UNIVERSITY OF ABERDEEN

Department of

Mathematical Sciences

Mathematical Computation MX3015

Ian Craw

September 9, 2003, Version 3.0

Additional copies may be obtained from:

Department of Mathematical Sciences
University of Aberdeen
Aberdeen AB9 2TY

DSN: mth203-105030-7

Foreword

What the Course Tries to Do

It is always a good idea to start by saying what we are trying to do, and how you will recognise if you have succeeded! Modern education jargon sums up this by emphasising "aims" and "learning objectives". Here they are in the same form as in the "Catalogue of Courses". It is hard to understand what the objectives *mean* before you have finished the course; but at that stage, you should come back here and check that they do make sense. If they don't, you may have missed something important.

Aims

The overall aim of the course is to present modern computer programming techniques in the context of mathematical computation and numerical analysis and to foster the independence needed to use these techniques as appropriate in subsequent work.

Learning Objectives

By the end of the course the student should:

- be aware of the background of computing within mathematics, and of possible choices of programming language;
- be able to analyse appropriate mathematical problems in a form suitable for programming;
- be able to construct programs in a modern object-oriented programming language, using the available facilities appropriately;
- be able to describe, analyse, program and contrast a number of methods for investigating problems in symbolic computation and numerical analysis. These problems will include at least three topics from the following:
  - root finding;
  - numerical integration;
  - the solution of ordinary differential equations;
  - digital signatures;
  - Monte Carlo methods;

- be able to write programs to assist investigations in subsequent mathematics courses; and
- have improved analytic skills.

Syllabus

The following detailed course description was made available in a separate sheet, but it may be convenient to repeat it here. And I confess that, as the course evolves, I'm not promising to get through it all.

Lecture 1: The File System and "Hello World". An introduction to the computer: the CPU, memory, disc, files. Organising: directory (or folder) structures and filenames. The concept of a program; the "hello world" program in JAVA.

Practical 1 Concentrates on the use of the local environment. At the end of the practical you should be familiar with how to log on, run a simple JAVA program, manipulate your file space to keep it tidy, get printed output, use a browser to look at documentation on the web and (of course) log out at the end of the session.

Lecture 2: Writing Simple Programs begins the main description of the JAVA language, starting with variables, data types and simple control structures You will also meet π and functions such as the square root function.

Lecture 3: More Loops discusses control structures, presenting more elaborate examples, and includes recursion.

Practical 2: Simple Programs You will practice writing simple programs of your own using some of the built in functions and primitive data types.

Lecture 4: An Introduction to Objects We discuss classes in more detail, including access rights. You will see how inheritance works and how and why you should consider accessors and mutator methods.

Lecture 5: A Review of the Java language We review and extend our discussion of JAVA syntax and then move on to the crucial concept of an *object* and describing member variables, methods etc.

Practical 3: Programs with Control: Classes We start using more elaborate control structures, providing further practice of the work in Lecture 3. We also create some simple classes.

Lecture 6: Using Objects This continues the introduction to object oriented programming which is one of the more important features of JAVA. Attention is focussed on subclasses and interfaces, and the structure of earlier programs is discussed. You will revisit some built-in functions as static methods.

Lecture 7: Arrays You will meet arrays of simple data types and of objects and apply them to natural tasks including computing statistics and matrix manipulation.

Practical 4: More on Classes provides an opportunity to work with objects and allow them to interact. You will learn to read sample classes, and design and write simple classes of your own.

Lecture 8: Introduction to Computing and Numbers will start with a background to computing aiming to put JAVA in its historical context. We shall also discuss the problems of data representation and describe how that is done within JAVA.

Lecture 9: The Art of Programming continues a discussion of representation and moves on to talk about program organisation and the value in keeping structures hidden, commenting, testing, debugging, documentation etc, including the JAVADOC system which you will be expected to use to document your own programs.

Practical 5: Arrays This provides another opportunity to practice writing JAVA; this time arrays are available to ease certain tasks.

Lecture 10: Built-in Objects discusses object which are already available for use, including the classes such as Integer which wrap primitive types and the String class, which unusually is final. You will also meet the Object class itself, and container objects including HashTable and Vector.

Lecture 11: Errors and Exceptions concentrates on program organisation and discusses the need for disjoint namespaces. Packages are introduced as a solution and we show how to embed classes within a package. The benefits of exception handling are discussed and the facilities available with JAVA are described. We illustrate by showing how to write data to a file.

Practical 6: Classes, Files and Exceptions Consolidation of class creation and further practice using built-in classes. The practical also gives experience with defensive programming using exceptions and writing data to file.

Lecture 12: File Handling; and Formulae concentrates on input and output, showing how files and data sets can be manipulated. It also illustrates programming some common functions such as $\sin x$.

Lecture 13: Solving Quadratics shows that even invoking the quadratic formula can lead to unsatisfactory results. There is a discussion of other types of problem which can arise in numerical analysis.

Practical 7: Using Formulae in which a "correct" version of the quadratic formula is explored. This is based on a quadratic class capable of initialisation and output to file. We also look at the formula for $\sin x$.

Lecture 14: Finding Roots of equations discusses approximate methods starting with the bisection method. We discuss orders of convergence.

Lecture 15: More methods of root finding introduces more elaborate methods including the Secant method and Newton's method. Convergence criteria are discussed.

Practical 8: Finding Roots concentrates on finding roots of equations and uses the classical methods; the bisection and secant method and Newton's method.

Lecture 16: Numerical Integration and the Trapezium Rule discusses the problems associated with numerical integration and illustrates them with a detailed discussion of the trapezium rule.

Lecture 17: Numerical Integration Revisited Simpson's rule is discussed in detail, Romberg's method is mentioned briefly and adaptive methods are described and an implementation outline discussed.

Practical 9: Quadrature, or Numerical Integration This concentrates on the classical methods; trapezium rule and Simpson's rule as well as adaptive variants.

Lecture 18: Randomness We show how a "random Number" generator can be built and explore the facilities in Java for generating various random objects.

Lecture 19: Simulation This continues the theme of the previous lecture; more random structures are introduced, including method such as "conveyor belt" sampling. An example simulation is presented and Monte Carlo integration is discussed as another simulation example.

Practical 10 Random Numbers concentrates on returning random objects of various sorts, and simulation including Monte Carlo methods of integration.

Lecture 20: Cryptography An introduction to the use of prime numbers in cryptography, and a discussion of modern uses of cryptography including digital signatures.

Lecture 21: Large Integers and Primes Handling large integers and the generation of large primes with a view to their use in an RSA algorithm.

Practical 11 Primes and Cryptography is devoted to exercising the `BigInteger` and getting some results on prime numbers.

Lecture 22: Applets and GUIs We show how java can be used to provide a modern user interface, and can be used on web pages. This involves a discussion of many new Java classes, including those that are available in `swing`

Lecture 23: Events further work on the GUI leads to more interesting applets with interaction. Neither this nor the previous lecture will be directly tested in examination.

Practical 12 Tidying In this practical you simply have one aim - tidying up the exercises we have requested for submission. Note that in this practical we are *not* willing to help with your Java; only with system difficulties.

Lecture 24: A review of the Java language In the final lecture we stand back and review the parts of the language which we have met. We discuss how to continue Java programming.

The MX3015 Mailing List

There is a mailing list associated with this class. You can subscribe to it by sending email to `majordomo@maths.abdn.ac.uk` with a message that has an empty "subject" field and contains the single line `subscribe mx3015-list`. If you add your signature automatically, you can also add the line `end` after the subscribe request, and the signature will be ignored. You are encouraged to join this list. You then mail `mx3015-list@maths.abdn.ac.uk` to contribute to the list.

I have always been happy to deal with questions by email, and have made a point of publishing my email address both on the web and in the printed notes. This list provides a more general way of communicating in which both questions and their answers go to everyone on the list. Here are some reasons why this might be useful:

- It is rare for just one person in a class to have a given problem; by sending the answer to this list, others can benefit from it.
- When a topic causes general concern, the lectures can be changed to cover it in greater detail, or pick up the problem area.
- Often other members of the class can be of more help than the lecturer; this way everyone is aware of the problems and is invited to help or comment.
- I have always been careful to make statements about the examination in front of the whole class – the list provides an equivalent public forum.

Please note that this list is being maintained on the mathematics machines rather than the central University ones, so you need to mail `maths.abdn.ac.uk` to reach it.

Finally some points of netiquette.

- Please remember the usual courtesies; although this is email, it is still fairly public.
- If you send email directly to me, I will not copy it to this list without first getting your permission.
- The list is low security. Although you are technically able to impersonate someone else, please don't do so.
- Keep a copy of the message you get when you join the list to remind you how to leave; you are welcome to leave and re-join as often as you wish.

Sometimes it is useful to pass a message to the whole class. I believe that for most people, this list is more useful way than posting it on a notice board. One natural use would be to cancel a lecture if that has to be done unexpectedly. The message may not reach everyone, but those who read it will be saved the inconvenience of turning up.

Any more questions? Why not mail the list? You won't be the only one with them.

Books

An earlier version of these notes was written in conjunction with Andrew Mellanby, who was the Department's Computing Officer for many years. His name is no longer on the author list, since I am no longer sure he would wish to accept responsibility for them. I hope these notes will prove a useful reference but you will need a "proper" book as well. I'm not working directly from any book, so choose according to your needs. If you are just starting computing, my recommendation is probably Bishop & Bishop (2000). It is a nice recent book with an emphasis on numerical calculation and has a website. The same authors have a more general JAVA book which you may prefer. As an alternative, you may wish to look at Davies (1999). This is certainly not comprehensive, but provides a very gently introduction and has much the same motivation as this course.

A significantly more advanced book, which covers the basics carefully but also considers abstract data types in detail and and worthwhile discussions on design and testing issues is Winder & Roberts (2000). I myself use two more "big" JAVA books - those I go to when I want details of something fairly obscure. I have found both very useful. The first, Eckel (1998) has a web site which contains all the examples in the book and also the complete text of the book, which you are welcome to download if you wish. The most recent one I use is Horton (1999). This too has a web site containing code. And now that JAVA is taught to first year Computing Scientists, their recommendation (Deitel & Deitel 2002) is easy to find in the bookshop.

> **These notes don't pretend to describe the Java language fully and properly; you need a book.**

Although we don't go into a lot of detail about numerical analysis, it is worthwhile noting some books. A very standard one, if a little dated is Conte & de Boor (1980). More recent, with a greater emphasis on actual computation is Faires & Burden (1998). And I shouldn't leave this section without drawing your attention to the "bible", where most workers in the area go to get authoritative algorithms, namely Press, Flannery, Teukolsky & Vetterling (1992). This version happens to be in C, but there are versions in a number of languages. Finally, for the number theory, I've used two books; start with at Burton (1995); ten for a more advanced approach in our spirit, take a look at Motwani & Raghavan (1995, Chapter 14).

Ian Craw and Andrew Mellanby
Department of Mathematical Sciences
Meston Building
email: `Ian.Craw@maths.abdn.ac.uk`
`www:http://www.maths.abdn.ac.uk/~igc`
September 9, 2003

Contents

Chapter 1

Basics

We begin with some of the concepts and terminology of computing, particularly those relevant to writing computer programs.

At the first practical session, you will be logging in to one of the classroom computers, configuring you computing environment to make the task of programming more comfortable and creating a very simple program written in the JAVA programming language. You will have to compile and run your program and check it for errors. This Chapter should help you with this.

We assume that this is your first serious use of computers. Please ask for help in the practical classes when there is a problem; if you can't do what you are trying to do, there may may be other factors relating to the system software or the computer network which are causing problems. And if you *do* have some computing experience, please try to help those with less experience.

1.1 The Computing Environment

A computer has components of two types; **hardware** and **software**. Hardware consist of the physical components of the computer; things you can, in principle see and touch: software consists of steams of bits, or signals which can be either present or not, which control the computer and make it do something useful. This course is concerned with programming in the language JAVA, showing how to write certain fairly general types of software. We take up that study in Chapter 2. Before that we need to know a little about the organisation of a computer and its **operating system**.

Physically a computer normally contains most of the following components:

- **A CPU:** the Central Processing Unit, which interprets programs and controls the other computer components;
- **Memory:** short term storage for programs and data which are being processed by the CPU (also known as **RAM**);
- **Disk(s):** long-term storage for program files and data which remains even when the computer is switched off;
- **Input/Output devices:** devices such as a keyboard, mouse, screen or printer, which allow the computer to communicate with the outside world; and often

- **Network connections:**: which allows connection to other computers, so that data can be interchanged.

