (`int i=0`) is known as the *initial expression*. It is used to do things like declare and initialise variables before the *loop* starts. It is evaluated only once. The second expression within the round brackets (`i<10`) is the *test expression*. This is evaluated at the beginning of each iteration. It must be a *logical expression*, as described in the previous section on `if` statements. If it is found to be *true*, the iteration continues and the code within the *loop* is executed. If false, the *loop* ends. The final expression within the round brackets (`i++`) is the *update expression*. This is evaluated at the end of each iteration, after the code within the *loop* has been executed.

In the example, `i` is declared as an int and initialised to 0. Then the test `i<10` is evaluated. This is clearly *true* since `i` is 0. So the line of code inside the *loop* is executed with the result that the value of `i`, 0, is printed to the screen. The *update expression* `i++` is short-hand for `i = i + 1`, i.e. the value of `i` is increased by one. This is repeated until it is no longer *true* that `i` is less than 10. At this point, the test fails and the *loop* ends.

Exercise 3.1 ——— (4 marks)

First, predict exactly what numbers the above code would print out. What are the first and last numbers that will be printed? Explain your reasoning. Then, write a program containing the above code, run it and compare the results with your prediction. If you got it wrong, explain what actually happened and why.

Note: the ++ operator is very useful in *loops*. There is also a -- operator, which does the opposite, i.e. subtracts 1 from the variable.

You can put any valid Java code inside the `for` *loop*. For example, variable declarations, `if` statements, calculations, mathematical functions, and even another *loops*.

While loops

Below is an example of a `while` *loop*. Like the `for` *loop*, it contains a *test expression* (`i<10`) and some code inside the *loop* to be repeated as long as this expression remains *true*. The variable `i` must be declared and initialised before the loop as you see on the first line, `int i = 0;`. The example below does exactly the same thing as the `for` *loop* above.

Example:

```
int i = 0;
while ( i < 10 )
{
    System.out.println(i);
    i++;
}
```

The following example shows another simple use of the `while` *loop*. For each *loop* iteration, x is doubled and the new value is printed out. This continues until x exceeds 1024. Hence powers of two are printed up to 1024.

```
// Print powers of 2 up to 1024.
float x = 1;
while ( x < 1024 )
{
    x = x*2;
    System.out.println(x);
}
```

Exercise 3.2 ——— (5 marks)

Take the example class `IfExample2` (page 19) which you can download from the course web site: `http://www.pp.rhul.ac.uk/~george/PH2150/downloads/IfExample2.java`. Modify it so the user is asked to re-enter a number if the input is not valid, i.e. outside the requested range. Include the program listing and sample output in your lab book. Hint: use a `while` *loop* around the input.

So, which sort of loop should you use — `for` or `while`? They are both very flexible, so in almost any situation, either could be used to create the desired effect. The choice is therefore mainly a question of style and convenience. As a rough guide, `for` is well-suited to *loops* which have a definite number of iterations that you know in advance, as in the example above. Is also most appropriate when you need a variable inside the *loop* which keeps count of the number of iterations, like `i` in the above examples. A `while` *loop* is best when you don't know how many iterations there will be, but you can write a *logical expression* which tells you whether to continue for another iteration or not. If in doubt, a good rule is to use which ever type of *loop* seems simplest.

There is a third *loop* statement, `do`, which is very similar to `while`. Again, the example does the same thing as the two previous examples.

```
int i = 0;
do
{
    System.out.println(i);
    i++;
}
while (i < 10);
```

The only difference between `do` and `while` is that the *test expression* (`i<10`) is evaluated at the end of each iteration, instead of at the beginning. Therefore the code inside the curly brackets {} is always executed at least once.

Note: it is possible to create an *infinite loop*, i.e. one which repeats for ever. A simple example would be:

```
while (true)
{
    // never ends
}
```

If you have such a *loop* in your program, with no other means of escape the program will run for ever. If this happens, you can usually stop the program by pressing Ctrl-C.

Flow control

There will be sometimes be points in the code inside the *loop* where it will be convenient to jump out of the *loop* completely, or skip the rest of the *loop* code and proceed directly to the next iteration. In Java, there are commands to do this. They are, respectively, `break` and `continue`. They allow finer control of the program flow, as the *loop* can be ended without waiting until the next time the *test expression* is evaluated. You should use these commands when it makes your code simpler or easier to understand. They are most useful when your programs become larger and more complex. The example below shows how they are used.

```
// generate random numbers and sum those that are <= 0.5,
// until the running total exceeds 6.0.
double sum = 0;
while (true) // infinite loop, but break will be used to escape
{
    // get a new random number
    double r = math.random();

    // if r is outside the required range, skip ahead to the next iteration
```



```
    if (r > 0.5)
    {
        continue;
    }

    // do something with the random number
    sum = sum + r;
    System.out.println("random number " + r + " running total " + sum);

    // exit the loop when the running total exceeds the target
    if (sum > 6.0)
    {
        break;
    }
}
```

Scope

Now that you have learnt about the main control statements `if`, `for`, `while`, you need to know about something called *scope*. The *scope* of a variable is the region of the program within which the variable can be referred to. A variable will only exist from the point at which it is declared until the end of the block of code it is in. A block of code is contained within curly brackets. If you declare a variable inside a *loop*, it will not be available outside the *loop*. At the end of the braces, the variable “goes out of scope” which is to say that it is no longer recognised by the compiler or available to use in your program. Study the examples below:

```
// Scope example 1 - this won't compile because of the last line.
// the scope of i is inside the loop
for (int i=0; i<10; i++)
{
    System.out.println(i);
}
System.out.println("final i=" + i); // wrong, outside the scope of i

// Scope example 2 - this is a corrected version of scope example 1
// the scope of i is outside the loop, because i is declared before
// the for loop.
int i;
for (i=0; i<10; i++)
{
    System.out.println(i);
}
System.out.println("final i=" + i); // correct, i is still in scope

// Scope example 3 - this won't compile either
boolean test = true;
if (test)
{
    boolean success = true;
    System.out.println(success); // this is ok
```

```
}
System.out.println(success); // wrong - outside the scope of ‘success’
```

If the third example is modified so that the boolean variable ‘success’ is declared before the `if` statement, then it will work.

In general it is a good idea to declare variables within the most limited scope that they need, and to declare them as near as possible before they are used. This means if you use the variable by mistake elsewhere, you are more likely to get a compiler error rather than strange run-time behaviour, which is hard to debug.

So, if a variable is only needed inside a *loop*, it is best to declare it inside the *loop*. For example, if you like to use `i` as the iterator in `for` *loops*, you can declare it in the *initial expression* of the `for` *loop*, as shown in the example at the beginning of this section. Since `i` is now limited to the scope of the *loop*, you can declare it again in the next *loop* and be sure you are getting a different variable every time.

Exercise 3.3 (4 marks)

Which of the following examples will not compile due to a scope-related error? Explain why not, before you try to compile them. You can use the compiler to see if you are right. NB compiler error messages are not an acceptable answer to this question — please give a full explanation of any flaws you identify.

```
// Example 1
// Sum the integers from 0 to 25?
for (int i = 0; i<=25; i++)
{
    int sum;
    sum = sum + i;
}
System.out.println(sum);

// Example 2
// Sum the absolute value of integers from -25 to 25?
int sum = 0;
for (int i = -25; i<=25; i++)
{
    if ( i < 0)
    {
        int j = -i;
    }
    else
    {
        int j = i;
    }
    sum = sum + j;
}
System.out.println(sum);
```


4 Arrays, nested loops and composite operators

Creating an array

Arrays are useful when an ordered group of values need to be stored. The values should be of the same *type* (`int`, `double`, etc.) and there should be a fixed number of them, for example the entries of a vector or matrix. There are similarities in the behaviour of arrays and variables, but also some differences — here is how to create an array that will contain entries that are of *type* `double`.

```
// way to create an array and fill with some numbers
double[] vector1 = {1.2, 0.0, 5.3, 1.4};

// way to create an array but fill in values later
double[] vector2;
vector2 = new double[4];              // where 4 is the size (no. of entries)
```

This code creates the arrays {1.2, 0.0, 5.3, 1.4} called `vector1` and {0.0, 0.0, 0.0, 0.0} called `vector2` — note that unlike a variable, if you do not specify initial values, all entries are automatically initialised to zero. The first line in both examples doesn't look unlike declaring a variable, and you can equally well use other data *types* as desired, to get arrays of integer values for example. The second line in both cases gives the length or size of the array, for `vector1` this is done by explicitly giving the values of each *element*, in the case of `vector2` space is reserved for four values of *type* `double` but the values will be filled in later in the code. As with variable declarations the two lines of code can be reduced to one, e.g. `double[] vector2 = new double[4]`.

Figure 4.1 illustrates the creation of an array, which results in the values in the array occupying four adjacent words in the computer's memory. The array itself is a reference to this data.

```
double[] a = {1.0, 2.0, 3.0, 4.0};
```

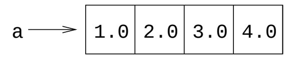


Figure 4.1: *Example of an array of doubles in memory.*

Array manipulation and references

To make use of the values stored in arrays you should refer to each *element* individually. The numbering of each *element* starts at zero, thus for `vector1` above, the number 1.2 is the zeroth *element* `vector1[0]`, 0.0 the first *element* `vector1[1]`, and 5.3 the second *element* `vector1[2]`, and so on. Individual array *elements* `vector1[2]` can then be used as any double variable would be, so you can for example assign a number to it, use it in a calculation, like this:

```
    System.out.print(vector1[1]);
    vector1[0] = 7;
    vector1[2] = vector1[0]*(-7.0) + vector1[3];
```

Often though you will want to perform the same or similar operations on all the entries of an array in turn. This can be done using loops, and shows the major advantage of an array over using many individual variables.

```
// create array called 'list' and then print out elements to screen
int[] list = {1, 4, -3, 66, 19};
for (int i=0; i<5; i++)
{
    System.out.println(list[i]);
}
```

Which saves us typing out `System.out.println(...)` five times. Note that here the loop variable `i` runs from 0 to 4 as the *elements* of the array are numbered from zero. It is a very common mistake to let the loop variable increase to too high a value. For example it would be easy to type `i<=5` by mistake, by thinking about the array having five entries. If a variable does go beyond the end of an array, the error will not be picked up by the compiler, but instead results in an `ArrayIndexOutOfBoundsException` when the code is executed. Also be careful to start with `int i = 0` to use the first array entry.

```
// create array of general length filled with random no.s
double[] randomNos = new double[N]; // value of N set somewhere prev.
for (int i=0; i<N; i++)
{
    randomNos[i] = Math.random();
}
```

In this example the loop not only saved us typing `randomNos[...] = Math.random();` many times (N could be 10, 1000, 100000 or more!), but in fact we don't even need to know whether the array is 10 or 100000 long, so long as we have the value stored in N.