These components interact under the control of a program. In a very simple computer, that program could simply be an arrangement of wires — the program is “hard wired” — but in general, there will be one supervising program, called the **operating system**, which is used to decide which other programs, usually called **application programs**, are run.

The operating system or **OS** is the main program that controls how the computer works. It is loaded automatically from disk when the computer is switched on and is in charge of everything else that goes on, from reading key presses on the keyboard or displaying things on the screen, to putting up a password prompt and verifying that you are entitled to log in. In the classrooms, you will be interacting with the MICROSOFT WINDOWS 2000 operating system.

Application programs can be very elaborate. You are probably familiar with a number of such programs: the Microsoft Office Suite including Word and Excel; Maple for doing symbolic mathematics; Eudora for managing your email; or Netscape, or Internet Explorer for browsing on the Internet.

In this course, we are going to create some simple applications software. We are going to create programs which perform useful tasks such as computing statistics from data and finding the roots of equations. To create these programs, we will write **source code** in a language called JAVA and use an application program, a JAVA **compiler** to convert the source files into an executable program. Once we have created an executable program, it can be run.

The part of the operating system which controls interaction with a user is called the **User Interface**. Often such an interface is Graphical (a **GUI**), involving windows displayed on the screen, small windows called **icons** and interaction using a **mouse** as well as a keyboard. Programs are usually launched by double-clicking an icon or selecting them from a menu. You have probably already met the standard Microsoft desktop user interface; variants are used in Windows95/98/ME as well as Windows XP and Windows 2000.

In this course, you will be using the Microsoft desktop user interface, but you will also use a command line interface (**CLI**), where you simply type commands into a window and get textual feedback about what happened. This simplicity can sometimes be helpful, allowing you to concentrate on the task in hand. The program you interact directly with, which accepts your commands and provides the feedback, is called a **shell**. We choose to use a non-Microsoft command line interface with a shell called **bash**[1]. This, and in fact *all* the software we use in the course, is freely available to download from the Internet. If you have a reasonably capable PC at home, you can duplicate, and in fact improve on[2], the environment we use in the classroom.

1.2 Files

Data, programs and such like are stored in individual **files**, the basic organisational unit of storage on a computer. We are concerned here with files that you will create and work

[1]One of the earliest shells on UNIX, was written by Steve Bourne. The shell we use, `bash`, the Bourne Again SHell, is a development of the original.

[2]Keeping both programs and your files on a single machine makes things faster. My own system, comparable to a classroom machine, runs a set of tests around 20 times faster than they run in the classroom.

with. More generally, all data, such as your email and even the operating system itself are stored in files.

Modern operating systems try to hide this organisation from the user. When you work on email in `Eudora` you are more than happy to let `Eudora` worry about storing them; you are more likely to see an organisation, such as a collection of mailboxes, which is more appropriate to the task in hand.

However writing and running program on a machine requires more contact with the operating system and with files. In particular, there are some conventions in JAVA about how files are to be named, which we will discuss later. At present we need to say more about how files can be organised and manipulated.

1.2.1 File names

Almost any computer now will need to store many thousands of files. Conventions have grown up about how they should be named in order to simplify their organisation and use. A convention used very frequently is that the a file name is split into two parts, joined by a single '.'. The first part of a file name (before the dot) is called the **basename**; the part after the dot is called the **extension**. Usually the basename describes the *purpose* of the file, while the *extension* describes the *type* of the file.

For example a file `letter_home.doc` is likely to be a file created by `Microsoft word`. And usually, if you click on the file, `Microsoft word` is started, and given access to the datafile `letter_home.doc`. Files may be renamed as you please. Renaming a file will not change its contents, but it may change the way it is treated, so you should rename with caution. We will see that not only do JAVA source files have to have a `.java` extension, but the basename must correspond to the class declared within the JAVA code in the file.

File names are in principle *case sensitive* and the interface we use will distinguish between `MyProgram.java` and `myprogram.java`. There are quite strong reasons in what we do, *not* to have a space in a filename. It isn't wrong to have one, but manipulation can sometimes be much harder.

1.2.2 Grouping Files

Naming files sensibly goes a long way towards keeping things organised, but there are many benefits from more organisation still. It is helpful to keep related files together. Microsoft uses the analogy of a **folder** to describe a container holding either files or other folders or both. This puts a natural tree structure on your files and folders. You may also have heard of these as **directories**, the name that we will use for them. By using lots of directories, lots of subdirectories and so on, and adopting a sensible naming convention, we can access projects and files of many different types very quickly. For example we will recommend, or perhaps insist, that you keep all of your work for this course in a directory called `mx3015`. In fact almost every practical session will need a different subdirectory of that directory. Without this organisation you can easily get lost or confused; at present I have 513 files containing bits of JAVA associated with the course. You may be familiar with the MICROSOFT WINDOWS 2000 `Explorer` which allows you to see this organisation easily; we discuss the command line equivalents on Section 1.3. And of course files with the same name but in different directories can be easily distinguished. Many people adopt the convention that, in general, a directory either contains (sub) directories *or* files.

1.2.3 Naming Files and Directories

A file can always be specified by its full or **absolute** name. The full name of a file is much like a web address, except that it goes from the root of the tree to the branches or files[3], and it uses the `/` separator, just as subparts of a web address do. As an example, this chapter of my notes might be in a file called `H://tch/mx3015/notes/basic.tex`. The file itself is called `basic.tex`; the rest of the name is the **full pathname**; the list of directories you must traverse, starting at my home directory to locate the file. Having your home directory at the root of a filesystem is not the only choice; on my own system, there is a single root, usually called "/", or "slash". My home directory then appears as `/home/igc` and drive letters, now really an anachronism, can be hidden.

In a highly structured and organised filesystem, typing full pathnames is tedious. To make life easier, there are shorter versions, called **relative pathnames**, which we will use almost all of the time. To discuss them we first need the concept of a *current directory.* This is primarily used in a command line interface; it is simply the directory that you are "in" when you type a command. When you first start a command window, your current directory is at the top of *your* file hierarchy. This directory is referred to as your **home directory**, and is the place from which you will start extending your directory hierarchy — for example by creating the directory `mx3015`. Relative pathnames are then the path names of files or directories, *relative to the current directory.* In particular if `myfile` is in the current directory, its relative pathname is just `myfile`. A relative pathname does *not* start with `H://`, but rather with a directory or filename. Thus in your home directory you can simply speak of `mx3015` to mean the directory we have just discussed creating. The current directory, or the "current working directory" can be changed with a command like `cd mx3015`; in our example the current directory would then be `H://mx3015`. At any time you can check the current directory with the command `pwd`.

There are some even shorter directory names; '`~`' means the present user's home directory, '`.`' means the current directory, '`H://`' means the root of the filesystem while '`..`' is the parent directory of the current directory. This latter is often pictured as a 'back arrow" in a windows environment. Such short names can be used whenever they are convenient. And a plain '`cd`' is interpreted as '`cd ~`' or "take me home".

1.3 Manipulate Files and Directories

One way to move around the file system and manipulate files and directories, is to issue commands at the `bash%` prompt. The general form of a command is

```
bash% <command name> <options> <arguments>
```

Although the difference isn't very great, you can think of the *options* as governing how the command works, and the *arguments* as saying what the command should work on. We shall make use of quite a large number of options for some commands, which need to vary with the situation. In these circumstances, it is often easier, as we do, to avoid the "point and click" idiom and work directly at the command line.

There are many different commands available to you; Table 1.1 lists some of the more important. The behaviour of the command can often be modified by specifying `<options>`. Thus `ls -l` with the option `-l` specifies a *long* listing of files and directories, giving more

[3]Web addresses like `www.maths.abdn.ac.uk` go from the specific to the `.uk` suffix which is very general.

information. Options usually begin with the `-` symbol; often you can find out what options are available by using the command with the option `--help`. Finally, there come the `<arguments>` if any. If you wanted a long listing of the `mx3015` subdirectory, sorted by time of creation (the option `-t`) rather than last access, with the most recent last (option `-r` for reverse the usual order), the command would be

```
bash% ls -lrt mx3015
```

We list the commands for creating files and directories, copying and moving files and directories, as well as the commands for moving about in the file system, in Table 1.1. Exercises in the first Practical will give you experience of their use.

Command	Short for	Function
`pwd`	print working directory	Show me where I am in the file system!
`ls`	list	Show me what is in the current directory.
`cd <newdir>`	change directory	Make `<newdir>` the current directory
`cp <old> <newfile>`	copy	Copy the contents of file `<old>` and save it as file `<newfile>`.
`cp <file> <dir>`	copy	Copy the contents of file `<file>` and save it as file `<file>` in directory `<dir>`.
`mv <old> <new>`	move	Rename the file (or directory) `<old>` and call it `<new>`.
`mkdir <name>`	make directory	Make a new directory called `<name>`
`cd <newdir>`	change directory	Make `<newdir>` into your current directory.
`rm <file>`	remove	Delete the file called `<file>`. There is *no* undelete command, so use this carefully.
`rmdir <name>`	remove directory	Remove the *empty* directory called `<name>`
`rm -r <dir>`	remove recursively	Delete the file or directory called `<dir>` and anything it contains. Again there is *no* undelete command.
`touch <name>`	$\Longrightarrow$ existence?	Simply create an empty file; one with nothing in it.
`cat <file(s)>`	concatenate	Lists the contents of `<file(s)>`, one after the other.

Table 1.1: Commands available at the `bash` prompt

Almost all of these commands have options which tailor their behaviour. An example is the `-r` option given above for the `rm` command, which made it do much more work. A complete list for each of these command can be obtained using the option `--help`.

1.4 A Java program

With this background, I now describe the whole process of creating and running your first JAVA program. There are several steps.

- Create the source file in a text editor; then
- compile the program; and finally
- run the program.

As before I'm going to describe *one* way of doing each stage. There are usually many different ways of doing what is needed. If you are an expert, you are welcome to try your own way. Overall however, the way we describe here is probably the simplest and most reliable.

1.4.1 Editing Text

In order to create the source file, you need to use a **text editor**. A source code file must be a **text** file. These can only contain standard alphanumeric symbols — the symbols you get by typing on your keyboard — although other symbols may be used in other types of files, usually called **binary** files. Files produced by word processors use non alphanumeric characters to indicate formatting information such as the size and shape of text. Such files would be invalid as source code files; thus you *can't* use (say) Microsoft word as an editor.[4] A text editor ensures that *only* the allowed characters are included in our source files. We (very) strongly suggest you use `emacs`. This understands the JAVA language and can help to lay out your code sensibly. It is also freely available on the Internet.

Much of the time you will use two separate windows, perhaps with a browser (INTERNET EXPLORER) as well. You are advised to remove other clutter from the desktop.

First start `emacs` from a `bash` window with the command

```
bash% emacs &
```

The '`&`' lets you continue to use `bash` while `emacs` is running. It opens a new **editor** window on the screen, into which you can type text. The text is retained in something called a "buffer". You will normally start `emacs` only once during each session, keeping the window open for the rest of your session. The `emacs` editor has a menu like many windows applications complete with pop-up dialog boxes. Alteratively you can use keyboard shortcuts, in which case, `emacs` prompts for instructions on the bottom line of the display. The line above that, called the "status line" has information you may find useful.

From the `emacs` menu, select Select 'Files→Open File'. The 'File name' box will be empty, while the'Look in' box shows the name of the current directory.. To create a new file with the right name, type `HelloWorld.java`[5] in the 'File name' box and select 'Open' to show you have finished entering the filename.

[4] without a great deal of care.
[5] The tradition of having "Hello World" as the first program in a new language goes back to 1973!

It is now time to create your first JAVA source code. Type the following few lines into `emacs`.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

As you type you should see that the words are being displayed in different colours[6]; you will soon be able to use this to spot mis-types. Note also that pressing `<Tab>` anywhere in the line corrects the indentation used. Both this and the syntax highlighting show that `emacs` understands quite a lot about the JAVA language. Finally, make sure you write the buffer from `emacs` back to your file space. You can do this from the `emacs` menu, selecting 'Files→Save Buffer'.

Much of what you have typed, and its arrangement, will seem completely meaningless. Finding out about all the bits used in the program, like `static` or `String[]` is the job of Chapter 3. For the time being, the important thing is to copy accurately!

1.4.2 Compilation

Source files are only the first stage in creating a program. The CPU cannot directly work with a JAVA source file which has been designed to be meaningful to people. Instead the source file has to be compiled into a more machine compatible form, sometimes called **byte code**. In fact JAVA compiles to a **virtual machine** which is then interpreted on the actual machine. This way JAVA byte code can be independent of the CPU; it can run on any computer with a JAVA interpreter.

To compile the program, we run `javac`, the JAVA compiler with the command

```
bash% javac HelloWorld.java
```

Remember about "command completion"; just typing `javac H<tab>` is probably enough. If there were no errors the command prompt will return without comment. It is easy to mis-type and introduce errors. In this case the compiler often reports the line number of the offending line, which can help you spot your mistake. If there were errors, they should be corrected in the editor and the new version of the source code written out, so that the program compiles.

Compiling the JAVA program will have created a 'class' file `HelloWorld.class`. This JAVA program can then be run with the command

```
bash% java HelloWorld
```

Class files may be deleted when you are finished for the day; they can easily be recreated at a future date by recompiling the source file.

1.5 Making Things Easy

Finally we have a list of tips which can make working in the combination of `emacs` and `bash` seem simpler.

[6]This is called syntax highlighting.

- Remember that filenames are treated in the same way in both programs. In each case you can refer to the file `H:/mx3015/Test.java`. In fact `emacs` can cope with `\` as the separator; `bash` can't; `bash` can handle `//h/mx3015/Test.java` while `emacs` has problems.
- If you get stuck in `emacs`, the panic button' is `<ctrl-G>`; in other words, hold down the "Control" key and at the same time, type "g". You may have to do this several times, but eventually it will return the editor to the state you expect.
- If you make a mistake in `emacs`, you can undo it, as in many programs, by pressing `<ctrl-Z>`. Keep pressing and you undo more and more until you eventaully get back to where ytou started. And if you press any key other than `<ctrl-Z>`, the next "undo" starts, as always, by undoing your last actions; in this case, by undoing the undoing!
- Use command and file completion. Both programs understand `<Tab>` as a completion character. So if you have to type a long filename or path, just type the first letter or so and then press `<Tab>`; it will be completed until there is ambiguity, when you can type some more.
- Use command history and edit the command line. In both `bash` and `emacs` on the command line, the arrow keys will allow you to move around; the "up-arrow" key allows you to go back to the previous command; you can then edit it if you wish — and press `Return` to execute it.
- In `bash` `<ctrl-U>` removes anything you have typed on the command line, while `<ctrl-C>` interrupts and stops the current command.
- Remember that this is still a windows environment. If you *just* have `bash` and `emacs` open, the usual `<Alt-Tab>` command will switch smoothly between these applications.
- Finally note that you can't quit the `bash` window from which you started `emacs` until you quit `emacs`. It is probably easiest to open `bash`, open `emacs` and then keep both open and in use until you quit.

Chapter 2

Elementary Java language

Before we start I need to make a point about the examples I present in these notes. In “real” programs a significant amount of the code is devoted to checking for errors in the data on which the program is going to work. So before creating a circle with a given radius, it is a good idea to check that the radius is positive; again the length of the side of a rectangle should be positive and so on. This is vital, but can make it harder to see the underlying ideas.

> **In these notes, examples are usually presented *without* error checking code. This is done to show the underlying ideas. It is not good practice in general to miss out such checks.**

One of the strengths of JAVA is that a way of doing such error checking, using an `Exception` is built into the language. We look at this in more detail in Chapter 7.

2.1 Objects

The first computer programs were simple affairs, a little like the program you would use on a pocket calculator today. They simply described what had to be done, and in what order to do things. This idea worked well for many years and is often described as **procedural** programming. In recent years however people have found that a (more) fruitful way of arranging a computer program has been to think in terms of objects and the methods by which they can be manipulated. This is **object oriented** programming and the ideas were built into the structure of JAVA.

It is commonplace that everything in mathematics is an object. We speak of sets, say A and B, of real numbers x, y and z, or of a group G with elements g and h and so on. If we are being formal, we will speak of the *class* of all sets and describe A and B as *objects* from this class. We may wish to think of x and y as objects from the class of real numbers. In the same way we might consider the class of all members of a fixed group G and describe g and h as members of G or perhaps as objects in G. So in mathematics much of our concern is with objects of one sort or another. Here are some more examples:

- numbers of various sorts; rational or complex numbers or even algebraic integers;
- geometric objects, perhaps shapes of various sorts such as rectangles, circles, ellipses and so on;

- permutations, or maybe cycles[1];
- sequences or series;
- functions of various sorts, including polynomials and rational functions; or
- statistics (mean and variance for example) associated with a dataset.

Already it is clear that the list can be extended to encompass much of mathematics. The point of an object like (say) a cycle is that there are many different cycles and they share common properties, while often we will wish to manipulate one or two specific cycles. This prevalence of objects means that JAVA often fits a problem within mathematics or statistics quite nicely.

This language lays down what types of object we are dealing with, and often gives access to familiar *methods* for manipulating such objects. It would be normal to speak of $x + y$ without comment if x and y were real numbers; the addition on $\mathbb{R}$ is standard. In the same way, if G was an *abelian* group, then $g+h$ would make perfect sense; but it would look like a misprint if G was non-abelian, when we would expect the composition to be written as gh. And while, if A and B were sets, we might be happy with $A \cup B$, the notion of $A + B$ would be unexpected except in certain contexts.

This type of convention has turned out to be useful in programming, and for much the same reasons. Thus many modern programming languages such as JAVA are what is known as **object oriented** languages. Everything in JAVA is an **object** with a few primitive exceptions and we have conventions for describing and manipulating objects. Here is how JAVA might describe a class of `Rectangle`s:

```
public class Rectangle {
    // Data for the class
    double width;
    double height;
    // A constructor
    Rectangle (double w, double h) {
        width = w;
        height = h;
    }
    double area() {
        return width*height;
    }
    boolean isLandscape() {
        return (width > height);
    }
    // More methods here perhaps?
}
```

As you might expect, there are in principle many different objects in this class. Each `rectangle` has its own data; values for its `width` and `height`. In addition we describe a

[1]A reminder: the cycle $(1, 3, 4, 7)$ can be regarded as a permutation in (say) S_7 in which $1 \to 3$, $2 \to 2$, $3 \to 4$, $4 \to 7$, $5 \to 5$ and $6 \to 6$.

way of constructing a `Rectangle` and give some methods such as one to find the rectangle's `area()`, or to say whether the rectangle is longer than it is high with `isLandscape()`.

There is nothing particularly special about this example. Here is one way to write a `Complex` class.

```
public class Complex {
    // Data for the class
    double x;
    double y;
    // A constructor
    Complex (double realPart, double imagPart) {
        x = realPart;
        y = imagPart;
    }
    // Methods
    double modulus() {
        return Math.sqrt(x*x + y*y);
    }
    Complex conjugate() {
        return new Complex(x,-y);
    }
    public static void main(String args[]) {
        Complex z1 = new Complex(1.0,2.0);
        Complex z2 = new Complex(-3.0,-4.0);
        Complex z3 = z2.conjugate();
        System.out.println("z1 has modulus " + z1.modulus());
        System.out.println("z3 is " + z3);
    }
}
```

Objects in this class are just complex numbers. Again each objects has its own data, we can construct such an object, and this time there are methods which are appropriate to a complex number; we can compute its modulus or even its complex conjugate. The final method shows how to use the class and some of its methods.

I assume that all the language used here to describe and manipulate the class is unfamiliar. Each of our classes above is an example of **source code**. This is the language (usually) written by humans, designed to convey instructions to a computer as opposed to the *compiled* version designed to actually execute efficiently. In the remainder of this Chapter we discuss this language, returning in Chapter 3 to talk more about objects.

2.2 Variables

Like mathematics, a computer language can be considered at two levels: the first concern is that a sample must conform to its own rules of construction, or its **syntax**; only then are we concerned about the meaning or the **semantics** of the sample. Thus $n + 1 =$ is syntactically incorrect in mathematics, because $=$ needs something on each side of it; in contrast $n + 1 = n$ is semantically interesting in that it is true for no value of n. In this chapter we introduce some of the syntax of the JAVA programming language. Most of the ideas will be familiar to anyone who has used virtually *any* programming language.

2.2.1 Parking Cars

I want to introduce an everyday problem chosen to map very closely onto the basic parts of a computer language. You may wish to re-read this part as you become more familiar with the language.

> We have two cars called `hisCar` and `herCar` and two parking spaces, called `hisSpace` and `herSpace`. At the start of the exercise, we have `hisCar` parked in `herSpace` and `herCar` parked in `hisSpace`. The aim is to move the cars in such a way that each ends up in their "own" space.

There are some rules which you have probably understood, but which I should make explicit. We only have one driver, so only one car can be moved at any one time; and a parking space can hold only one car.

And so the "obvious" solution; drive `hisCar` to `hisSpace` and then drive `herCar`[2] to `herSpace`, fails. We have to come up with a **method** of solving our problem. It looks as though we have no choice but to create another parking space. I'll give it a name, say `tempSpace` to indicate we only need the space temporarily. Now we can solve the problem. The method is to move `hisCar` to `tempSpace`; then move `herCar` to `herSpace` and finally move `hisCar` to `hisSpace`.

Let me analyse this situation a little more closely. We have two cars - you may wish to think of them as **values**, and two value - holders, usually called **variables**. Our aim was to swap the values of the two variables. This is exactly what our method did, but to do so we needed a spare or temporary variable. Notice also that the method is generic; although if worked for `hisCar` and `herCar`, exactly the same method would work for `hisDaughtersCar` and `herSonsCar`.

The data for each of the classes above was held in **variables** like `width` or `realPart`. The name, or strictly the **identifier** of a variable must start with a letter and is then made up of a combination of letters and digits. There are many special characters, such as "+" and "." which are not allowed in variable names, although some other characters, such as the underscore "_", are acceptable. Case is significant, and by convention, variable names begin with a lower case letter. Thus `rectangle` may be a very good variable name to hold a particular `Rectangle`; the convention then helps to distinguish between a class and objects which are members of the class.

Essentially all computer languages use variables to store data but they vary in how strict they are about what you can do. JAVA is deliberately strict; it aims whenever possible, to catch mistakes when they are first made rather than later when code is running. Here are two ways that this philosophy affects variables:

- a variable must be declared before it can be used; and
- a variable can only hold one type of object.

[2]By now a mangled wreck!

By the first of these rules, the first time a variable name appears it has to be in a **declaration**. The form is simple:

```
int myName;
Complex z;
double aVeryLongVariableName;
double legal_names_like_this_are_avoided;
```

Here the variable `myName` is **declared** to have type `int`, the variable `z` is of type `Complex`, in other words `z` can (only) hold an object of the `Complex` class, while the remaining two variables are of type `double`. Notice that each statement ends with a **semicolon(;)** to indicate that the statement has finished. If the statement is long it can be continued on the following line and there is no need to do anything, other than omit the semicolon, to indicate this.

How long does a variable declaration last? Where can you use the variable you've declared? These are questions about the **scope** of the variable. One crucial point about the scoping rules is that you *can't* use a variable before it has been declared. There is more detail about this in Section 3.3.1.

2.2.2 Types

The data in the `Rectangle` class, namely the `width` and `height` were both declared to be `double`s. This is a **primitive type** in the sense that such a variable is not made with data from simpler types. There is fairly large choice of primitive types; here are the ones you are most likely to use.

int: an `int` is used to hold an integer like "7" or "-273". It is represented in the machine in a storage location which can hold just 32 bits. We discuss the representation in more detail in Section 5.1.1; it means that an `int` must lie between -2^{31} and $2^{31} - 1$.

long: sometimes the restricted range of an `int` is just too small; JAVA has a second type of integer each of which takes up 64 bits of storage. A `long` can hold any integer between -2^{63} and $2^{63} - 1$.

double: a `double` is used to hold an approximation to a real number. The representation used is described in Section 5.1.2; the resulting number lies between-1.7×10^{308} and 1.7×10^{308}.

boolean: a `boolean` is designed to hold the result of a test. It can have one of the two special values written as `true` and `false`.

We need to be able to handle letters and words as well as numbers. The `char` is the primitive type designed to hold a single character. However it is used very little in practice; JAVA class `String` bundles characters together is a helpful way and is used very much like a primitive type. We‘ve already met the `String` "Hello world." in the program on page 7 and in the first Practical. We discuss Strings in more detail in Section 6.2.