If you wish to use the length of any array in your code, append `.length` to the end of the name of the array. For example the value of `list.length` would be 5 or `randomNos.length` would be the value of N. In the former case it is best to use `list.length` instead of typing 5 explicitly because it makes the code more general and requires less alteration if the length of the array were altered in a second version of the program.

Exercise 4.1 (5 marks)

The aim of this exercise is to practice using arrays in loops and accessing individual array elements.

Write a program, 'Arrays', that creates an array of length N, where N is an `int` you declare and set in the program. You will now fill the array with the first N numbers of the Fibonacci sequence. This is a sequence which crops up frequently in nature. The first two numbers are 1, then the rule to compute the rest is that each number is the sum of the two previous, i.e.

$$x_1 = 1, \qquad x_2 = 1, \qquad x_n = x_{n-1} + x_{n-2} \tag{1}$$

The resulting sequence is $1, 1, 2, 3, 5, 8, 13, \ldots$.

At the END of your program, print out the first ten numbers of the sequence using a loop. Separately it should also print out the 30th and 46th numbers in the sequence. A print-out of your program and sample output should go in your lab notebook.

What if you want to create a new array which contains the same values as an existing array? You may be tempted to type `double[] b = a` where a is the array to be copied to the new array b. This is valid code, but it has a slightly different effect to the one desired. In order to create a copy of an array you should use a `for` loop and set each value individually:

```
// create array b of same length as array a
double[] b = new double[a.length];

// set each value of b equal to equivalent value of a
for (int i=0; i<a.length; i++)
{
    b[i] = a[i];
}
```

Exercise 4.2 (5 marks)

Copy the java program `ArrayCopyDemo.java` from the course web site[8]. Read the code and try to predict what it will do. When you have decided, compile it and run it. Comment on the different effects of the two copying methods and explain what has happened.

For a usual variables, `double b = a`, copies the value of `a` to `b`. With arrays, this is not the case. Instead, `b` is set to to refer to the same data that `a` refers to. There is therefore still only one copy of the data so any change to the values of `b` is a change to the values of `a`. This is due to arrays being *reference data types*, that is they are handled *by reference*, unlike `int`, `double`, `boolean` etc. which are *primitive data types*, handled *by value*. For example in `int[] list = {1, 4, 3}` the array {1, 4, 3} is stored somewhere in the computer's memory, and `list` is a variable name that *refers* to it, but we might set other variable names to refer to this same area of memory using the `=` operator. For now it is hard to see the advantage of arrays behaving in this way, but some should become clear in later sections.

[8] `http://www.pp.rhul.ac.uk/~george/PH2150/downloads/ArrayCopyDemo.java`

Note that anything created with a `new` command, not just arrays, is a *reference data type*.

Figure 4.2 illustrates what happens when you assign one array to another.

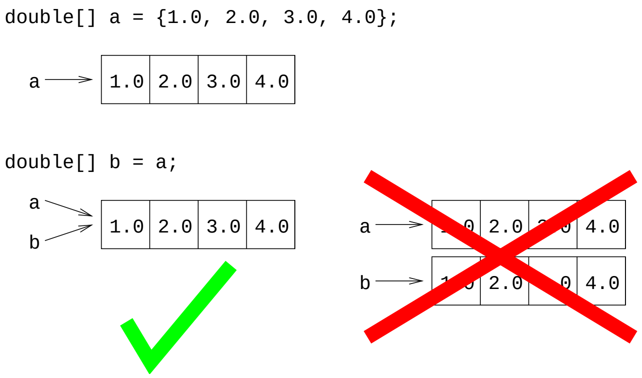


Figure 4.2: *What happens when array* `a` *is defined and then array* `b` *is set equal to* `a`*? Since* `a` *and* `b` *are references, they both point to the same array data (bottom left). It is important to understand that* `b` *does* **not** *get its own copy of the array data (bottom right).*

The only other strange behaviour of *reference data types* to be aware of at this stage is that of the `==` comparison operator.

```
double[] a = {1.0, 2.0, 3.0, 4.0}
double[] b = {1.0, 2.0, 3.0, 4.0}

if (a == b)
{
    // code here not executed
}
```

Rather than comparing the values of `a` and `b`, their references are compared — that is the code asks whether `a` and `b` refer to the same area of memory. So even though they have the same values the test `a == b` will be false and the code within `if` never executed.

Conversely, if the references are both to the same data, then they are considered to be equal so the comparison will be true:

```
double[] a = {1.0, 2.0, 3.0, 4.0}
double[] b = a

if (a == b)
{
    // code here always executed, a==b true!
    // both references refer to same array
}
```


Composite operators

As a short diversion from arrays, you will by now have been using the arithmetic operators `+`, `-`, `*`, `/` and `%` enough to appreciate the usefulness of the *composite operators*. These are `+=`, `-=`, `*=`, `/=` and `%=`, and their uses are:

```
x += y          // shorthand for x = x + y
x -= y          // shorthand for x = x - y
x *= y          // shorthand for x = x * y
x /= y          // shorthand for x = x / y
x %= y          // shorthand for x = x % y
```

Multi-dimensional arrays and nested loops

It is also possible to create arrays with two or more dimensions. Two dimensional arrays are useful for storing matrices for example. In Java, arrays of more than one dimension are created by making arrays of arrays. This is known as *nesting*. The follow examples should make this more clear.

```
// two ways of creating two dimensional arrays
int[][] matrix = {{0, 1, 2, 3},
                  {4, 5, 6, 7},
                  {8, 9, 10, 11}};

int[][] square = new int[5][5];    // creates 5 by 5 matrix with zero values
```

Taking the above 2D array `matrix` as an example, in order to perform operations on each value of a multi-dimensional array, it is necessary to nest several `for` loops. The example code below sets every entry of `matrix` to 1.

```
for (int i=0; i<3; i++)
{
    for (int j=0; j<4; j++)
    {
        matrix[i][j] = 1;
    }
}
```

You can think of the outer loop (`i`) moving down the rows of `matrix`, then the inner loop (`j`) moves along each row. Note that when you nest loops, the loop variable of each loop (`i` and `j` in the example above) must be different for this to work properly.

Particularly with two dimensional arrays such as matrices, it can be helpful to print the entries to screen in columns and lines as they would normally appear in a matrix. Firstly creative use of System.out.println() and System.out.print() (the latter does not move to the next line after printing the output) can help. Secondly there are some special characters:

"\t", which produces a tab and "\n" which moves to the next line. These can be included anywhere in a string.

Exercise 4.3 ——— (10 marks)

Write a program that will multiply together two 3×3 matrices (as defined below) with integer entries and print the resulting 3×3 matrix to the screen, making use of nested `for` loops, and the `+=` composite operator. Use `"\t"` to help format the output. Include the program listing and output in your lab book. You can check the answer by hand or ask a demonstrator to check that it is correct.

```
int[][] matrixA = { { 1, 0, 1 },
                    { 1, 2, 3 },
                    { 1, 4, 5 } };

int[][] matrixB = { { 5, 4, 0 },
                    { 4, 8, 1 },
                    { 1, 1, 0 } };
```

Hint: carefully analyse how matrix multiplication works before you write your program. Show this analysis in your lab book to justify your program design.

5 More advanced input and output

Formatting output

Sometimes you will want to control the way in which the output of a program is written. For example, for some numbers not all the significant figures will be required, or in fact they may get in the way if you wish to write more than one result on each line. It can be helpful then to able to *format* output.

There are three steps to printing formatted output. The first is to create the format[9] that you wish to use. This is done using

```
DecimalFormat myFormat = new DecimalFormat("###.000");
```

where `myFormat` is what you decide to call the format. The format is defined using the string `"###.000"` where `#` denotes a digit and `0` denotes a digit or a zero (if there is not a digit to go in this place a zero is printed instead). The decimal point has its usual meaning. E.g. the number 23 in this format would be printed as `23.000`.

For comparison, if the format was `"000.000"`, the output would be `023.000`. This shows the difference between `#` and `0`.

Note that to use `DecimalFormat`, you will need to add `import java.text.DecimalFormat;` at the top of your program, so that the compiler understands what you mean by it. If you forget this, you will get a compiler error "cannot resolve symbol".

Once a format has been created in this way it may be used again and again for different numbers being printed out. To output a number, here 23, in the format `myFormat` use the code:

```
// creates a string (called sOutput),
//  which is the number in the desired format
String sOutput = myFormat.format(23);

// print out as usual
System.out.println(sOutput);
```

The formats created may be used on all number variable types in exactly the same way. Despite the name `DecimalFormat` the value may be an integer, as demonstrated above, or formats need not include a decimal place, thus printing numbers truncated to integer form.

As well as `#`'s and `0`'s an `E` may be used to specify the exponential from of a number. For example `"0.##E0"` would give a number in scientific format, with a maximum of two decimal places — any extra digits being rounded off.

```
DecimalFormat mySciForm = new DecimalFormat("0.##E0");
String sOutput = mySciForm.format(157.98642);
System.out.println(sOutput);
```

[9] In the example, `myFormat` is actually an *object* of *type* `DecimalFormat`, but *objects* will not be covered until section 8, so for the moment you can assume that it works like a variable which, rather than a number, stores something more abstract which describes how to format numbers.

The above code outputs 157.98642 in scientific format with two decimal places. The output from the program is therefore `1.58e2`

As a shortcut, there is no need to create the intermediate `String`. The following code does exactly the same as the previous example.

```
DecimalFormat mySciForm = new DecimalFormat("0.##E0");
System.out.println(mySciForm.format(157.98642);
```

Exercise 5.1 (5 marks)

What would the following print out? Work it out without using the computer and write the answer in your lab book. Then write a program to check your answers.

```
DecimalFormat fmt1 = new DecimalFormat("0.000E0");
DecimalFormat fmt2 = new DecimalFormat("0.0##");
System.out.println ( fmt1.format(0.1) );          // (a)
System.out.println ( fmt1.format(3.14159) );      // (b)
System.out.println ( fmt2.format(1.5) );          // (c)
System.out.println ( fmt2.format(0.0009) );       // (d)
System.out.println ( fmt2.format(2.7e3) );        // (e)
```

Note: your test program will need to `import java.text.DecimalFormat`.

More detailed information on formatting numbers can be found in the online Java API documentation [4].

Input from a file

As with reading input from the keyboard, you will need to include some particular code near the beginning of your source file in order to use the *methods* for reading input from a file.

```
import java.io.*;

public class InputDemo
{
    public static void main(String[] args) throws IOException,
                                                  FileNotFoundException
    {
```

This is the same as for keyboard input but with the addition of `FileNotFoundException`. This is the *exception* that will be written to your screen if the input file you specify does not exist.

To open your file (the example here is named 'file1.in') from which the input should be read:

```
// Open file to read input from
```



```
BufferedReader br = new BufferedReader(
                     new InputStreamReader(
                      new FileInputStream("file1.in")));
```

As with keyboard input you are not yet expected to understand how this code works, it is enough to be able to copy and make use of it. The indentation is not necessary but again just makes things clearer. This code need only be included once in your program. Once the file is open, a line of the file is read using:

```
temp = br.readLine();  // where temp is a String you should declare first
```

Every time you use this command the next unread line of the file will be placed into the string `temp`. As with keyboard input, one of `Double.parseDouble(temp)`, `Int.parseInt(temp)`, etc. must be used to convert the string `temp` into the number type if required.

To read a file, line by line, until the end of the file is reached, you can do something like this (assuming temp and b are declared as above):

```
while ((temp = br.readLine()) != null)
{
    System.out.println("read line: " + temp);
}
```

Output to a file

Printing output to a file is quite similar to getting input from a file, although the file need not already exist. At the beginning of your source file `import java.io.*` and `throws IOException` are needed.

To open the file, here called 'file2.out', to write output to:

```
// Open file to print output to
PrintWriter q = new PrintWriter(
             new FileOutputStream("file2.out"), true);
```

This line is only needed once when you first want to output to the file. The file will be closed (with the contents saved) automatically when the program ends. Don't forget the `true` when creating the `PrintWriter`, or this will not be the case! If the output file already exists when you run the program, e.g. from the last time you ran it, it will be overwritten and lost, so copy or rename files containing output you want to keep, before re-running a program.

Then whenever you want to write data to the file:

```
q.println("output to be written to file");
```

The above line can be treated in the same way as System.out.println(), so you can output variable values etc. as usual. With `println()` data will be written to a new line each time.

Exercise 5.2 ______________________________ (6 marks)

Using the techniques described in this section:

(i) Write a program that will read in numbers from a given file and print them on the screen. Use the file input.txt as input which you can download from `http://www.pp.rhul.ac.uk/~george/PH2150/downloads/input.txt`.
Include your program output in your lab book.

(ii) Modify the program to write the numbers out to a different file, in scientific format with 6 significant figures, e.g. 3.2 is written as 3.20000E00, 0.5 is written as 5.00000E-01. Include the program listing and a printout of the output file in your note book.

Don't forget to write the following at the top of your program:

```
import java.io.*;
import java.text.DecimalFormat;
```

---

6 Multiple methods

Basics of using another method

So far all the source files you have written have had the same basic structure. Firstly a `public class` for which you choose the name, the contents of which are inside a set of curly brackets. Secondly, inside the class, a `public static void` *method* which must be called `main`, and must have the *argument* `String[] argv`. Again the contents of the `main` *method* are contained within a set of curly brackets `{}`.

```
/**
 * Purpose: To show most simple program structure.
 */

public class Simple
{
    public static void main(String[] argv)
    {
        //some program code goes here
    }
}
```

We are now going to add to this program structure a second *method*. See that this second *method* is still inside the `class`, but unlike all the other code you have made use of so far, it is outside the `main` *method*.

```
/**
 * Purpose: To show structure of program with multiple methods.
 */

public class LessSimple
{
    // another method - can be called what you like
    public static void doSomething()
    {
        //code may go in here in the usual way - variables, ifs, loops etc.
        //must end with a return statement...
        return;
    }

    // main method as usual
    public static void main(String[] argv)
    {
        //some program code goes here
        //should use the doSomething() method
        //this is done by typing
        doSomething();
    }
}
```

The lines `doSomething()` and `return` will be explained shortly.

When the program is run, the `main` *method* is executed. Any other methods that exist will only be executed if they are used by the `main` *method*. The `main` *method* can be thought of as the top level of the program, from which other methods are invoked. It is good practice to keep `main` very short and simple, by splitting your program up into different methods which can be called from `main`.

The convention is to have the `main` *method* written as the last *method* in your source file, just so that it is easy to find. The order in which *methods* are written doesn't affect the order in which they are executed — even though it is at the end, the computer always begins by executing the `main` *method*.

Note that the additional *method* is *declared* using `public static void` just as for the `main` *method*.

Here is a simple example with exactly the same structure as above.

```
/**
 * Purpose: Prints the first 10 square integers,
 *          as an example to show the use of two methods.
 */

public class Trivial
{
   // method to print squares from one to a hundred
   public static void squares()
   {
       for(int i = 1; i < 11; i++)
       {
           System.out.println(i*i);
       }
       return;
   }

   // main method
   public static void main(String[] argv)
   {
       squares();      // calls the squares method
   }

}
```

The line of code `squares();` is where the `main` *method* makes use of the `squares` *method*, asking it to carry out its set of instructions. This is known as a *call* to the `squares` *method*. The `return` statement at the end of `squares` sends flow of the program back to the *method* that called it, and so execution of the `main` *method* continues.

You may have found that writing programs where all your code is within the `main` *method* can become very long, which makes the source file look messy or unclear. It can also be awkward if you have similar groups of calculations or instructions that need to be performed and need to be typed in separately each time. Using additional *methods* can help with both of these issues, making code tidier and therefore easier to understand, and reducing the need to duplicate code. There is little obvious advantage in having separated the above program into two *methods* — the same could very easily be achieved with just a `main` *method* — it is just a simple example in how to write and call *methods*. However with larger programs there are significant benefits of separating your code into several *methods* in this way.

Passing information between methods

What if you want the behaviour of a *method* to vary depending on a value of a certain variable in your `main` *method*? It is possible to *pass* several values to *method*, and receive up to one value back. You may for example have several numbers you would like some calculation performed on. You could pass these values to a *method* and then receive the result back.

Let's start with how to give, or *pass*, a value **to** a *method*. This is done by giving the *method* an *argument*, which goes between the round brackets `()` after the *method* name. Look at the following example, an explanation follows.

```
/**
 * Purpose: Example of program with two methods.
 *          The main method uses the printSquare method.
 *          The printSquare takes an double argument and prints the
 *          value of the argument squared to the screen.
 */

public class ArgExample
{
    // method to print square of a value to screen
    public static void printSquare(double y)
    {
        System.out.println(y*y);
        return;
    }

    // main method
    public static void main(String[] argv)
    {
        double x = 5.0;

        printSquare(3.0);  // call to print the square of 3.0
        printSquare(x);    // call to print the square of x
    }
}
```

The *method* `printSquare` is *declared* as `public static void printSquare(double y)`. Unlike before, the brackets that follow the *method* name now contain a variable declaration, in this case for a variable `y` which is of *type* `double`. The variable `y` exists within this method and can be used here for calculations etc..

Because the variable declaration for `y` is in the round brackets, whenever the *method* `printSquare` is called the *method* expects a `double` value to be in the round brackets in the call statement. So the call for `printSquare` may look like:

```
printSquare(3.0);
```

In the `printSquare` *method* `y` then takes this value, for the case above 3.0, so the value 9.0 is printed to the screen.

Within the `main` *method* a variable `x` is declared and given a value. When the call `printSquare(x)` is used, `y` takes whatever value `x` has in the `main` *method* at the time, in this case 5.0, so 25.0 is printed to the screen.

It is possible for a *method* to have several arguments. These should be separated by commas, e.g.

```
public static void calculate(double x, double y, int i, short n)
{
    //method code goes here
    return;
}
```

then to call the *method* simply enter values, or variables, of the expected *type*:

```
calculate(3.0, result, 2, num) // where result and num are
                               // double and short variables respectively
```

Exercise 6.1 ——— (6 marks)

The purpose of this exercise is to write your own class with multiple methods.

- Write a class called `MultMethEx1` which has a *method* called `printTriangleArea` to calculate the area of a triangle, given the lengths of the three sides as arguments.[10] It should print out the result.
- Write a `main` *method* which asks the user to enter values for the sides of a triangle and then uses the `printTriangleArea` *method* to calculate and print the area. **Do not** write code to check the input yet.
- Include the program listing and sample output in your lab book.
- Bear in mind the constraints on the sides of the triangle[11] when testing the program.

So far all the additional *methods* you have seen have been declared as `void` and have not passed back, or *returned*, any values to the *method* that called them. This is the meaning of `void`, that the calling *method* should not expect any result to be passed back to it from the *method* being called. However *methods* can also be declared as `double`, `int`, `boolean`, `String`, or any other variable *type*, in which case the code for the *method* must return a value of that *type*. This is probably easiest to understand from an example.

```
import java.io.*;
public class NonVoidMethodExample
{
    public static double function1(double x)
```

[10] The area of a triangle with sides a, b, c is given by Heron's formula:

$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

where $s = (a+b+c)/2$.

[11] In order for a, b and c to form a triangle, two conditions must be satisfied: all side lengths must be positive; the sum of any two side lengths must be greater than the third side length.


```
    {
        /* write some mathematical function of x here, e.g. x + 3x^2 */
        return (x + 3*x*x);
    }

    // and then to call the above method

    public static void main(String[] args) throws IOException
    {
        BufferedReader br = new BufferedReader(
                                new InputStreamReader(System.in));
        System.out.print("Enter a value of x at which to evaluate f(x) ");
        System.out.print("or return to end: ");
        // read input until an empty line is returned
        String nextLine;
        while (( nextLine = br.readLine()) != null)
        {
            if (nextLine.equals(""))
            {
                break;
            }
            double x = Double.parseDouble(nextLine); // JDK 1.3.1
            System.out.println("f(x) at x=" + x + " is " + function1(x));
            System.out.print("Enter another value of x, or return to end: ");
        }
    }
}
```

The above example defines a *method* called `function1` which has the return *type* `double` and also takes a `double` argument. The method simply performs a calculation using the argument and returns the result.

The use of this *method* is demonstrated in the `main` *method*. In this example, the user is prompted to enter a number, then there is a loop which takes each entered number, uses the function1 *method* to calculate another number, and prints them. The program ends when the user enters nothing.

Notice that the call to a `void` *method* exists on its own on a line. In contrast, in the example above, there is a value returned from the *method*, so this needs to be used somehow in a calculation or stored in a variable.