As a convenience, when I want to talk about *either* an `int` *or* a `long`, I will speak of an **integer variable**. As well as `double`s, there is also a `float` type (using half the storage space) and even smaller ones. In the same way I will refer to any of these types as a **real variable** when I don't need to distinguish between them.

2.2.3 Assignment

As soon as a variable has been declared, it can be given (or **assigned**) a value. Here are some examples; you can choose whether to declare and assign on separate lines, or do them both on the same line. If a variable is not of a primitive type, then a value needs to be constructed to assign to the variable. It will come as no surprise that this is what the constructor method within the class is used for.

```
int n;
n = 6;
double x = 1.0;
double y = 1.414;
Complex w = new Complex(x,y);
```

In some circumstances, if you don't give a variable a value, it will be given a default value (numerical variables are set to zero, a boolean to false); in other circumstances it is simply an error to use such an **uninitialised** variable. The rules are made precise in Section 3.3.1. We mentioned the scope of a variable above. In the example the variables `x` and `y` have no connection with the variables of the same name declared within the `Complex` class; they have a completely separate existence; again more details are coming.

The value of a variable can be changed after it has first been assigned, although its new value must always be of the same type as the old one. Here is an example in which the variable `k` ends up with the value 9;

```
int k = 17;
k = 4;
k = 2*k+1;
final int theAnswer = 42;
```

You can see from this example that the $=$ sign used for assignment is very different from the $=$ sign used in mathematics. In assignment, the right hand side can be any expression which gives a result of the correct type, while the left hand side must be the name of a variable which is to store that value. Note that k is declared as an `int` just once; it is an error to do so more than once, or to try to change k to another type like a `double`.

Constants Sometimes it is useful to insist that a variable has a fixed value. The `final` keyword tells the compiler that the "variable" cannot be changed; that it is a constant. In the example above the variable `theAnswer` is fixed at 42. Predefined constants that you may use include π, called `Math.PI` and e, called `Math.E`.

2.3 Expressions and Operators

In the last example, I slipped in the expression `2*k + 1`; now we discuss them more carefully. You are familiar with the idea of a mathematical expression, made up from a number of simpler terms or variables. A computing **expression** is built from simpler expressions or variables using the allowed operators. In this section we look at the various operators and show how they can be combined to generate expressions. In general (but not an the left hand side of an assignment) an expression may be used anywhere in place of a variable.

2.3.1 Arithmetic Expressions

It is relatively simple to do arithmetic on numbers. Let me suppose that `a`, `b` and `c` are variables that of the same (numerical) type — `int`s or `long`s or `double`s and so on. Then the following make sense and probably do what you expect:

Addition: `c = a + b`, so "+" *adds* two variables;

Subtraction: `c = a - b`, so "-" *subtracts* one variable from another;

Multiplication: `c = a * b`, so "*" *multiplies* two variables;

Division: `c = a/b`, so "/" *divides* one variable by another;

The last one isn't quite straightforward. Of course you can't divide by zero. If you try it, JAVA sets the result to "`Infinity`" or "`-Infinity`". If you try to manipulate with `Infinity`, perhaps subtracting it from itself, the answer you get is "NaN", short for "not a number". A second problem occurs when both `a` and `b` are `int`s or `long`s. You almost certainly want the result to be of the same type, and that is what JAVA does. So `14/2` has the value `7` as you expect, but `14/3` has the value `4`; the rule is that division is done *with a remainder* and then the remainder is thrown away.

This opens the door for another operator the *modulo* operator, written as `%`. We have `14%3 = 2`; in other words this gives the remainder that was thrown away by the `/` operator. The result of `a%b` is clear if both `a` and `b` are positive. If either is negative, you may need to experiment.

> *The result is well defined, and I could specify the rules. However if I include that sort of detail, these notes will not only become far too bulky, but will also make it much harder to see the crucial points. If you need to know, either set up some quite simple* JAVA *experiments or go to one of the* JAVA *books where all the details are given.*

As we saw in Section 2.2.3 a value, perhaps computed using arithmetic operations, can be assigned to a variable. This means that the example we've already seen:

```
k = 2*k + 1;
```

which is mathematical nonsense, makes perfect computing sense. It is an assignment, in which the right hand side is computed using arithmetical operations, and then assigned to the variable `k`. The fact that `k` appeared on the right hand side is almost irrelevant. In the same way `n = n + m` simply increases `n` by `m`.

2.3.2 Short forms

The form `n = n + m` in which we increase `n` by `m` can also be written as `n += m`, thought of as "add m to n. With variable names like `m` and `n` there seems very little difference in the two forms, but when the variable names are long there is a significant gain in using the short form. Other arithmetic operators have the same short form, so `n -= m`, `n *= m` and so on also make sense. Indeed there are a number of other operators you can use in the same way. If you need one, go to the books or construct an experiment.

A further abbreviation can be used in a very common situation, where we wish to **increment** or **decrement** a variable by 1. Thus `n++` is the same as `n += 1` or `n = n + 1`. It has the advantage that it can be used within an expression; the change in value occurs as a side effect of using the variable. As you might expect, `n--` decrements `n` by 1 in the same way.

We can think of `n++` as saying "use the value of `n` and then increase it by 1. Sometimes it is more convenient to use the incremented value; This is done by writing `++n`, which *first* does the increment and *then* delivers the new value of `n`. As soon as we meet `for` loops, most of these shorthands will prove useful. However, like any convenience, it is for you to judge whether you find it valuable.

2.3.3 Precedence

We are going to meet a number of operators like `-` or `/`. Here is a general question: suppose α and β are two such operators; how does JAVA interpret $a\alpha b\beta c$? I will only answer this question in part: `+` and `-` have equal precedence, and so are evaluated "left to right"; but are only considered after `*`, `/` and `%` (which have higher precedence) have been evaluated. In other words operators are considered in order of precedence, with operators having equal precedence being considered in "left to right" order. This means for example that

```
6 - 4 + 1/2 + 3 * 2
```

has the value `8`. First `1/2` (remember that this has the value 0 since the operator forces the answer not to contain fractions) and then `3*2` are calculated; the results are then substituted and the rest of the line computed. We shall meet more such operators; a **precedence** is always defined so the result is unambiguous. But unless you are an expert, it may not be what you *intended*. You can always achieve the result you intend by using (round) brackets, known formally as **parentheses**. Thus

```
6 - (4 + 1/(2 + 3))*2
```

may not be sensible, but it is unambiguous, and has the value `-2`. You can think of this as saying that brackets "bind most tightly" in that they always win a "precedence fight". It is never wrong to add brackets, although `(((1 + 2) + 3) + 4)` might be regarded as an excessively fussy version of `1 + 2 + 3 + 4`. More importantly, if things aren't behaving as you expect, try adding brackets to check you are getting the interpretation you wanted.

2.3.4 Coercion

We've already seen an interaction between the normal ideas of arithmetic, and the needs of a computer language to have different types of numerical variables; that there is good reason for `1/2` to have the value `0`. So what happens if we want to get the value `0.5`?

There would be no problem if we had written `1.0/2.0` since this would automatically pick up the (in fact double) form of the division operator; the numbers 1.0 and 2.0 are taken to be of type `double`. More generally the expression `1.0/2` looks like a problem because we are mixing types. In this situation there is really only one useful thing to do — to convert the "2" into a `double`, when the expression again gives the answer 0.5 as we expect. This process of converting between types is known as **coercion**. In general it happens *only* in situations when you would expect it to happen.

You will find coercion can happen much more generally when we look in greater detail at objects.

So how can you force a change of type? In other words how do you turn a `double` into an `int`? there may be good reasons for wanting to do this. For example you may wish to compute the cost of an item, including $17\frac{1}{2}\%$ VAT giving the answer in £s, and ignoring the pennies and fractions of a penny. One way to do this uses a **cast** to turn the total cost, as a `double`, to an `int`.

```
double d = 23.4;
double vat = 0.175;
int pounds = (int)(d*(1 + vat));
```

Here we force (or *cast*) the `double` to be an `int` with the `(int)`. Casting tends to be permitted whenever it is sensible; we say more about it in Section 3.4.3. The compiler will alert you if you are trying to cast when it isn't allowed, such as trying to cast a `boolean` to an `int` or vice-versa.

2.3.5 Boolean Expressions

The `boolean` data type has already been mentioned; you may be surprised how useful it proves to be. It is a primitive data type which can only take the values `true` or `false`. It is *not* a numeric type and can't be coerced into one. Boolean expressions can be constructed from boolean literals (ie `true`, `false`), expressions that yield boolean values and the **logical operators**.

One obvious source of `boolean`s is when we compare two numeric variables. The result is a boolean value, and so can be assigned if necessary to a boolean variable. Here the most useful comparisons:

```
boolean result;
result = (k == n);
result = (k != n);
result = (k < n);
result = (k > n);
result = (k <= n);
result = (k >= n);
```

The first comparison is true iff `k` and `n` have the same value, while the second is true iff they have different values. The other comparisons have their obvious meanings. The first comparison shows the difference between the assignment operator `=` and the comparison operator `==`.

It is easy to confuse assignment "`=`" and comparison "`==`"; if you do confuse them, you may find the error messages from the compiler are puzzling.

When you are thinking of using `==` you should note the lessons of Section 5.1.2. We don't necessarily expect 100 tenths to be the same as 10; we know that real numbers are only represented approximately. This means that it is almost never sensible to do a comparison of real numbers using `==`. Exact comparisons are best done using integer types.

2.3.6 Logical or Boolean Operators

Boolean values may only be combined into expressions using the **logical operators**. The two most important are **and**, written as `&` and **or**, written as `|`. Each takes two arguments; `a & b` is true when each of its (boolean) arguments is true, while `a | b` is true when *either* argument is true. In addition there is **negation**, written as `!`; thus `!a` is true iff `a` is false.

These operators give the facility for writing quite complicated tests, whose individual components may involve other operators, including the arithmetical ones, such as

```
result = ((k > 8 & (n + k < 4) | result) & !(k == 3));
```

You are *very* strongly advised to use brackets to make your meaning clear, rather than rely on the in-built precedence rules.

There are variants of the `&` and `|` operators. Suppose you are evaluating `a & b` and you find that `a` is false. At this stage you already know that `a & b` is false, so there may be no need to evaluate `b`. But evaluating `b` may have **side effects**; for example it may contain the expression `n++`. The programmer needs to be in control of whether logical operators **short circuit** during evaluation, and so in addition to `a & b`, which guarantees to evaluate both arguments, there is the short circuit variant `a && b` which stops evaluating as soon as the result is determined. In the same way `a||b` short circuits as soon as it can.

The final operator you may meet is the **exclusive or**, written as `^`; the expression `a^b` is true if exactly one of `a` and `b` is true.

2.1. Exercise. Assume you have three real numbers (`double`s) called `x`, `y` and `z` which describe the position of a point in $\mathbb{R}^3$. Write a boolean expression which is true exactly when the point lies within the tetrahedron with vertices at the origin and the points $(1,0,0)$, $(0,1,0)$, $(0,0,1)$. You need to start by doing a small amount of geometry to get a convenient description of the tetrahedron.

Solution: This is available in the html version of these notes.

2.3.7 Testing

At this stage you will want to be able to try things out to see what works; armchair programming isn't too useful! But the "outer" structure of a JAVA program is a little complicated, as we saw with the "Hello World" program on page 7. In fact that structure is all that you need. Here I show it again, this time with a different name so it doesn't interfere with the "Hello World" example itself. The content is there simply to show how you can do your own testing; simple manipulation, printing the output to the screen.

```
public class Test {
    public static void main(String[] args) {
        // Replace the next four lines with *your* tests.
        double a = 1.0;
        double b = 1.0;
        double sum = a + b;
        System.out.println("a + b = " + sum);
    }
}
```

Save this to a file called `Test.java`, compile it with `javac` and run it with `java`. Note the use of the `+` in the "println" statement. The first is within this string, so you get what you have asked for. The second is the **String concatenation**; it is the way to print more than one thing with one statement.

2.4 Flow Control

In order to write non-trivial programs it is necessary to decide during the program, based on its state at that time, what to do next. If they receive input, and almost all real programs do, they should be able to take action based on the input. An obvious example is an ATM, which should properly refuse to pay out if the account is overdrawn without some form of overdraft arrangement. In some circumstances we can specify a 'catch all' safety net which we will learn about that in Chapter 7; but in general we need to make decisions based on what is available. Such decisions manage the **flow** of control in the program.

2.4.1 Deciding: the "if" Statement

The simplest control structure makes a decision based on the value of a variable. Here is how it is written:

```
if (n < 0) {
    System.out.println("n is negative");
}
```

It behaves exactly as you would expect. The variable to be tested has to be a `boolean` variable, so must hold the value `true` or `false`; and this decides whether the next *statement-block* is executed or not.[3]

There are more elaborate forms of the test; here is an obvious extension:

```
if (n < 0) {
    System.out.println("n is negative");
} else {
    System.out.println("n is non-negative");
}
```

This could be replaced by two consecutive "`if`" statements but the "`else`" is probably easier to understand. There is even a form suitable for "multi-way" tests; in the next example, you can have as many `else if` steps as you need to get the logic right.

```
if (n < 0) {
    System.out.println("n is negative");
} else if (n == 0){
    System.out.println("n is zero");
} else {
    System.out.println("n is positive");
```

[3]Strictly an `if` test can be followed by a *single* statement and the { } brackets can be omitted. You may think this is convenient if you only want to execute one statement in the block; there are quite strong reasons why you should ignore this convenience.

```
        }
```

These are simple examples, but let me emphasise the structure: based on the value of a `boolean` variable, which as we saw in Section 2.3.5 can be built up from a number of components, we choose whether to execute one or more statements.

The syntax of the `if - else` test is as follows:

```
        if (<condition1>) {
            <if-body1>
        } else if (<condition2>) {
            <if-body2>
        } else if ...
            ...
        } else {
            <else-body>
        }
```

Here each `<body>` is just a statement block; one or more statements which are to be executed if the corresponding test is true. The `else` part is optional; the corresponding `<else-body>` is executed if it is included. And if there is an `else` part, there may be one or more additional `else if` parts. Necessarily only one of the statement blocks is executed.

2.2. Exercise. You are to print the statement “This is water.” You have a `double` available called `temperature` which you can suppose is measured suitably. Nothing is to be printed if the matter is ice, or steam.

Solution: This is available in the html version of these notes.

2.4.2 The Ternary Conditional

Suppose you want to examine one variable and set another variable based on the value you find. As an example, suppose given an integer n you wanted p to have the value $|n|$, the modulus of n. We can easily do this with an “`if - else`” block as follows:

```
        if (n > 0 ) {
            p = n;
        } else {
            p = -n;
        }
```

This happens so often that there is a special syntax, sometimes called the **ternary conditional** which is designed for just that occasion. Here is how to rewrite the $|n|$ example:

```
        p = (n > 0)? n : -n;
```

The syntax of the ternary conditional is as follows:

```
        (<boolean>)?<true_expression>:<false_expression>
```

Here `<boolean>` is the `boolean` value that is tested. If it is true, the result is set to the `<true_expression>`; otherwise it is the `<false_expression>`.

It works by first evaluating the boolean expression; if that turns out to be true, then the value of the whole thing (the "right hand side") is taken to be the expression after the "?"; otherwise it is the expression after the ":". It is sometimes worth using this, but you may need to put in more brackets than you expect to force the compiler to interpret things the way you want it to. For example the following

```
boolean whatever = false;
System.out.println("It is " + (whatever?"this":"that"));
```

fails without the brackets round the ternary conditional; the compiler applies its precedence rules and attempts to interpret `whatever` as a `String`. This fails and the compiler stops indicating it has found an error.

2.4.3 Looping: the "`for`" Statement

Very often you will want a program to repeat a section of code a fixed number of times. There is no difficulty about adding up consecutive integers; I'm sure you've not forgotten that

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2},$$

and that there is a very simple proof available. But let me calculate the sum of the first 10 integers. Here is one easy way:

```
long total = 0;
for (int i = 1; i < 11; i++) {
    total = total + i;
}
```

The syntax of the `for` loop is as follows:

```
for (<initialisation>; <repeating_condition>; <increment>) {
    <body>
}
```

Here `<initialisation>` is executed before the loop starts and so, as above, it can be used to initialise variables. The `<repeating_condition>` is tested each time the loop completes, and should produce a `boolean` which is `true` if the loop should be repeated and `false` otherwise. And `<increment>` are commands than should be performed at the end of each trip through the loop. Finally the `<body>` is just a statement block; one or more statements which are to be executed each time round the loop.

2.3. Exercise. Write a loop which prints each of the first 8 positive integers on a separate line on the screen

Solution: This is available in the html version of these notes.

Here is a more complicated example in which both the initialisation and the stopping condition have multiple statements separated by commas. This loop sums the first hundred squares, and does all the calculation within the header of the `for` loop. Indeed the body of a `for` loop is often empty.

```
for (int i = 1, sum = 0; i <= 100; i++,sum+=i*i) {
    if (i == 100) {
        System.out.println("\nThe sum is " + sum);
    }
}
```

There is no reason to have a loop counter as such; it might be more convenient to increment some `double`— perhaps a length which is gradually increasing — but in any test, recall the comments in Section 2.3.5 about exact comparison of real numbers. Of course loops can be nested. Here is an example with one loop inside another which writes out a simple multiplication table.

```
for (int i=1;i<10;i++) {
    for (int j=1; j < 10;j++) {
        System.out.print(" " + i*j);
    }
    System.out.println();
}
```

Note that this time, I *don't* want a newline after each entry has been printed.

2.4. Exercise. The multiplication table above isn't nicely spaced. Use one or more tests to write it out "correctly".

Solution: This is available in the html version of these notes.

2.4.4 Looping: the "`while`" Statement

Sometimes it is more convenient to have the condition "up front" where it can easily be seen. Here is an example, in which we keep adding up squares until the total exceeds a fixed number.

```
sumsq = 0; n = 0;
while (sumsq < 1000) {
    n++;
    sumsq += n*n;
}
System.out.println("Sum of " + (n-1) + " squares < 1000.");
```

With a little rewriting we could have used a `for` loop instead. Choose the construction you think makes your meaning most clear. The syntax of the `while` loop is as follows:

```
while (<condition>) {
    <body>
}
```

Here the `<body>` is just a statement block; one or more statements which are to be executed each time round the loop. If the `<condition>` is `true` the loop is executed, otherwise the body is skipped and execution continues with the next statement.

2.4.5 Looping: the "do while Statement

Here we rewrite the previous example so that the statement block being repeated is done first, and then a test is made. In other words we invert the order of executing the statement block and the test.

```
sumsq = 0; n = 0;
do {
    n++;
    sumsq += n*n;
} while (sumsq < 1000);
System.out.println("Sum of " + (n-1) + " squares < 1000.");
```

Again the choice should be made depending on the most natural way to express your logic. The syntax of the `do - while` loop is as follows:

```
do {
    <body>
} while (<condition>)
```

Here the `<body>` is just a statement block; one or more statements which are to be executed each time round the loop. These are first done and then the `<condition>` is tested. If the `<condition>` is `true` the loop is repeated, otherwise execution continues.

2.5 Methods

We now introduce the idea of a **method**, as the way in which the statements we've just been learning about are grouped together. The simplest type of method is the one I discussed in section 2.3.7, where I described how to test things. That class contains a single method called `main()`, with a list of statements to be done one after the other. When a java class is run, JAVA looks for a method called `main()` and starts executing the statements given in that method. The first statement in the method then determines what happens next, but by default, the statements are executed one after the other.

One of the things that a `main()` method can do is call other methods, and this is the key step to writing programs which are of a sensible length. You saw examples of methods (`area()` was one) in the classes at the start of this Chapter. In other programming languages, these are called subroutines, functions or some such; in any case, they are the building blocks of the language and collections of actions to be "canned" or "packaged" under a single name.

A method can just be a list of commands to execute. Often a method will produce a *value* of some type as a result of its computation; this values is said to be **returned** by the method. And very often a method will need to work on some parameters or **arguments** which are supplied to it. Here is a simple example to show what a method looks like; it simply computes the sum of the first n integers.

```
static int sumToA(int n) {
    int sum = 0;
    while (n > 0) {
            sum += n--;
        }
        return sum;
    }
```

You could check that this does what you expect in two steps:

- add this code to the test class, immediately *after* `public Class Test` and before the start of the main method.
- add a line like `System.out.println(sumToA(5));` to the main method to print out the sum of the first 5 integers.

The result will surprise no-one but you have now both *defined* a method and also *used* it in another method.

By now it is probably becoming clear that there are many ways of doing the same thing. You may prefer the next version of our method.

```
    static int sumToB(int n) {
        int sum;
        for(sum = 0; n > 0; n--) {
            sum += n;
        }
        return sum;
    }
```

2.5. Exercise. Write a `static` method to convert temperature in degrees Fahrenheit to degrees Celsius. Write a main method to exercise your conversion method.

Solution: This is available in the html version of these notes.

2.5.1 Method Syntax

It is now time to find out in more detail what is needed to write a method. The syntax of a method is as follows:

```
    <return-type> <method name> (<argument list>) {
        statement block;
    }
```

Here the `<return-type>` can be a primitive type, like the `double` in the `area()` method for the `Rectangle`; it can be the same or a different class; or maybe the special word `void` if nothing is returned. The `<method-name>` is just an identifier, while the `<argument-list>`, enclosed in brackets, is a comma - separated list of "type – value" pairs, like the `int n` in the `sumToA()` method, or the `double l, double h` we saw in the `area()` method. There can be more or less arbitrary code within the `statement block`; however, if the method has a (non void) return type, there must be a `return something` statement, where `something` has the required return type.

In fact the above syntax is too simple, there can also be an `<access-modifier>`, like `public` at the start of the declaration; we'll say more on these in Section 3.5.1, and this can be followed by the word `static` if it is a static method; this is discussed in Section 3.1.3.

In English, what this is saying is as follows:

- A method can be given a list of parameters to work on. Each parameter must have its *type* specified as well as a name. It can be much easier to understand and write methods if the parameter names are functional ones, like `height` and `width`.

- Normally this list of parameters is the *only* way that a method interacts with its surroundings, so it is easier to test it and get it right without worrying about other things.

- A method needn't return anything (it has type `void`), but if it does so, it can only return one thing, and the type must be declared as part of the definition of the method. You will see when we look more carefully at classes that this isn't the limitation it seems.

2.5.2 Recursion

It is probably clear that a method can call other methods. It is less obvious that it is useful (or even permitted) to allow a method to call *itself*. This idea of a method calling itself is known as **recursion**; it is very similar to mathematical induction.

For example to find the sum of the first n integers we can simply add n to the sum of the first $n-1$ integers; used inductively, that is one way to prove the formula is correct. Recursion is available in JAVA; here is a recursive way to add integers:

```
public static long SumTo(int n) {
    if (n<=1) {
        return 1;
    }
    return n + SumTo(n-1);
}
```

There are two parts to the recursion.

- if $n = 1$ we can sum the first n integers and the answer is 1.

- otherwise we add n to the sum of the first $n-1$ integers.

The first part ensures that the recursion is **based**, while the second part is the **inductive** step. In fact, just to be on the safe side we immediately stop the recursion if for some reason the method has been called with a negative number or 0 as argument.

Recursion can often be an inefficient use of machine resource, but can be *very* efficient in terms of programmer time and comprehension.

2.6. Exercise. The Fibonacci sequence is defined as follows:

$$a_0 = 0, \qquad a_1 = 1, \qquad a_{n+2} = a_{n+1} + a_n \quad \text{if } n \geq 0.$$

Write a recursive routine to generate the sequence.

Solution: This is available in the html version of these notes.

2.5.3 Example: the Highest Common Factor

Calculating the highest common factor of twointegers provides a very good example of the use of recusrion. Let $m = kn + r$ be a decomposition as integers, and suppose that $p|m$ and $p|n$; then clearly $p|r$. In particular the highest common factor of m and n is the same as the highest common factor of n and r. Now assume that m and n are both positive, that $m > n$ and that r is the remainder when m is divided by n as integers. Calculating $\mathrm{hcf}(n, r)$ is less work than the original problem. Then, on the simpler problem, we can use the same idea. Eventually one of the remainders will be 0, at which point we've found the $\mathrm{hcf}(m, n)$. The code is equally simple.

```
public static long hcf(long m, long n) {
    if (m == 0) return n;
    if (n == 0) return m;
    return hcf(n,m%n);
}
```

In fact there is no point in identifying the larger of m and n since getting it wrong just uses one more recursive step. And if either or both m and n is negative the algorithm still works usefully.