Other information about methods

A *method* can not alter the values of any of its arguments. Thus the following is not valid code and will not compile.

```
public static int increment(int i)
{
    i += 3;           // this is not allowed, code will not compile
    return i;
}
```

You do not have to restrict yourself to calling additional *methods* just from the `main` *method*. If you have two or more additional *methods* one may call another.

Exercise 6.2 (9 marks)

Starting with the class MultMethEx1 you wrote for the previous exercise, make a new version of the class called MultMethEx2.

(i) Add a new *method* called `calcTriangleArea` that performs the same calculation as `printTriangleArea` did, but instead of printing the area to the screen, it returns the area as a `double`. Modify `main` to use both methods and show that they produce the same result.

(ii) Change the code in the *method* `printTriangleArea` so that it uses the new *method* `calcTriangleArea` instead of calculating the area itself. No changes to `main` should be necessary. Demonstrate that your program produces the same result.

(iii) Add a *method* `testIfValidTriangle` which takes the three sides as arguments and checks that they form a valid triangle, according to the criteria given in the previous exercise. This *method* should return a boolean value: *true* if the sides do make a valid triangle, or *false* if not. Use the `testIfValidTriangle` *method* in the `main` *method* to validate the input before trying to calculate the area.

Note that in answering question (ii) you have benefited from having *encapsulated* a well-defined part of your program inside a separate *method* (`printTriangleArea`). This allows you to modify the internal workings of this *method* without requiring any changes elsewhere in the program where the *method* is used.

Include the program listings and sample outputs from all the questions above in your lab book.

Method naming

Notice from the above examples that method names follow several conventions.

- Method names are often verbs because methods should do something.
- A method should do what its name says - no more or less!
- Start names with a lower case letter, just like variables.
- Subsequent words in the name start with a capital letter, just like variables

Class variables, method variables and scope

Look again at the example that was used earlier, reproduced below. There are two variables `x` and `y`.


```
/**
 * Purpose: Example of program with two methods where one method
 *          takes an argument.  The program prints the value of
 *          x squared to the screen.
 */
public class ArgExample
{
    // method to print square of a value to screen
    public static void printSquare(double y)
    {
        System.out.println(y*y);
        return;
    }

    // main method
    public static void main(String[] argv)
    {
        double x = 5.0;

        printSquare(3.0);   // prints the square of 3.0 to the screen
        printSquare(x);     // prints the square of x to the screen
    }
}
```

Variable `y`, which is declared at the same time as the `printSquare` *method*, only exists within this *method*. Each time the *method* is called, the variable is recreated, and then ceases to exist when flow returns to the `main` *method*. Code outside of the `printSquare` *method* cannot use `y`. That is, the *scope* of `y` is the *method* `printSquare`. The concept of scope was introduced in section 3, but now it has to be extended to include methods and classes.

Similarly `x` is defined inside the `main` *method*, and may only be used in there. The *scope* of `x` is the `main` *method*. So `x` and `y` exist independently of each other, within their respective *methods*.[12] They are known as *method variables*. It is not possible for `printSquare` to use the variable `x`, or for the `main` *method* to use `y`, because the variables do not exist there.

Before this section on multiple methods, all the variables you have used have either been declared in the `main` *method* (in which case their scope is from their declaration to the end of the *method*), or within a `for` loop, `if` statement or similar block of code (in which case their scope was from declaration to the end of the code block, denoted by the closing curly bracket `}`). Now you have also created variables in other *methods*, but the principle is just the same.

There is one further level at which variables can be declared. These are *class variables*. They are variables declared outside of any *methods*, but within the class, and as such exist for all *methods* in the class. Their basic declaration is the same as with any variable, however like *methods* they should have `public static` added to the beginning. Here is a simple program which does something similar to the previous example, but where the scope of `x` is different.

```
/**
 * Purpose: Example of program with a class variable
 *          The program prints the value of x squared
 *          to the screen.
 */
```

[12] Because the variables exist independently in their separate regions, there is no reason why they should not be given the same variable name if you wish. Even if both were called `x` the two variables would exist completely independently in their respective scopes.

```
public class ScopeExample
{
    // declare class variable x
    public static double x;

    // method to print x squared to screen
    public static void printXSquare()
    {
        System.out.println(x*x);
        return;
    }

    // main method
    public static void main(String[] argv)
    {
        // give x a value and print its square to screen
        x = 5.0;
        printXSquare();

        // change value of x and print square to screen
        x = 2.0;
        printXSquare();
    }

}
```

Since `x` exists within the entire class, both *methods* can make use of it.

There is no need to pass the value of x to the *method* `printXSquare`, so there is no need for the *method* to take an argument. Note that because `printXSquare` has been written to print the value of x^2 to the screen, it can not be used to print the square of any other value. It is not in general a very useful *method*.

To summarise these ideas of variable scope:

Class Variables An example is the variable `a` in the code below. For now you should always declare using `public static`. The variable is declared in the class, but outside of the *methods*. It is visible to all the the class, so all the *methods* of the class may use it.

Method Variables Examples are `b` and `c` in the code below. *Method* variables are declared inside a *method* (`c`), or as an argument in a *method* declaration (`b`). The scope of `c` is from its declaration to the end of the *method*. The scope of `b` is the entire *method*.

Variables declared in a block of code E.g. the variables `d` and `e` in the code below. They are normally declared within a block of code (code within a pair of curly brackets `{}`) and their scope is from declaration to the end of that block of code. In the case of a `for` loop the declaration in in the round brackets following `for` and the variable exists for the duration of the loop.

```
/**
 * Purpose:  Program that does nothing useful, but using variables
 *           that have a variety of scopes.
 */

public class ScopeSummary
```



```
{
    // scope of a is the entire class
    public static double a;

    // scope of b is all of method alpha
    public static double alpha(int b)
    {
        double c;              // scope of c from here to method end
        c = a*b;
        return c;
    }

    public static void main(String[] argv)
    {
        // scope of d is all of for loop
        for (int d=1; d<100; d*=2)
        {
            if ((d%3) != 0)
            {
                a = d%3.0 + 1.234;
                double e = alpha(d);    // scope of e starts here
                System.out.println(e);
            }                           // scope of e ends here
        }
    }
}
```

Exercise 6.3 ________________________________ (4 marks)

With reference to the example code above, categorise the following statements as true or false.

(i) The variable `a` is visible in any method of the class.

(ii) The variable `d` is visible inside the method `alpha`.

(iii) The variable `c` is not visible inside the method `main`.

(iv) The variable `e` is visible anywhere inside the method `main`.

Passing arrays to methods

You will remember from the section about arrays, that arrays are *reference variable types* and as such their behaviour is slightly different from that of variables of *primitive data types* such as `double`, `int`, etc. Passing arrays to or from *methods* can be very useful. However it is the `main` area where *reference data types* behave slightly differently from *primitives*.

To pass an array to a *method*, the *method* declaration should look like this:

```
public static void takeArrayMethod(double[] numberList)
{
    // method code
    return;
}
```

and then the *method* call might look like:

```
takeArrayMethod(vector1);                    //or
takeArrayMethod({1.0, 3.4, 5.2});
```

where `vector1` would be an array of double values already created by the program.

For a *method* to return an array the *method* should be declared as you might expect:

```
public static int[] returnArrayMethod()
{
    // method code which includes creating an int array called vector2
    return vector2;
}
```

When a *primitive data type*, e.g. `double`, is passed to a *method*, its value is copied to the new *method* variable. For *reference data types*, a new *reference* is created, but unlike for *primitives*, the data that is referenced is **not** copied to a new area of memory. Instead the

new *reference* is set to refer to the original area of memory storing the data. This is known as *passing by reference*.

As was mentioned earlier it is not possible for a *method* to change the value of its arguments. E.g.

```
// method that tries to change argument value
public static double increase(double x)
{
    x += 10;                            // code not allowed so
    return x;                           //    will not compile
}
```

The equivalent for an array is that the variable (i.e. the array name) can not have its reference changed. E.g.

```
// method that tries to change argument reference
public static int[] change(int[] vector2)
{
    vector3 = {1,2,3};
    vector2 = vector3                   // code not allowed so
    return vector3;                     //    will not compile
}
```

However it is possible to change individual values in the array, in the area of memory that `vector2` refers to. This is in contrast to primitive data *types* where the value of an argument can not be changed. E.g.

```
// method that changes the values of an array
public static void valueChange(int[] vector2)
{
    for(int i=0; i<vector2.length; i++)
    {
        vector2[i] = i+1;               // this IS allowed
    }
    return;
}
```

Note that in this particular case, since any value changes are made to data in the original area of memory, there is no need to have `return vector2;` thus the *method* is declared as `void`.

Exercise 6.4 ________________________________ (5 marks)

It is useful to have generic methods to perform common tasks that you will use again and again, saving yourself a lot of typing. It also makes your code more concise, abstract and easy to understand. One common programming task is to print the contents of an array. In a large program with arrays this might have to be done many times. For this exercise you will write a method to do this and use it to simplify some code you wrote in a previous exercise.

- Take your answer to exercise 4.2, and rename it.
- Add a separate method to print out any integer array.
- Change the main method to use this utility in place of any existing loops that print the contents of an array.
- Put the modified program listing in your lab book along with sample output.

If you need to print arrays in any future exercises, you should try to re-use this method.

---

7 Numerical problems and multiple classes

Numerical integration

Numerical methods are a collection of techniques for solving mathematical problems which lend themselves to implementation as computer programs. They exist to solve all sorts of problems, for example differentiation, minimisation, sorting data and solving differential equations. Numerical methods are particularly useful when it is either impossible or impractical to solve a problem analytically.

The trapezium rule for numerical integration of a function $f(x)$ is

$$\int_{x_1}^{x_N} f(x)dx = h[\frac{1}{2}f_1 + f_2 + f_3 + ... + f_{N-1} + \frac{1}{2}f_N] + O(\frac{(b-a)^3 f''}{N^2})$$
$$= h[\frac{1}{2}(f_1 + f_N) + \sum_{i=2}^{N-1} f_i]$$

where $N-1$ is the number of trapezia, x_1 and x_N are the lower limit and upper limits of integration respectively, and h is the trapezium width.

Here you will use the trapezium rule to numerically integrate a function. This exercise will allow you to put into practice many aspects of Java programming that you have learnt so far. A reminder/primer on the trapezium rule can be found in Appendix C.

Exercise 7.1 (20 marks)

Write a program that will make use of the trapezium rule to numerically integrate a mathematical function.

The following steps are given as a guide on how to proceed with the program. It is recommended that you follow these steps closely to get all the available marks.

1. Write a separate method to return the value of the function you want to integrate. For example, this is a definition of $f(x) = 2x$:

   ```
   public static double f(double x)
   {
       return 2*x;
   }
   ```

2. Write a method which implements the trapezium rule. It should take as arguments the range over which to integrate (x_1 and x_N) and the number of steps (N). It should use these parameters to compute the integration using the trapezium rule and return the result. It will call the method `f` you defined in the previous step.

3. Write a `main` method to prompt the user to type in values of x_1, x_N, and N, call the above method and print the result. It should check the value of N supplied is valid before proceeding to any calculation, but it is sufficient just to exit with a auitable error message if the value is invalid.

4. Test your program, using a function linear in x as shown above, first with $N = 1$ (which should fail the validity check) then $N = 2$ and a few values where $N > 2$. Calculate the answer by integrating the function by hand. You should get exactly the right answer regardless of the value of N (why?)
5. Next, test your program with a second order polynomial for several values of N. Again, compare with the analytical (hand calculated) result.
6. Study what happens to the accuracy of the answer as N increases. Show how the convergence relates to N. Is this what you expect from the formula for the trapezium rule?
7. Based on your findings, devise a method to estimate the accuracy of a calculation when the correct answer is unknown.

At this point you should have established enough confidence in your program to use it on a function which cannot be integrated analytically.

When you have written your program, show a demonstrator that it works. You will then be allocated a function to integrate from those below. It will help to sketch the function before you try to integrate it.

(i) The function $x^4 \log(x + \sqrt{x^2 + 1})$ over the interval $x = 0, 1$.

(ii) The function $\log(3x^2)$ over the interval $x = 0.7, 2$.

(iii) The function $\cos(x^2)$ over the interval $x = -1, 1$.

(iv) The function $\cos(x^3)$ over the interval $x = -1, 1$.

(v) The function $\sin(x^2)$ over the interval $x = 0, 1.7$.

(vi) The function $\sin(x^3)$ over the interval $x = 0, 1.4$.

(vii) The function $-\log(\sin(x))$ over the interval $x = 0.1, 1$.

(viii) The function $\cos(\tan(x))$ over the interval $x = -1, 1$.

Make sure you document the **design**, **code** and **testing** of your program as well as the final result. Most of the marks for this exercise will be awarded for these three things. As well as giving the answer, **estimate the accuracy of this number using the method you devised in step 7 above**. An accuracy of around 3 significant figures is sufficient.

Multiple classes

You've now added extra methods to the basic program structure, but it is also possible to add extra *classes* too. In the basic program structure you had one *class*, which was declared `public`. Since a `public class` must always have the same name as the file in which it is contained, if your program is to include an extra `public class`, it should be in a separate file of the same name. Additional *classes* **do not** contain a `main` method.

One advantage of using additional *classes*, is if you've written some *methods* for use in one program that you think may be useful in another. By writing these *methods* in a separate `public class` (in a separate file) any program may make use of them in exactly the same way as usual just by adding the name of the `class` containing the *method*, to the beginning of the *method* name, i.e.

```
ClassName.methodName(argument);
```

Here's a short example, based on the `printSquare` example from section 6, page 39. The following code should be in a file called 'MainClass.java'.

```
/**
 * Demonstrate the use of the printSquare method in ExtraClass.
 */
public class MainClass
{
    public static void main (String[] args)
    {
        double x = 3.0;
        ExtraClass.printSquare(x); // call to print the square of x
    }
}
```

The above code uses the method `printSquare` which is in this file, 'ExtraClass.java'.

```
/**
 * Utility class with some general methods
 * which can be used by other classes
*/
public class ExtraClass
{
    // method to print square of a value to screen
    public static void printSquare(double y)
    {
        System.out.println(y*y);
        return;
    }
}
```

Both these files can be downloaded from the course web page.

Splitting code up like this means that any other program you write may also make use of the method `printSquare` without you needing to retype it. Obviously if you are going to do this, it pays to make your code as general as possible and to comment it as fully as possible, to avoid problems if you end up re-using it some time after it was originally written.

When compiling the file containing the `main` method, the compiler will automatically detect if any other files/*classes* are used by the program and compile these too if necessary.

Splitting up programs over several files also makes it easier for more than one person to work on a project at the same time — useful if you are programming in a team. One person can be busy writing a `main` method, and as long as he/she knows what the *class*, *method* names and their *arguments* are, they can make use of them in the `main` method. Another programmer can worry about the details of how to perform the necessary calculations etc. within these additional *methods*.

You've probably noticed by now that the form of the call to a *method* of another *class* looks quite familiar. This is because you have been using it already, e.g. `System.out.println()` and the mathematical functions such as `Math.sin(x)`. These *classes* and their *methods* are a standard part of the language and are automatically available for you to call in any Java program.

8 Object oriented programming

Introduction to object orientation (OO)

By now you have mastered all the basics that enable you to write programs that will perform all the calculations you need. However we have not yet touched upon the style of programming that Java was really designed for, *object orientation*, or *OO*. There can be great advantages in writing your programs in an *object oriented* way. This section should serve as an introduction to *OO* programming in Java. If after this you wish to find out more, try chapter 6 of [1] and chapter 2 of [2].

This section will require you to start thinking about classes in a different way. However, despite the different style, you will still need the Java covered previously in the course. Also bear in mind that, however a program is written, when running the program the computer always starts with the one and only `main` method.

Creating an object

Objects are not dissimilar to variables or arrays in some ways. Like arrays they are *reference data types*, and so may be passed to and from *methods* in the same ways as arrays. The major difference is that the programmer can define the form that the object takes — such as how many values it has and methods that are particular to objects of that *type*[13]. To see how this works, read carefully though the following example.

Before creating any objects, we must write a sort of blueprint or template for them. This 'blueprint' is the *class*. Previously you have been using classes really only as a way of grouping together methods in a program. The classes that act as 'templates' for objects are behaving differently from the ones you have used before because we now begin to omit the word `static` in class variable and/or method declarations.[14]

It is possible to imagine many scenarios when you might want to perform calculations using complex numbers. It would be useful then to create objects that behave in the ways that complex numbers do. Here is a class `Complex` that defines a complex number.

```
/**
 * Purpose: A class from which objects can be created, which behave
 *          like complex numbers.
 *          NB: This class should be saved in the file Complex.java
 */
public class Complex
{
    // two non-static class variables
    // representing the real and imaginary parts of a complex number
    public double realpart;
    public double imagpart;
}
```

[13] The word *type* is used here in the same way as a variable may be of *type* `int` for example.

[14] The `main` method for the program should still always be declared `public static void` and should be in a separate `public class` that does **not** contain **any** non-`static` *methods* or *class variables*.

Then in any program where you wish to use complex numbers you can create an *object* of this *class*. The *object*, just like a variable, can be given a name of your choosing. To create an *object* of the *class* `Complex` write, in the method of another class:

```
Complex a;           // creates a reference "a"  (c.f. arrays)
a = new Complex();   // creates a new object of class Complex
                     //     and sets "a" to refer to it

// often this is all written on one line
Complex b = new Complex();
```

Each object created from this class will have its own `realpart` and `imagpart`, and you can refer to these in your program by adding `.realpart` or `.imagpart` to the end of the *object* name. Thus a program that created a `Complex` *object*, gave it values and printed them to the screen would look like this.

```
/**
 * Purpose: Demonstrating how to create an object.
 */

public class ObjDemo
{
    public static void main(String[] args)
    {
        // create an object of type Complex
        Complex a = new Complex();

        // give object values
        a.realpart = 1.2;
        a.imagpart = -5.9;

        // print to screen
        System.out.println(a.realpart + " + " + a.imagpart + "i");
    }
}
```

As with any programs that involve more than one *class* (and therefore more than one file), compiling the file that contains the `main` method will automatically compile any other files needed for the program to work.

Exercise 8.1 ——— (5 marks)

Type in the Complex class given above and save the file as 'Complex.java'. Write your own short program (`main` method only) similar to 'ObjDemo' above that creates two Complex objects and gives them values, then adds them together by refering to the real and imaginary parts as `a.realpart` and `a.imagpart`, and prints out the result. Put the program listing of ObjDemo and output showing your tests in you lab book.

This might seem quite a long-winded way of adding two complex numbers, and indeed it is. Fortunately, using *objects* there is a simple way to do this that is considerably more versatile.

Non-static methods

The idea of a *method* is that it gives an object some capability which it ought to have. Complex numbers should be capable of complex arithmetic, such as addition, subtraction, multiplication and so on. For example, complex numbers should be able to add themselves to other complex numbers, so it is reasonable to write a *method* to do this.

Below, a simple method for complex number addition has been added to the `Complex` *class* used above.

```
public class Complex
{
    public double realpart;
    public double imagpart;

    // method to add together two complex numbers & return the result
    public static Complex add(Complex z, Complex w)
    {
        Complex sum = new Complex();
        sum.realpart = z.realpart + w.realpart;
        sum.imagpart = z.imagpart + w.imagpart;
        return sum;
    }
}
```

This first *method* `add` is similar in form to those you are already used to writing. When in a program it is necessary to add two `Complex` *objects* together, the *method call* should look like:

```
    // where a, b and c are each objects of the class Complex
    c = Complex.add(a,b);  // add a and b, result stored in c
                           // (values of a and b remain unaltered)
```

as you may expect from having used methods in other classes in Section 7.

Here's another addition *method* as an example, called `increaseBy`, note that this one is **not** declared as `static`. This *non*-`static` *method* also goes inside the `Complex` class.

```
    // method to add complex "z" onto this object
    public void increaseBy(Complex z)
    {
        realpart += z.realpart;
        imagpart += z.imagpart;
        return;
    }
```

The calling of a *method* which *isn't* declared as `static` is slightly different. Rather than adding the method name to the class name as in `Complex.add(...)` the method name needs to be added onto the end of the name of an object which has been created from that class.

```
a.increaseBy(b);        // add b to a (result stored in a), or
b.increaseBy(a);        // add a to b (result stored in b)
```

This is because `increaseBy` uses `realpart` and `imagpart` which can take different values for each *object* created from the *class*. Thus it is necessary to specify which *object* of the ones you've created it is that you want the *method* to be applied to.

Note that (unlike `add`) the method `increaseBy` *acts on* one of the *objects* and takes the other as an argument. The `realpart` and `imagpart` are the real part and imaginary part of the object on which the method acts. Whereas `z.realpart` and `z.imagpart` are the real and imaginary parts of the object passed as an argument to the method. In the case of the addition method above `a.increaseBy(b)` will leave `a` with the same value as `b.increaseBy(a)` will give to `b`. So using `increaseBy` in this way is like `a=a+b` or `b=a+b` while using the `add` *method* is like `c=a+b`.