Here is how this method could be incorporated in a class, and thus run.

```
public class RecurseHCF {
    public static long hcf(long m, long n) {
        if (m == 0) return n;
        if (n == 0) return m;
        return hcf(n,m%n);
    }
    public static void main(String[] argv) {
        int n = 4567;
        int m = 23412;
        System.out.print("The hcf of " + m + " and "+ n);
        System.out.println(" is " + hcf(m,n) + ".");
    }
}
```

Chapter 3

Java Classes

3.1 Variables and Methods

Almost everything up to this point apart from Section 2.1 is true of languages like FORTRAN which go back quite a long way. In this Chapter we get to the heart of JAVA by introducing the notion of a **class** and the **objects** which are instantiations of the class. I'm going to start with the `Rectangle` class we first met on page 10.

```
public class Rectangle {
    // Data for the class
    double width;
    double height;
    // A constructor
    Rectangle (double w, double h) {
        width = w;
        height = h;
    }
    double area() {
        return width*height;
    }
    boolean isLandscape() {
        return (width > height);
    }
    // More methods here perhaps?
}
```

I hope that after Chapter 2, most of the low level syntax is familiar to you. The file starts with a **declaration**; that it will describe a class called `Rectangle` which is `public`. This is an *access modifier* which says the class can be freely used. We discuss such modifiers further in Section 3.5.1. Almost always there is single public class[1] in a file; the name of the file is then forced by JAVA to be the same as that of the class; in this case, the file must be called `Rectangle.java`

Any class contains just two sorts of things: class or **instance variables**, sometimes called **data members**; and **methods**. In this case there are just two instance variables,

[1]The only alternative is to have *no* public classes.

the length and height which are declared as you would expect. We are prepared to have many different rectangles; these variables will enable each one to store its own length and height. For this reason, variables like this are called **instance variables**; different objects, or instances of the class can have different values. This contrasts with *static variables* which we discuss in Section 3.1.3.

The rest of the class consists of **methods**, sometimes called **instance methods**. In other programming languages, these are called subroutines, functions or some such; in any case, they are the building blocks of the language and say what actions can be performed. Typically a method is a chunk of source code which can be given values as arguments, and which may (or may not) return a *single* value or object.

In our example, the area() and isLandscape() methods should seem straightforward. The other method, Rectangle(), is distinguished because it *doesn't* have a return type, and it's name is the same as the name of the class itself. It is called a **constructor**, and its job is precisely that; to set up the data for a new object of the class.

Here is an additional method, which would normally appear just before the end of the class:

```
public static void main(String[] args) {
    Rectangle r;
    r = new Rectangle(1.0,2.0);
    System.out.println(r.height);
    Rectangle s = new Rectangle(5.0,3.0);
}
```

This is special in several ways. It is static, which means it isn't associated with a particular rectangle. In addition, the main method is required to have the argument list shown and the void return type. But the most important point is that when (an appropriate form of) the java command is used to run the Rectangle class, execution begins in the main() method.

The first step in the main() method is a declaration; r is a variable that is to hold an instance of the Rectangle class, and nothing else. In the next line, an **instance** is created using the new operator by applying our (only) constructor. This process of **instantiation** is essential: and when it is done, we can print out the height of rectangle r as in the following line; its value is 2.0. In other words, our new rectangle now exists and has its data properly initialised.

Often the declaration and instantiation of an object is done in a single statement like the final one in the fragment above, where another rectangle s is created. And as you expect, this is a *different* instance of the Rectangle class; the height of s is 3.0.

It is simply a convention, but one I strongly suggest you stick to, that class names start with an upper case letter, while variables and method names usually start with a lower case letter. A common idiom is

```
Rectangle rectangle = new Rectangle(1.0,2.5);
```

which looks redundant at first sight. I hope it appears more reasonably now.

We also met the Complex class earlier (Page 11). Take another look at it; you should find that it follows exactly the same pattern as above. Note in particular the main() method in which objects are created and the methods exercised.

3.1.1 Object Oriented Programming

Much of the rest of this Chapter is devoted to showing *how* you can use classes, explaining appropriate syntax and so on. We need to do this to get to the “proper programming” part of the course where we tackle some “real” problems.

But before starting this let me try to give a preview of the underlying philosophy of object oriented programming. Briefly this argues that programming is easiest if specific parts of the program to be written map quite tightly to specific parts of the problem. In the Object Oriented Programming (**OOP**) paradym which we are considering, this is done by identifying objects in the problem and methods by which these objects interact.

One way to analyse a problem before starting to code is is to write out a clear natural description of the problem. Starting with this description, one next underlines all the nouns; this list probably contains all the natural candidates for the objects; doing the same thing for verbs gives the methods.

I will illustrate with an example you are familiar with; suppose the Mathematical Sciences Department decided to process student marks with a JAVA program. For example in course MX3015, there are two assessments each equally weighted, which contribute to the final grade. In one of those assessments, four individuals marks are combined to produce the overall result of the assessment. And so on.

You are quickly led to a number of classes; the Course (eg Hons maths), the Module (eg MX3015), the Assessments and so on. In fact some of the interactions between classes give rise to other classes; an Enrolment (a particular student signs up for a particular module), and even a Mark (typically associated with an enrolment). Even the value of the mark isn’t simple. At first sight you might expect an integer in the range [0..20]; but there are other marks such as Medical Certificate, No Paper, Good Cause and even “not yet assessed”. And since marks are ultimately rounded up, there is need to maintain intermediate calculations accurately, by using rational rather than real numbers.

Having identified the the classes, the next task is to identify methods; an `Assessment` might have a method `getMarkList()`; in order to check a mark, the `Mark` class might have a `boolean earnedBy(Student s)`, and so on. And the whole process would evolve as such natural methods and objects are gradually written and allowed to interact with each other. I won’t go further into this example; however a simpler example simulation a queue is worked out in full in Section 10.9.2

3.1.2 Accessing Instance Variables

We can work with instance variables in two different contexts:

inside the class where they are declared, in which case the simple variable name, like `height` can be used; or

outside the class or in static methods, in which case we need to say which variable we want by adding the instance name; if `r` is a `Rectangle`, we talk about `r.height`.

This makes sense; if there are different instances of class `Rectangle` available, we need to say which rectangle we want the height of. In contrast, when defining (instance) methods within the class, we are likely to be talking about *this* Rectangle.

In fact that use is acceptable; within the class Rectangle, an instance variable can be qualified for example as this.height. Some people, as a matter of style, *always* write instance variables in this way when used within the instance methods of the class.

3.1.3 Static Variables and Methods

However it doesn't always make sense for each object from a class to have their own copy of a variable. Suppose we had a convention that the variables of class Rectangle were measured in cm. We might want the following two variables available; for example to calculate whether a rectangle would fit on a sheet of A4 paper.

```
static double A4Width = 21.0;
static double A4Height = 29.7;
```

But the values are the same for all rectangles; to distinguish this type of variable from instance variables, they are called **static variables**.

In the same way we can have **static methods**; those that don't need a particular instance of the class to work on. The static method A4() below simply creates a rectangle of fixed dimensions. You would refer to the resulting object as Rectangle.A4().

```
static Rectangle A4() {
    return new Rectangle(A4Width, A4Height);
}
```

Another example may help to show the distinction. It is from our Complex class. The first of is an instance method, in that this complex variable is added to the argument w of the method. In the second, static method there is no this complex number around; instead the two arguments are added.

```
Complex add(Complex w) {
    return new Complex(x + w.x,y + w.y);
}
static Complex add(Complex z, Complex w) {
    return new Complex(z.x + w.x, z.y = w.y);
}
```

I hope the distinction is becoming clearer and that the naming conventions seem natural. Although it isn't *required*, it makes sense to initialise A4Width when it is declared. It may make less sense to do so for instance variables, although, as above, it might be useful.

Static variables are referred to just like instance variables within the class. Outside the class we don't need to create a fictitious Rectangle r to talk about r.A4Width; instead Rectangle.A4Width works.

An example you've already seen is Math.PI. The Math class is in the package java.lang. The class has no objects at all; but you can probably see why that was appropriate.

3.1.4 Accessors and Mutators

I've presented variables and methods as the two components within a class. I now want to change viewpoints and introduce **accessor** and **mutator** methods. These are designed to ensure that an instance variable is never used directly; instead access is only done through methods. Here are the accessor or "get" methods for our Rectangle:

```
        double getWidth() {
            return width;
        }
        double getHeight() {
            return height;
        }
```

All they do is deliver a copy of the variable; it is made available in a method rather than used directly. The mutator, changing, or "set" methods, are used to replace the current value by a new one. They are equally simple:

```
        void setWidth(double w) {
            width = w;
        }
        void setHeight(double h) {
            height = h;
        }
```

Why bother with this inconvenience?

It is a general principle when designing classes that the user of a class (by which I mean any method which manipulates particular objects) should not be concerned about the internal design of the class. This is often called **information hiding**; that you should only make use of the properties of an object that you need, and not accidents of a particular design.

Such restraint is very useful if you need to change the design of a class after it has first been used. Perhaps it turns out to be better to rotate every rectangle so it is in portrait mode, and record whether this was necessary in a separate instance variable, perhaps `boolean flipped`. Thus if we are trying to create a landscape rectangle, the height and width would be interchanged and the variable `flipped` set to be `true`. If accessor and mutator methods have been used consistently, we need make no change to any code outside the class; otherwise we must hunt through the code and replacing every reference to `height` by `flipped?width:height`. Of course we still have to make sure that other methods *within* the class are correct, but the effects of the change have been restricted. The names I've given to the methods with "set" and "get" are only a convention, but I suggest you stick to them.[2]

3.1.5 Method Overloading

Method **overloading** is the ability to define several methods all with the same name. At first sight, that sounds stupid, but there are good reasons to want to. For example you are happy with the idea that $\mathbb{C} \supset \mathbb{R}$; that it is easy to treat a real number as a complex number. Suppose we wish to do so in our `Complex` class. It would seem natural to write a new `add()` method as follows:

```
        Complex add(double x) {
            return new Complex(this.x + x, y);
        }
```

[2]This convention is *required* for JavaBeans.

This is described by saying that the `add()` method has been **overloaded**. Although the name and return type are the same as the one used with a `Complex` argument, the arguments consist of different types and the JAVA compiler has no problem choosing the right one. We refer this combination of the number and types of arguments of a method as its **signature**. The combination of name and signature should be unique. Another example is the `PrintStream` class; you've seen that the `print()` and `println()` methods can take many different arguments.

Although you can overload methods, you *can't* overload operators: `*` and `+` can't be extended to work with matrices as they can in some other languages[3]. You have already seen the single exception to this: the `+` operator *has* been overloaded within JAVA for `String` concatenation.

3.1.6 Creating and Destroying Objects

We met constructors briefly in Section 3.1. A **constructor** is a method that, unusually does not have a return type. Its name must be the same as the name of the class; you can think of this as a hidden return type, since a constructor has to create a new member of the class. Indeed a constructor is the *only* way to create new instances of a class.

So far we've seen fairly simple classes. In general a constructor might initialise certain instance variables to default values, it might construct other classes being held as instance variables, and it might even check that its environment is suitable before going on. Constructors can be overloaded. Here are two examples. The first no argument constructor would be useful if rectangle should default to be A4. The second is what is sometimes called a "convenience" constructor, helpful in this case if we expect to create lots of rectangles whose sides are integers.

```
Rectangle() {
    this(A4Width,A4Height);
}
Rectangle (int w, int h) {
    this((double)w, (double)h);
}
```

The use of `this` is like that discussed at the end of Section 3.1.2; it refers to the class *constructor* in this case. Thus each constructor calls the "normal" constructor with appropriate arguments. This is a general technique. It ensures that we can do all the complicated setup of the class in a single place. For example we only need to change the "normal" constructors if, for example, we subsequently make the change discussed in Section 3.1.4 and introduce `flipped`. However we *are* forced to use another constructor as the *first* statement we give. This avoids conflicts during initialisation.[4]

Object Destruction

An object is actually something called a **handle**, which you can think of as an indirect access to the memory associated with the object. During object construction, memory is allocated and a handle created. It is possible to lose this memory subsequently. For

[3]The most popular is probably C++.

[4]Many times we will be vague like this — go to the books for more detail

example, given that Rectangle r and Rectangle s are both declared, the statement s = r will now leave r and s defining (strictly " a handle to") the same object, whereas there is no handle to the rectangle that used to be called s, unless it has been saved in another part of the program. If this hasn't happened, we refer to the memory that once held Rectangle s as *garbage*; having it around is a waste of resource. Here is an example in which we choose to reclaim this resource:

```
Rectangle rectangle = new Rectangle(1,2);
//  .... some useful code
rectangle = null;
```

It constructs an object of class Rectangle, perhaps does some useful manipulation, and finally releases the storage associated with the object, unless some steps have been taken before that to save it. The special keyword null is a valid handle which is recognised as *not* pointing to an object.