Methods like `increaseBy` in the example above are known as *non-static* and defined without the `static` keyword because they act on data which belongs to *objects*. In this case, the data are the variables `realpart` and `imagpart` which clearly have different, independent values in each *object* of type `Complex`. You will see the distinction between *object* and *class data* at the end of this section.

Constructor methods

Constructor methods are a particular type of *non*-`static` method. They are given the same name as the `class` and are automatically called when an *object* is created using `new`. They are declared slightly differently from usual *methods* — just `public`, there is no `void` or `double` etc. and should be the first methods in the class. A *constructor* method is used to automatically *initialise* an *object*'s variables to values of your choice. Here is an example:

```
public Complex(double x, double y)
{
    // initialise class variables to values specified in method call
    realpart = x;
    imagpart = y;
    return;
}
```

Enabling `Complex` *objects* to be created with any initial values like this:

```
Complex z = new Complex(3,4);
```

The `new Complex()` you have used before this is a default *constructor*, which *initialises* all variables to zero. However, if you write your own *constructor* like the one above, the default will not exist. It is possible to write more than one *constructor* for a *class*, if each takes

a different number of *arguments*. Therefore you could add the default *constructor* to your *classes* if you wish — i.e. in addition to the `public Complex(double x, double y)` above, the *class* can also contain:

```
// default constructor
public Complex()
{
    realpart = 0.0;
    imagpart = 0.0;
    return;
}
```

Using private class variables

So far you have been using `public` *class variables*. The word `public` makes them visible to the other classes, so they can be referred to in the `a.realpart` way. This is not a good way to design a class, for the following reason. Say you wanted to change the way the complex number was stored from Cartesian to polar, so you replace the `public` *class variables* `realpart` and `imagpart` with `argument` and `modulus`. Now you have a problem because anywhere in any other program that Complex objects have been used, you will have to change them to use the `argument` and `modulus` instead of `realpart` and `imagpart`.

It would be much better to somehow hide all the internal workings of the `Complex` class, so any alterations to the class did not require changes to be made to any of the other classes using `Complex`.

This is possible with a combination of `private` rather than `public` *class variables*, and *methods* that can be called to find out what these variable values are. *Class variables* that are declared as `private` can not be referred to from other classes, they are only visible within their own *class*. It is considered better programming practice to use `private` rather than `public` *class variables*, and you should aim to do this in the remainder of the course. Here is the `Complex` class rewritten using only `private` *class variables*.

```
/**
 * Purpose: A class from which objects might be created, which behave like
 *          complex numbers.  Private variables are used to demonstrate
 *          good programming practice.
 */

public class Complex
{
    // private class variables
    private double realpart;
    private double imagpart;

    // default constructor, real and imaginary parts are initialised to zero.
    public Complex()
    {
```

```
        realpart = 0;
        imagpart = 0;
        return;
    }

    // another constructor
    public Complex(double x, double y)
    {
        realpart = x;
        imagpart = y;
        return;
    }

    // method to find out value of real part
    public double getReal()
    {
        return realpart;
    }

    // method to find out value of imaginary part
    public double getImag()
    {
        return imagpart;
    }

    // method to add complex "z" onto this object
    public void increaseBy(Complex z)
    {
        realpart += z.getReal();
        imagpart += z.getImag();
        return;
    }

    // method to add together two complex numbers and return the result
    public static Complex add(Complex z, Complex w)
    {
        Complex sum = new Complex();
        sum.realpart = z.getReal() + w.getReal();
        sum.imagpart = z.getImag() + w.getImag();
        return sum;
    }

    // method to print out in usual complex number form
    public void print()
    {
        System.out.print(realpart);
        if (imagpart < 0)
        {
```



```
            System.out.print(" - " + (-1*imagpart) + "i");
        }
        else if (imagpart > 0)
        {
            System.out.print(" + " + imagpart + "i");
        }
        System.out.println("");
        return;
    }

}
```

See how the values of `realpart` and `imagpart` may now be obtained by calling the *methods* `getReal` and `getImag`. Any calculations etc. that might need to be done with complex numbers should be achieved by writing static methods such as `add` and non-static methods like `increaseBy`.

Exercise 8.2 ———— (8 marks)

Copy the file 'Complex.java' from the web site and save it in your current Java project or directory: `http://www.pp.rhul.ac.uk/~george/PH2150/downloads/Complex.java`. Open the file and look at it — it contains the Complex class listed above with `private` class variables discussed above with the constructors, the `increaseBy` method and the `print` method.

(i) Write a program, in a separate class called something like "TestComplex", that makes **use** of the Complex class **without changing it**, to create two Complex objects, u and v, give them values, add v to u using the *non*-`static` `increaseBy` method and print the new value of u to the screen, using the `print` method. When you have checked that it works, use it to calculate the result of $(4 + 3i) + (2 - 7i)$.

(ii) Now adapt 'Complex.java' by adding a new *non*-`static` method that turns a complex number into its complex conjugate. Test it thoroughly by getting your program to check all cases for the original number: imaginary part positive, imaginary part zero and imaginary part negative. Show the results for $(4 + 3i)$, $(2 - 7i)$, and 2.

Printouts of both files and sample outputs should go in your lab notebook.

Discussion of objects and static vs. non-static methods

This section begins with a slight diversion to teach you more about *objects* and *object orientation*.

You have come across *objects* before — the code you were given to read input from a screen, and to write and read data to and from files, involved creating and objects and using non-`static` methods from classes that have been written by someone else. You haven't seen the files containing these classes because they are automatically included as part of the Java language.

Despite never reading these classes, by being told what classes and methods exist you have still been able to make use of them. This is the advantage of *OO*, it is much simpler to use another class like this than try to understand the intricate workings of somebody else's code.

Object Orientation is a powerful technique when it comes to modelling real systems in a computer program. When it comes to designing your own programs using objects, as a general guide, it is a good idea to write a *class* and create *objects* for things are are objects in real life. Think about what you will be asking the objects to do or to tell you; these things can be implemented as their methods.

Take the following example. Say you have a farm and keep sheep. You have several pens in which the sheep are kept so each sheep is assigned a pen number. You may wish to find out which pen a sheep is in, move it to another pen, or count the total number of sheep that you own. To model this using *OO* you can write a *class* called `Sheep` which will make use of both `static` and non-`static` *class variables* and *methods*.

```
/**
 * Purpose: A class from which objects might be created, which behave like
 *          sheep.  Demonstrates use of classes to represent classes
 *          of objects in real life, and the difference between using
 *          or omitting static.
 */

public class Sheep
{
    // each sheep is kept in a pen for which the number is
    private int penNumber;

    // total number of sheep created from this class
    private static int totalSheep = 0;

    // sheep constructor
    public Sheep(int n)
    {
        penNumber = n;
        totalSheep ++;
        return;
    }

    // find which pen a sheep is in
    public int find()
    {
        return penNumber;
    }

    // move sheep to another pen
    public void moveTo(int differentPen)
    {
        penNumber = differentPen;
```



```
        return;
    }

    // count all sheep
    // NB: this is a static method, and it doesn't
    //     use any non-static variables
    public static int countAll()
    {
        return totalSheep;
    }
}
```

This program can be downloaded from the course web page:
`http://www.pp.rhul.ac.uk/~george/PH2150/downloads/Sheep.java`

When in a program you create an *object* of *type* `Sheep`, each is created from the *class* which acts like a template. Each sheep *object* has its own `penNumber`, however because the *class variable* `totalSheep` was declared as `static`, each object does not get its own variable of that name. There is just one variable `totalSheep`, belonging to the *class* `Sheep`, which is shared and used by all of the *objects* created from this *class*.

The same is true for *methods*. Those declared without `static` are specific to each *object* and are used by acting on an *object*. E.g. if there is a sheep called `alfred`, to find out the pen it is in use `alfred.find()`. However the `static` method isn't specific to each *object* so to find out how many sheep there are in total, use `Sheep.countAll()` as a method call.

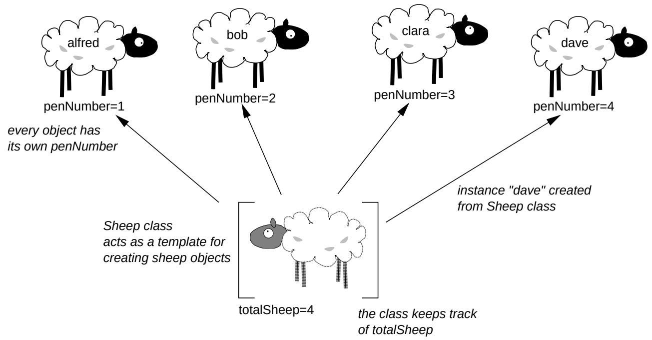


Figure 8.3: *Illustration of the Sheep class in use. Four instances are created from the Sheep class. Each instance has its own* `penNumber`*. The Sheep class also has a static class variable* `totalSheep` *which keeps track of the total number of sheep in the class, not in the individual instances.*

See how adding the word `static` changes the behaviour of a *class*. When an *object* is created from a class, it does not gain its own version of that *method* or *variable*. When all *methods* and *class variables* are declared as `static` creating an *object* from the class would not do anything, so *objects* can not be created from *classes* where everything is `static`. This is how all the programming you were doing before this section on *OO* worked.

Figure 8.3 illustrates this point, in association with an example class SheepTrial which uses the Sheep class:
`http://www.pp.rhul.ac.uk/~george/PH2150/downloads/SheepTrial.java`

```
/**
 * Trial program to demonstrate use of Sheep class
 */
public class SheepTrial
{
    public static void main(String[] args)
    {
        // create Sheep instances
        Sheep alfred = new Sheep(1);
        Sheep bob = new Sheep(2);
        Sheep clara = new Sheep(3);
        Sheep dave = new Sheep(4);

        // check total number of sheep
        // note that this is a class method, not an object method.
        System.out.println("Total number of sheep is " + Sheep.countAll());

        // check which pen dave is in then move him to another pen
        System.out.println("dave is in pen " + dave.find());
        dave.moveTo(5);
        System.out.println("moving dave ...");
        System.out.println("dave is now in pen " + dave.find());
    }
}
```

An *object* created from a *class* is sometimes known as an *instance* of a *class*. When everything in a *class* is static there can only ever be one *instance* of a class, i.e. all the variables etc. only exist once, resulting in the *procedural* programming style you were using before.

There is no exercise based directly on this material, but you will find it useful background to the next section.

Closing remark

This section has only scratched the surface of *object oriented* programming. Another important aspect of *OO* is *inheritance*, but it is beyond the scope of this introductory Java course. It is a large subject which encompasses both the design and implementation of software. You are encouraged to read more about it in the bibliography if you are interested.

9 Statistics exercise using external software

In this section, you will learn how to use external software packages to create and plot histograms and draw simple graphs. You will also get some experience of putting together a larger software project from several components. This is a little different from the previous sections, because you are expected to find out for yourself, from documentation and examples, how to use the software you are given.

Exercise 9.1 (15 marks)

Follow the instructions below to produce the programs described, which you should demonstrate with answers, listings, output and plots in your lab book.

Random number generator

Computer-generated random numbers are not truely random. This is because the computer can only carry out instructions, so the outcome is always predictable. However, it can give the appearance of randomness by using an algorithm which generates a sequence of numbers which behave very much like random numbers. Because the sequence has been calculated, it is reproducible. In fact, if you run your program several times with the same *seed*[15] it should always give the same sequence of 'random' numbers. For this reason they are known as *pseudo-random* numbers.

Java provides a *pseudo-random* number generator in the java.util package, documented here:
`http://download.oracle.com/javase/6/docs/api/java/util/Random.html`

This documentation explains the interface of the Random class in great detail: its constructors, methods and class variables. It is in a standard style of documentation for java, known as "javadoc". You may find it quite confusing at first but it is useful for reference so you should try to get used to it. However, all you need to know to use the Random class for this exercise is explained below.

Note that many of the explanations in the documentation contain code to show you how some methods are implemented. You do not have to write this code to use the Random class.

To use it, first you will need to put an import statement at the top of your program:

```
import java.util.Random;
```

You should create an object of `class` Random, using one of its constructors, e.g.

```
    Random r = new Random();
```

or, with a seed,

```
    long seed = 1220645228; // change this for a different sequence
    Random r = new Random(seed);
```

[15]an initial value supplied to the random number generator

Then you can call the *methods* of this *object* to get random numbers as required, e.g. `nextDouble()` and `nextGaussian()`. These are listed in the documentation.

Write a program to generate 100 random numbers with a distribution that is:

(i) uniform between 0 and 1,

(ii) Gaussian.

For now, your program should just print the numbers to the screen or a file.

Making histograms

The concept of a *histogram* is explained in Appendix D.

A Histogram *class* is provided for you to use. It is documented here:
`http://www.pp.rhul.ac.uk/~george/PH2150/jdocs/`.

To use the Histogram *class*, you need to download it from here:
`http://www.pp.rhul.ac.uk/~george/PH2150/downloads/Histogram.java`

There is also a test program in which a a Histogram object is created and used:
`http://www.pp.rhul.ac.uk/~george/PH2150/downloads/HistogramTest.java`

Copy the Histogram *class* from the web site and find out how to use it by looking at the documentation and the HistogramTest program.

Then, adapt your random number program from above to fill the generated random numbers into histograms. Use different histograms for the uniform and Gaussian distributed random numbers.

Add a *method* to the Histogram *class* which calculates the standard deviation of the histogram. The standard deviation is defined as:

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=0}^{M} W_i(x_i - \bar{x})^2}$$

where in this case, M is the number of bins, $\bar{x}$ is the mean of the histogram, for which a *method* is already provided, x_i is the central value of bin i and W_i is the height of the bin, which is used to weight the sum. N is the number of entries made into the histogram, not including over/underflows. Use this *method* to print the standard deviation of your Histogram objects.

Drawing graphs

The next step is to display your histograms graphically. To draw graphs you will use the PtPlot package which is freely available [6]. This is already installed on the network so you just have to know how to use it.

In order to make it easier to use, a `class` SimplePlot is provided for you. There are methods to plot a curve (`drawCurve`) and a histogram (`drawHistogram`).

To use the SimplePlot *class*, you need to download it from here:
`http://www.pp.rhul.ac.uk/~george/PH2150/downloads/SimplePlot.java`.

The SimplePlot *class* is documented here:
`http://www.pp.rhul.ac.uk/~george/PH2150/jdocs/`.

There is also a test program in which a a SimplePlot object is created and used:
`http://www.pp.rhul.ac.uk/~george/PH2150/downloads/SimplePlotTest.java`

Look at the SimplePlot *class*, documentation and example (the example is the best place to start). Using the instructions in the next section, try to compile and run the SimplePlotTest class. It should produce three graphs: a curve, a scatter plot and a histogram. Look at the code of SimplePlotTest to understand how to do this. Then use a SimplePlot object in your program to display the histogram.

Documentation for the PtPlot package is available here:
`http://ptolemy.eecs.berkeley.edu/java/ptplot5.3/ptolemy/plot/doc/index.htm`.

Compiling and running with PtPlot

To compile and run Java programs which use an external package, you need to provide some extra information when you give the compile and run commands. Note that these will only work in a MSDOS command window, not in the Jext console.

First you need to know where the external package is. For convenience, this location can be assigned to a MSDOS environment variable. It is recommended that you set `PTII` to the location of the PtPlot package:

```
set PTII=P:\Applications\Ptolemy\ptplot5.3
```

You can now refer to this location in java and javac commands by writing `%PTII%`.

To compile a program that uses some external package, you have to tell the compiler where to find this package, using the `-classpath` option, i.e.

```
javac -classpath %PTII%;. MyClass.java
```

Note that '`;.`' must be at the end of the *classpath*. This tells the compiler to continue looking in your current folder for files as well.

You only need to compile your class with the `main` *method*. The Java compiler will automatically work out which other classes need to be compiled for you and do this.

To run your program, you must again specify the *classpath*, i.e.

```
java -classpath %PTII%;. MyClass
```

Final exercise

Your program should now create and plot random number distributions. You can use this to investigate the *central limit theorem*.

Modify your program to make and plot histograms of the following:

(i) add 2 uniform random numbers, subtract the number 1;

(ii) add 4 uniform random numbers, subtract 2;

(iii) add 6 uniform random numbers, subtract 3;

(iv) add 12 uniform random numbers, subtract 6.

Uniform random numbers are taken to be distributed uniformly between 0 and 1. Make at least 1000 entries in each histogram.

Compare the mean and standard deviation of these histograms to those of the Gaussian you made earlier. Also compare the shapes you see in the graphical plots. What do you notice as the number of random numbers added together increases? What do you expect to happen as it approaches infinity?

This demonstrates the central limit theorem, which is explained further in many statistics books, such as [5].

References

[1] Richard Davies, Introductory Java for Scientists and Engineers, First Edition, Addison-Wesley, 1999.

[2] Mary Campione, Kathy Walrath, and Alison Huml, The Java Tutorial, Third Edition, Addison-Wesley, 2001.

[3] Online Resources for Java Programmers (a list of basic and advanced tutorials) `http://download.oracle.com/javase/tutorial/index.html`

[4] Java Platform, Standard Edition 6, API Specification `http://download.oracle.com/javase/6/docs/api/`

[5] Glen Cowan, Statistical Data Analysis, Oxford University Press, 1998. `http://www.pp.rhul.ac.uk/~cowan/stat/`

[6] Ptolemy Java Plotter (java classes for drawing graphs) `http://ptolemy.eecs.berkeley.edu/java/ptplot5.3/ptolemy/plot/doc/index.htm`

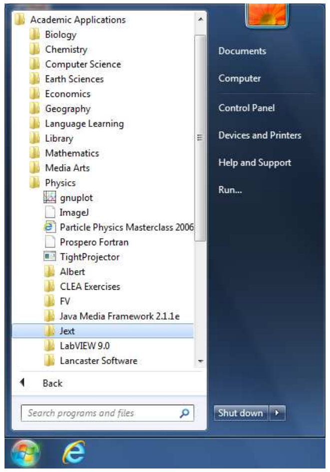
Appendices

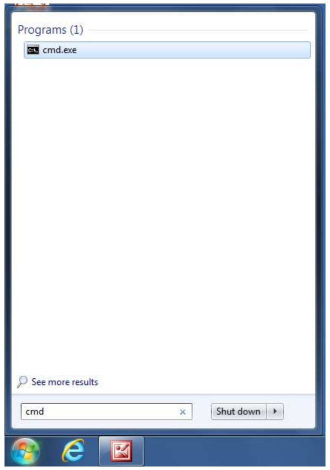
A How to use Java on the PC lab computers

Instructions

The recommended Java environment for this course is Oracle's Java Development Kit (JDK) version 6[16]. The standard edition is free to download and it contains all you need to compile and run Java programs. Later in this section there is information about installing it on your own computer. It is of course already installed on the lab PCs.

To write and edit Java programs, you can use any text editor such as Wordpad or Notepad, however the recommended editor is Jext. It is aware of the Java language, so it makes it easier for you by colour-coding key words and indenting correctly at the start of a new line. Jext is available on the lab PCs via the Windows Start Menu: Programs→Academic Applications→Physics→Jext





When you create a new file with Jext, you should set the language to Java on the menu at the top right of the Jext tool bar.

Files can be opened and saved through the File menu or using the appropriate icons. You must keep your own Java files in a folder under your home drive; it is suggested that you save them in a dedicated subdirectory called Y :\`java`. You can create this directory in the normal way using Windows Explorer.

To compile and run java programs, type commands into an MSDOS window[17]. To open a MSDOS window, go to the Windows Start menu and in the search box, type cmd then return.

[16] Older versions version of the JDK, such as 1.4.2 and 1.5 should also work fine.

[17] Alternatively you can use the Jext console. This is not generally recommended as it doesn't work correctly with programs that use text console input. To show the Jext console pane, go to the Edit menu of Jext, select "Options...", click on "General", then at the bottom of the screen, check the box "show top tabbed pane (Console)".

To change to the folder where you saved your Java file from the text editor, Y :`\java`, type:

```
Y:
cd Y:\java
```

Some other useful MSDOS commands and tips are given later in this section.

In order to run a Java program, you must first compile it. To compile the Java class defined in the file ‘ClassName.java’, simply type:

```
javac ClassName.java
```

To run a compiled class, type:

```
java ClassName
```

MSDOS tips

Directories

Directories, also known as folders, are organised in a tree-like structure. To move around directories:
`cd ..` moves up to the parent directory;
`dir` lists the files and sub-folders in the current folder;
`cd stuff` moves down into the folder called “stuff”, if it exists.

Screen size

You can change the font size of the MSDOS window and the number of lines it displays. Click on properties on the MSDOS toolbar, then once you have changed the settings, exit the MSDOS window and start it again for them to take effect. NB The font size must be small enough to accomodate the number of lines you wish to display on the screen.