Garbage Collection

To avoid wasting memory, the Java run time system will, from time to time, look for memory it "owns" that has no handle to it. This memory is then freed up for re-use. The process is known as **garbage collection** and occurs at unspecified times - although certainly will occur more frequently when the machine is low in memory. The algorithm used is quite slow and fairly straightforward - all memory "in use" is marked in a special way; that which is not marked is then freed and the marks "released".

Finalization

And just in case, before an object's memory is finally released, its finalize() method is called. It can be used to free up resources that otherwise might remain locked. For most purposes you can forget this facility, but should you ever need it, you will be glad it is available.

3.2 An Example: Points and Circles

It is time to look at how classes can interact. Here are two simple classes which are linked and which use a number of the features we've discussed so far. We start with a simple Point class.[5]

```
public class Point {
    /** The x - co-ordinate of the point. **/
    double x;
    /** The y - co-ordinate of the point. **/
    double y;
    /** One constructor needs the co-ords of the point. **/
    Point(double xx, double yy) {
        x = xx;
```

[5]You will find both this and the next example in the "samples" directory.

```
            y = yy;
        }
        /** A convenience constructor - it just casts. **/
        Point(int x,int y) {
            this((double)x,(double)y);
        }
        /** The <code>Circle</code> class is coming. **/
        boolean isInside(Circle c) {
            return (distanceFrom(c.centre) < c.radius);
        }
        /** Compute the distance from another point **/
        double distanceFrom(Point p) {
            double distanceSquared;
            distanceSquared = (x -p.x)*(x-p.x) + (y -p.y)*(y - p.y);
            return(Math.sqrt(distanceSquared));
        }
        public String toString() {
            return ("(" + x + "," + y + ")");
        }
    }
```

Most of this is routine. We have very simple constructors. The only point to note is the `toString()` method. Any object with a `toString()` method can be passed to the `println()` method eg of the `PrintSteam` class `System.out`. Note there is no `main()` method. It isn't required, although it is often included for testing. In our case the `Point` class is to be used in other classes, and in particular a `Circle` class.

```
    public class Circle {
        /** The center of the circle. **/
        public Point centre;
        /** And its radius. **/
        public double radius;
        /** The main constructor needs to know about all the data. **/
        Circle(Point p, double r) {
            centre = p;
            radius = r;
        }
    }
```

Here we see how it all comes together, and how natural it is to treat each point and each circle as an *object*. The code is broken into a number of very small pieces, each of which is very easy to check.

3.3 Variables

We now say more about variables; how they get values, how long they keep their values and who else can use them. We start with a new type, which is neither a static variable nor an instance variable, nor even an argument to a method.

3.3.1 Local Variables

It is often useful to introduce variables within a method to do some computation. Perhaps we have a polygon class and want a method to return the length of the longest side. Here is one way [6] to do it:

```
double getLongestSide() {
    double longest = side[0];
    for (int i = 0; i < side.length; i++) {
        if (side[i] > longest) {
            longest = side[i];
        }
    }
    return longest;
}
```

Variables like `longest` or `i` above are called **local variables** and have no connection with other variables in the class, even if they have the same name. A variable of any type has an associated **scope**. To over simplify, scope is defined by enclosing (curly) brackets. It is most easily described for local variables because their scope is limited. Thus `longest` is not known (is out of scope) until it has been declared; it is then available until the end of the method, when it goes out of scope, so is no longer known. The variable `i` has a very restricted scope although it follows the same rules; it is only known within the `for` loop.

What happens if there is an instance variable called `longest`? In general, variables with a local **scope** can *shadow* variables with a more global scope. This description is because the variable that is shadowed can no longer been seen while the local version is in scope. They then go out of scope, their storage is reclaimed, and the shadowed variables are again visible as if nothing had happened.

3.3.2 Shadowing

We discuss in more detail what happens if there is a local variable with the same name as an instance variable. Here is an example which illustrates several points. To make it convincing, assume we have a specialised application with many point arranged in clusters. Each point has to remember (ie store in `cCentre`) the point at the centre of its own cluster. The method `distanceFromClusterCentre()` then computes the distance from a foreign point to the centre of "my" cluster.

```
public class Point {
    double x;
    double y;
    Point cCentre;
    Point(double x, double y) {
        this.x = x;
        this.y = y;
```

[6] although it uses an array, which we don't meet formally until Chapter 4.

```
        }
        /** Use this version and Java initialises y to 0. **/
        Point(double x) {
            this.x = x;
        }
        //  ....
        /** Compute the distance from another point **/
        double distanceFromClusterCentre(Point p) {
            double x = cCentre.x - p.x;
            double y = cCentre.y - p.y;
            return(Math.sqrt(x*x + y*y));
        }
        //  ....
    }
```

Note in passing that there is no objection to having a class, even another example of yourself, as an instance variable. The previous version of `Point` had a constructor with arguments `xx` and `yy`. This time we adopt a more natural usage, but it almost gives a name conflict. The method argument `x` shadows the member `x`, but that was still available to us as `this.x`.

In the `distanceFromClusterCentre()` method, we choose to use `x` as local variable. This did *not* interfere with the instance variable `x`; the instance variable is *not* re-assigned, but simply shadowed. As before it is still available to us as `this.x`.

3.3.3 Variable Initialisation

You may note that JAVA will allows you to use instance variables before you have initialised them. In fact on creation, instance variables are set to something innocuous — typically 0 or the empty String "". Our `Point` example above illustrates this with the 1-argument constructor; by default the `y` value is 0. In contrast it is an error to use a *local* variable before it has been initialised. Note that *declaring* a variable is not the same as initialising it.

This may seem like an odd design choice but it is deliberate. Failing to initialise a local variable is a very common source of programming errors.

3.3.4 Argument Passing

What happens to the variables we pass as arguments to methods? To answer this needs a little understanding of how the compiler deals with a method. Unlike many other languages, JAVA does not change the values of arguments of primitive type. A method can return a value, but not change its arguments. You can think of this as the compiler inserting, as the first few lines of every method, a copy of every argument of primitive type to a local variable with the same name as the argument. This in effect shadows the argument, which thus can't be changed. I hope that makes the behaviour of `i` and `j` in the example below seems reasonable. It is known as passing an argument **by value**.

But such a copy could be expensive if the argument was an object, particularly if it was an elaborate one with lots of storage. To avoid this, objects (and arrays discussed in

Chapter 4) are passed **by reference**. This means that the method is given the (lightweight) handle used to refer to the object. The method thus has access to the original object and can make changes. The handle itself is copied, or passed by value, which mean that any re-assignment does not escape from the method. But any changes made to the original object *are* visible outside the method.

Here then is an illustrative example of passing arguments by value:

```
public class Arguments {
    static int test(Rectangle r, int i) {
        r.setHeight(17.0);
        r = Rectangle.A4();
        i++;
        System.out.println("Test: height is " + r.height +".");
        return i;
    }
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle(1,2);
        int i = 40;
        int j = Arguments.test(rectangle,i);
        System.out.print("The rectangle has height ");
        System.out.print(rectangle.height);
        System.out.print("; i and j have values ");
        System.out.println(i + ", " + j + ".");
    }
}
```

It has as output

```
Test: height is 29.7.
The rectangle has height 17.0; i and j have values 40, 41.
```

Thus the change made to the height of the rectangle whose handle was passed to `test()` becomes permanent. However the handle does not return from the method.

Notice also that it doesn't matter what the arguments are *called*. Sometime we use the same names in the calling method as in the definition of the method itself, but this a mere convenience. The only thing that matters about arguments is their values. Note also that the type is checked at compile time. An error here is a programming error and should be caught long before the code is run.

3.4 Subclasses and Inheritance

Creating classes gives a great deal of freedom when modelling the real world, but by itself isn't enough to impose good organisation. The analogy would be a file structure with no directories or folders. Given a class, we can create **subclasses** which start with everything available to the class and then get more. Here is a simple class.

```
public class GeometricObject {
    /** Often useful to name objects. **/
```

```
        String name;
        /** How close before things are considered equal. **/
        double fuzz = 1e-10;
    }
```

The idea of giving each object a name is routine. And since we are doing geometry in a space where numbers have gaps between them (Section 5.1.2), we need a notion of how close things should be before the difference is ignored. We can now create subclasses which inherit this behaviour in the following way

```
    public class Point extends GeometricObject{
        double x;
        double y;
        Point(x,y){
            ....
        }
    }
```

and we automatically have the `name` and `fuzz` variables available in our `Point` class. In the same way we can modify the `Circle` class to inherit from `GeometricObject` and create more subclasses as needed.

As you would expect, this works just as well for methods as variables; any methods[7] declared in the `GeometricObject` class are available to its subclasses. But they can be declared once and for all.

Subclasses can themselves be subclassed. We might have

```
    public class Line extends GeometricObject {
        private double a;  //ax + by + c = 0
        private double b;
        private double c;
        Line(Point p1, Point p2) {
            // ....
        }
        /** Does the line pass through a point? **/
        boolean passesThrough(Point p){
            return (Math.abs(a*p.y + b*p.x + c) < fuzz);
        }
        Point intersect(Line l) {
            // ...
        }
    }
```

and then a more specific `LineSegment`, as follows:

```
    public class LineSegment extends Line {
        private Point startPt;
        private Point endPt;
```

[7]We shall modify this statement; the full story is given in Section 3.5

```
        LineSegment(Point p1, Point p2) {
            super(p1,p2);
            startPt = p1;
            endPt = p2;
        }
        /** This gives the distance between the two end points. **/
        public double getLength() {
            return endPt.distanceFrom(startPt);
        }
        /** Convenience method for testing. **/
        public String toString() {
            return (super.toString() + " It runs from " +
                    startPt + " to " + endPt + ".");
        }
        /** A brief test of the class, which also demonstrates upcasting
            and downcasting. **/
        public static void main(String[] args) {
            Point v1 = new Point(1,5);
            Point v2 = new Point(0,3);
            LineSegment l = new LineSegment(v1,v2);
            System.out.println(l);
            Line line;
            // There is no objection to the next line
            line = l;
            // But line will still use a LineSegment "toString" method
            System.out.println(line);
            LineSegment lineSeg;
            // The next line is a compiler error without the cast.
            lineSeg = (LineSegment) line;
        }
    }
```

If now we have an instance `lineSeg` of class `LineSegment`, it makes sense to talk about `lineSeg.name` and use the method `passesThrough()`, a method available to all lines as well as the method `getLength()`, a method specific to line segments. We say that `LineSegment` **inherits** the methods and variables from `Line`, and that `Line` is a **superclass** of of the class `LineSegment`.

Note that a class can only have a single parent, which in turn can only have a single parent. In fact the class hierarchy has a single root; every object is ultimately a subclass of the `Object` class. This property of *single inheritance* is a design choice; other languages allow multiple inheritance. In fact the `Object` class has a number of methods defined, two of which, `toString()` and `equals()`, are familiar.

Subclasses are fully - fledged members of their super classes and can always be used in their place. Thus if `line` is an instance of the `Line` class, `line.intersect(lineSeg)` makes sense even though the argument has to be of class `Line`.

3.4.1 Shadowed Variables

You can probably guess how these work. If class LineSegment has an instance variable name, that will *shadow*, or *hide* the variable with the same name in its superclass. We can even introduce an int name in class LineSegment which would block direct access to the String variable. And methods of the superclass would still see the String version. Even when masked, variables can still be seen; in this example, we could refer to super.name if we wanted the String version.

3.4.2 Overriding Methods

We have already seen that methods can be *overloaded*; that we can have methods with the same name and different signatures, the combination of argument list and return types, and the correct one will be used. Here is another idea: you can have exactly the *same* signature in a subclass; the corresponding method is said to be **overridden**.

```
public class Triangle extends GeometricObject {
    // .....
    double area() {
        // Method based on the cross product
        // ....
    }
}
```

Now suppose we subclass our Triangle class:

```
public class RightTriangle extends Triangle {
    // .....
    double area() {
        // Method using half base times height
        // ....
    }
}
```

and our new area method is arguably simpler and so quicker, since in this class our triangles have right angles.