You can switch between full screen mode and back by pressing <Alt>+<Return>.

Editing on the command-line

With the Windows XP MSDOS window, you can use the up/down arrows to recall from the previous lines you entered, and left and right arrows to move around a line and edit it.

Cut and paste

To copy text from the MSDOS window, click on the top-left icon of the window to get a menu →Edit →Mark. Now use the mouse to mark out an area of the text in the window to copy, then press Return to copy it. You can paste this into other applications, such as MS Word.

To copy the whole MSDOS window as a picture, press <Alt>+<Print Screen>. You can paste this into other applications, such as MS Word. This works for any window, not just MSDOS.

Redirecting program output to a file

If you program produces a lot of output, it may be convenient to send it to a file instead of the screen. With MSDOS you can do this by typing a greater than symbol followed by the filename after any command. For example:

```
java Hello > output.txt
```

You can then open the resulting file output.txt with any text editor (Jext, Notepad, Word) to edit, print or copy the text.

Because the program output is redirected to the file, you don’t see it on the screen at all. If you program expects some input, you won’t see any prompts you made for that either, but your program will still be waiting for you to type something. There is no easy solution to this; you just have remember what is required and enter it blindly.

Installing Java

It is freely available in case you have your own PC and would like to install it. To download the latest Java Development Kit (JDK), go to `http://www.oracle.com/technetwork/java/javase/downloads` scroll down to Java SE 6 Update 27 (or the latest update) and click on the download button for the JDK. Choose the appropriate version for your computer and operating system and follow the instructions carefully. Unfortunately, support for the download and installation of software on your own computer is beyond the scope of this course.

Installing Jext

The recommended editor, Jext, can also be downloaded for free from `http://sourceforge.net/projects/j`

B Dealing with errors

It is inevitable that code you write will initially suffer from errors. Many will be detected by the compiler which prints details of the errors to the screen. However some will only become evident when you try and run the program (runtime errors). This may be an exception in which case some details will be written to the screen, or you may be testing your program and have discovered that it is not giving the results you expect. Either way, you will have to *debug* your program; that is, work out why it is going wrong and how to fix it.

There are several techniques you can use to find out what the problems are.

Compile-time errors

- With errors detected by the compiler always **start with the first and work through them in order**, recompiling each time. You may find that solving the first error will eliminate the rest as sometimes a simple error at the beginning of a program will cause many **cascading errors** later on.
- If the compiler detects an error then look at the details it gives about which line in the code the error was found and what the error was. Look carefully at the line of code in your source file — you may simply have spelled something wrong or omitted a semicolon. Forgetting a semicolon will tend to result in an error message that refers to the following line (as it is the following line that then fails to make sense to the compiler), so you should also check the lines around the one mentioned by the compiler.
- If the compiler gives an error "variable not defined" or "cannot resolve symbol" when you are using some Java feature that you are sure you have spelled correctly, then the problem could be that you have not got the necessary `import` statement at the start of the file.

Run-time errors

- If an exception occurs when you try and run the program, again, look at the details written to the screen — the line number and the type of exception thrown — then examine this area of your source code. You may have written code that tries to go beyond the end of an array for example.
- If you suspect a particular block of code may be causing problems try surrounding it with the longer form of comment `/*...*/` which effectively removes it from your program without lots of unnecessary deleting. Then compile (and run) the program again to see if the error/exception still occurs — if it doesn't then the problem lies within the code that was commented out. This method can be tried with both compiler and runtime errors.
- So long as the program compiles okay, you can add `System.out.println(...);` to your program at strategic points to determine how far the program gets before the exception is thrown. This allows you to narrow down on where the problem code must be.

C Trapezium rule for integration

The trapezium rule is a numerical method for integration of a function between two bounds.

Other methods exist which are better than the trapezium rule, e.g. Simpson's rule. You can read more about them in text books or on the web.

Why would you want to integrate numerically rather than analytically? Here are two reasons:

- the function cannot be integrated analytically;
- you have a lot of integrations to perform, so some automated process is preferable.

Trapezium rule

The trapezium rule states that you can approximate the area under a curve by a trapezium, as shown in figure C.4. The area is therefore calculated like this:

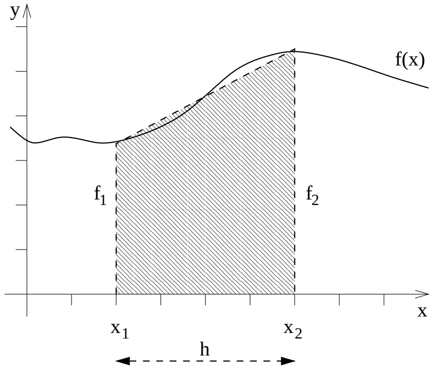


Figure C.4: *A straight line is used to approximate the curve $f(x)$ between x_1 and x_2. This forms a trapezium (shaded area) which can be used to calculate approximately the area under the curve between these bounds.*

$$\int_{x_1}^{x_2} f(x)dx = h\left(\frac{1}{2}f_1 + \frac{1}{2}f_2\right) + O(h^3 f'')$$

Where the notation f_1 is short hand for $f(x_1)$.

The error due to the approximation, $O(h^3 f'')$, indicates that this method is exact for polynomials up to degree 1, i.e. straight lines $f(x) = ax + b$, because for such functions the second derivative $f'' = 0$. It is clearly an approximation for higher order polynomials (x^2,x^3, etc.) and other functions. The error will also be small if h is small.

The area of a trapezium is its average height × its width, i.e. $\frac{h}{2}(f_1 + f_2)$.

Extended trapezium rule

To make this rule useful, break the curve to be integrated into many small intervals (small h). Then use the trapezium rule many times, like this (see figure C.5):

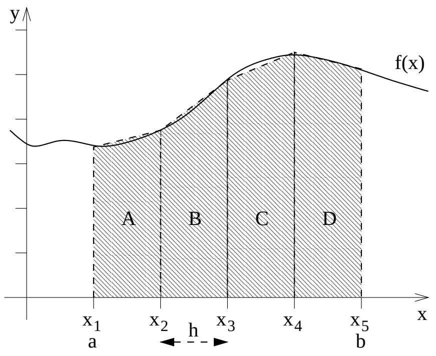


Figure C.5: *The curve $f(x)$ is now approximated by a series of short straight lines which each form a trapezium. Summing the area of these trapezia gives an approximation of the total area under the curve between a and b.*

$$\begin{aligned}\int_a^b f(x)dx &= h\left(\frac{1}{2}f_1+\frac{1}{2}f_2\right) & \ldots A\\ &+h\left(\frac{1}{2}f_2+\frac{1}{2}f_3\right) & \ldots B\\ &+h\left(\frac{1}{2}f_3+\frac{1}{2}f_4\right) & \ldots C\\ &+h\left(\frac{1}{2}f_4+\frac{1}{2}f_5\right) & \ldots D\\ &+O\left(\frac{(b-a)^3 f''}{N^2}\right)\end{aligned}$$

Generalise the formula to N points:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1+f_2+f_3+...+f_{N-1}+\frac{1}{2}f_N\right]+O\left(\frac{(b-a)^3 f''}{N^2}\right)$$

$$= h\left[\frac{1}{2}(f_1+f_N)+\sum_{i=2}^{N-1} f_i\right]+O\left(\frac{(b-a)^3 f''}{N^2}\right)$$

Note that h is still the width of a single interval, so

$$h=\frac{b-a}{N-1}$$

The approximation error is now

$$O\left(\frac{(b-a)^3 f''}{N^2}\right)$$

$b-a$ and f'' are normally fixed parameters of the problem. You can see that increasing N will improve the accuracy.

Programming

Think about the following when you plan your program.

- Break the problem down into smaller parts, and consider writing these as different methods of your class.
- Which parts of your program do you want to be able to change easily? Try to put these in separate methods.
- Which parameters would it be convenient to ask the user to enter?
- How to program this formula?

$$I = h\left[\frac{1}{2}(f_1+f_N)+\sum_{i=2}^{N-1} f_i\right]$$

- *Summation* is well suited to a particular type of Java command, as you have seen in the matrix multiplication exercise (exercise 4.3).

Pitfalls

Here are some (but not all) of the common problems associated with the exercise.

Code

- Data types — make sure your choice of `int` or `double` is appropriate for the variable. Remember the effects of integer division!
- Loop counters should be integers to avoid rounding errors.
- Don't forget to initialise variables where necessary.
- Limit the scope of variable declarations to the minimum required.

Method

- Beware of limitations of the Trapezium rule.
- Understand the behaviour of the function before trying to integrate it.
- Choose N to reach a reasonably accurate result. Vary N to understand the accuracy.

- The function must not be sharply concentrated in peak(s) — this implies f'' is large.
- The function must be characterised by a single length scale (i.e. y range does not cover several orders of magnitude) — again, this implies f'' is large.

D Histograms

A histogram is a way of representing the frequency distribution of a quantity which changes randomly each time it is measured. Most other types of graph are used to show the correlation between different variables. The key difference between a histogram and other types of graph is that there is only one variable and the aim is to show how its values are distributed.

A histogram is similar in appearance to a bar (or column) chart where categories are marked along the x-axis and the height of columns above these categories shows their relative frequency. A bar chart, although it looks similar to a histogram, has discrete categories on the x-axis rather than a continuous variable. The x-axis of a histogram represents a quantity which is continuous, but the the axis is still divided into bins. The column heights in each bin show the relative frequency of data with a value between the lower and upper edges of the bin. Figure D.6 shows an example of such a histogram.

Histograms are typically used to show data which are collected by repeated measurements of a quantity whose value varies according to some probability density, such as the lifetime of an unstable nucleus or particle before it decays.

Data
2.11181
2.2664
1.93247
4.13048
4.54348
3.35372
3.9448
1.83678
2.72451
6.02379
10.7733
...

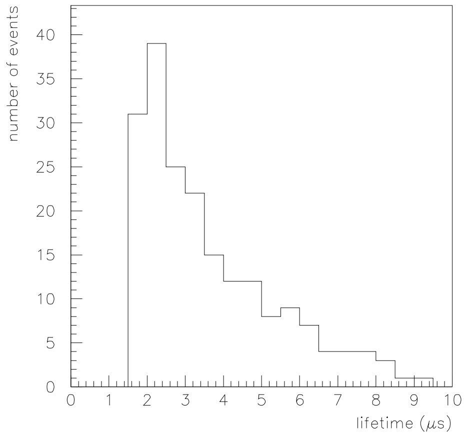


Figure D.6: *Example of a histogram with a continuous variable. Part of the data collected in the histogram is shown on the left.*