3.4.3 Object Construction

A subclass may wish to invoke a constructor of a superclass. A natural example is the LineSegment shown on page 39 which builds on our earlier superclass, Line. This has a constructor with two points which are used to build the equation of the line. As with a call to this in the constructor, the call to super must be the first statement of the constructor. Note that the Line called line actually gets the LineSegment version of the toString() method. Note also that although l can be **upcast** to a Line without comment, there *is* need of the **cast** in final line of Linesegment. If it is not there, the compiler will complain.

3.4.4 Abstract Methods and Classes

Suppose we want to insist in our GeometricObject that there be a toString() method. Clearly we can't write one, because we don't have the information at that level; yet it could be valuable to know that *every* object of the GeometricObject class has such a method. The abstract keyword does the job.

```
public abstract class GeometricObject {
    /** Often useful to name objects. **/
    String name;
    /** How close before things are considered equal. **/
    double fuzz = 1e-10;
    /** Force all subclasses to have a toString method **/
    public abstract String toString(); // No implementation/
}
```

Given this, any class like Point or Line which inherits from GeometricObject is *forced* to implement this method, unless this class too is declared as abstract. Since it is not possible to instantiate an abstract class, all such methods must have been instantiated before use. There is no objection however to an abstract class having actual (real as opposed to abstract) methods.

3.5 Packages

A **package** is the name for a group of related classes. In our case, we use a different package for each Practical Session. One effect has been to stop things with similar names interfering with each other; you may have noticed that by now we have three different classes called Circle, yet when Point refers to a class Circle there is no confusion about which one we get. In this section we discuss such issues further.

The class uk.ac.abdn.maths.mx3015.classes.Circle should have a (globally)unique name. The name is built from the package name, declared in the package statement, and the actual class name. The standard convention about the choice of package names is exactly the one we have been using; they are build from (backwards) Internet names. Thus maths.abdn.ac.uk is the name of the local domain on the Internet, and we have minimally added to this name to get the actual package names used. Another package name is com.axeon.util, which, as you expect, is associated with the owner of axeon.com. Even system packages like java.lang (almost) adhere to this convention.

When you take account of the requirement that a (public) class shall be stored in a file of the same name, and see the mapping of package names to directories, I hope our naming and storage conventions become obvious. The only non-obvious bit is that a package in a subdirectory is not a sub-package of one in the parent directory — indeed the concept of sub-package does not exist.

But life is too short to type uk.ac.abdn.maths.mx3015.classes.Circle rather than Circle every time we want to refer to a class name. Just briefly we shall refer to these as the *full* name of the class and its *short* name. Our desire to use short class names leads to the need for packages.

> *Any class within a package can be referred to within that package by its short name without ambiguity.*

Since the code for the classes in a given package is stored in files within a fixed directory, the correspondence between class names and file names enforces this uniqueness.

Sometimes we need to refer to classes in other packages. For example the `AddTwoInts` class needed access to the `KeyboardInput` class written to ease your first steps in JAVA.

```
package uk.ac.abdn.maths.mx3015.intro;
import uk.ac.abdn.maths.mx3015.util.KeyboardInput;
public class Circle {
    //  ....
}
```

It was the `import` statement which specifically made available this class from another package. By importing it, we no longer need to use its long name, but can use the short name. Note also that even with a short name available a class can always be referred to by its long name; you could still get access to `uk.ac.abdn.maths.mx3015.classes.Circle` if you wished.

Finally be aware that there is one big amorphous unnamed package. Your first JAVA program, writing "Hello World" was in this package. It is easier to get things going initially using the top level unnamed package, but namespace pollution quickly becomes a problem. You can also see, I hope, why we had to type

```
java -classpath h:/mx3015/classes uk.ac.abdn.maths.mx3015.intro.Circle
```

to run `Circle`. This is normal; although we can use short names when coding, we can't do so when running.

3.5.1 Access Modifiers

Having established these simple rules, it is time to complicate them! One of the principles of good computing design is **encapsulation**; hiding details of how things are done and only exposing a tightly specified interface. The aim of encapsulation in JAVA then is to ensure that as little as possible of the inner workings of a class are accessible from outside.

We've already met this briefly in Section 3.1.4 where I argued that in order to protect a class, instance variables should be manipulated using accessor and mutator methods rather than directly. We are now at the stage where we can *enforce* this requirement using **access modifiers** which specify the visibility of classes, variables and methods. Here is our `Rectangle` complete with (mostly appropriate)[8] access modifiers:

```
public class Rectangle {
    private double width;
    private double height;
    public Rectangle (double w, double h) {
        width = w;
        height = h;
```

[8] I can make the use of `protected` below appear sensible if I explain I'm worried about other users knowing that sizes have to be in `cm` before the `A4` specifications are correct.

```
            }
            Rectangle (int w, int h) {
                this((double)w, (double)h);
            }
            public double getWidth() {
                return width;
            }
            public void setWidth(double w) {
                width = w;
            }
            //  ....
            double area() {
                return width*height;
            }
            protected static double A4Width = 21.0;
            protected static double A4Height = 29.7;
            public static void main(String[] args) {
                //  ....
            }
        }
```

The possible access modifiers and their meanings are given in Table 3.1. We now discuss these in more detail.

Modifier	Effect
`public`	visible everywhere
package	unlimited access within the class
`protected`	available to subclasses
`private`	only available within the *class*

Table 3.1: Access modifiers and their meanings.

“Package”: this is the default; there is no access modifier called “package”. It simply means that with *no* access modifier, everything is visible within the package and nothing is visible outside. Almost everything we’ve done so far has been “package” simply because I was postponing the introduction of the other possibilities.

public: this is the modifier to indicate that the class, instance variable or function can be used anywhere;

private: this is the opposite access modifier, and ensures that nothing outside the class has access; and

protected: this is sometimes needed when creating subclasses. If a subclass is in a different package from its parent, only `public` methods and variables are accessible even within the subclass. Often using public is too open, and protected grants access *just* within a subclass.

Most of the time in your work, you won't need this much detail. Now you know it exists, you can look up the exact details when necessary.

These modifiers apply not just to instance variables and methods but also to the class itself. A class can be public or have default access, since neither protected nor private makes sense. There can be no more than one public class in a file; but there *can* be other classes. Again the details can wait until you need them.

Subclasses

One place where access modifiers impinge on even simple coding is through the rule that the visibility of a method or variable in a subclass can't be less than that in the superclass. The Object class already has a public String toString() method; hence we were forced in GeometricObject to make its method public as well. The rule makes sense; since we can always consider a sub-object as an an object in its superclass, we could always get the added visibility by first casting.

There are also perhaps unexpected complications with protected variables in subclasses. Again you need an accurate description from one of the "big" books if you go near this.

3.6 Interfaces

We have seen it can be helpful to use inheritance. But what happens if we want to inherit from two distinct things? For example both GeometricObject and DisplayObject might have desirable qualities we want things like Line and Point to inherit, if we are thinking of plotting on a screen. A number of languages have tried to implement such *multiple inheritance*, but each has encountered difficulties - typically conflicts can occur and there is a need for setting priorities. As noted in Section 3.4, a class can only inherit from a single super class. However a class can also inherit **Interfaces**. These are very like abstract classes and are essentially a summary of extra properties and variables that are inherited. Here is an example

```
public interface Displayable{
    String defaultColour="white";
    void display(Point position);
}
```

where there is no actual implementation of any of the methods and where any variables declared have to be initialised, and are then final. Any class that implements all the methods of the interface can then declare that it implements the interface.

```
public class Point extends GeometricObject
                   implements Displayable {
    double x;
    double y;
    String colour = defaultColour;
```

```
        Point(double x, double y) {
            // ....
        }
        //  .... and so on
        public void display(Point p) {
            // .... code to display the point
        }
    }
```

For our work, this is certainly *not* a curiosity. We shall need to work a lot with functions, performing tasks such as interpolation and (numerical) integration. So logically we would like to pass such functions as arguments. We *could* try to have a single `Function` class from which all other functions were derived. However there is an easier way, since the most important property of a function we need is that it can be evaluated. Hence the following

```
    public interface Evaluable {
        // The interface abstracts evaluating a function at a point.
        public double at(double x);
    }
```

Thus, to be evaluable, the class simply has to implement the `at(Double x)` method. Here is a natural example:

```
    public class Quadratic implements Evaluable {
        private double a; // coef of x*x
        private double b; // coef of x
        private double c; // constant term
        //  .... constructors etc
        public double at(double x) {
            return ((a*x + b)*x + c);
        }
    }
```

Just like classes, interfaces can be used in a method signature. We shall meet the following later:

```
    static double integrate(Evaluable f, double lowerLimit,
                            double upperLimit, int stepCount) {
        //  .... code here
    }
```

as a method in the `TrapeziumRule` class. We can also have **sub-interfaces** in the same way as subclasses; they are equally useful.

```
    public interface Differentiable extends Evaluable{
        // We can evaluate the derivative of the function at a point.
        public double dash(double x);
    }
```

Chapter 4

Arrays

4.1 The need for Arrays

So far we have looked at variables which hold "something". We started by considering variables which could hold a primitive type; an `int`, a `double` or a `boolean` and these gave us quite an expressive vocabulary.

The next step was to create our own types: mathematicians may well want a complex number to be treated in a very similar way to real variables. The JAVA solution to this problem was to create our own `Complex` class and then instantiate the class; in other words we made examples of it (ie objects), which behaved as complex numbers.

Creating objects in this way gives a very general mechanism for building new "things". However there is still one more facility we need if we are to use variables naturally. This is the ability to provide *indexed* variables. These are natural in maths: we speak of a sequence $\{a_n\}$, perhaps giving a formula such as $a_n = 1/n$ to define each member of the sequence. A finite variant of this occurs when we set up a point or vector $\mathbf{x} \in \mathbb{R}^n$, say $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Another obvious example is a "general" polynomial $p(x) = a_0 + a_1x + \ldots + a_nx^n$. Notice that in the first two examples it was natural to start counting at 1, while for the polynomial it was much more convenient to count from 0.

Let me take as an example the polynomial

$$p(x) = (1 + x)^5 = 1 + 5x + 10x^2 + 10x^3 + 5x^4 + x^5.$$

Here is one way we could set up the coefficients:[1]

```
        int a0=1, a1=5, a2=10, a3=10, a4=5, a5=1;
```

Each of the coefficients is closely related to the others, so it seems sensible to give them closely related names as above. In fact it would be useful to make the names even more closely connected with each other: this is what we now discuss.

In common with most programming languages, the JAVA solution to this problem is the concept of an **array**. An array is an ordered collection of variables of the same type, indexed by an integer which counts from zero. The closest thing in mathematics is a *vector* although formally a vector can only hold things like real or complex numbers; in contrast an array can hold essentially *anything* as long as each element holds the same type of thing.

[1]I normally write one variable per line. This syntax, using the comma "," works and seems helpful here.

In common with mathematics, a restriction with an array is that, before it is used, you first need to know how big it is. Here is how an array is declared, instantiated and assigned in Java.

```
int[] a;         // the declaration
a = new int[6]; // and the instatiation
// you can combine the above writing: int a[] = new int[6];
a[0] = 1; a[1] = 5; a[2] = 10; a[3] = 10; a[4] = 5; a[5] = 1;
```

It is almost the case that an array is an object; in any case each array has a member variable, it's `length` which is correctly initialised during instantiation. Each element of the array, such as `a[2]` is manipulated exactly like an ordinary variable. The gain is that they can all be manipulated together if needed. As an example, here we use a `for` loop to find the sum of the coefficients of the polynomial p:

```
long sum = 0;
for (int i=0;i<a.length;i++) {
    sum += a[i];
}
```

Note that arrays behave more like objects than like primitive types in that, as well as being declared, they also need to be instantiated. It is often convenient to combine both statements. I have used `a.length` in the above in a natural way. Using it means there is much less to change if for some reason you subsequently change the declared length of the array.

4.1. Exercise. Evaluate $p(2.4)$; in other words, compute $(1+2.4)^5$ using the form $1+5.2.4+10.(2.4)^2+..+(2.4)^5$. You should find it straightforward to do so. However there is a trick known as **Horner's method** which rewrites a general polynomial as

$$(...(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + ... + a_1)x + a_0.$$

This arrangement minimises the number of multiplications and additions needed for the evaluation. Use a `for` loop to do the evaluation using Horner's method.

Solution: This is available in the html version of these notes.

4.2 Declaration and Instantiation

I now review the steps to instantiating an array, discussing briefly how Java manages the storage. In the following fragment, we declare an array on the first line, and instantiate it on the second. We then declare and instantiate a second array in a single step.

```
double [] temp;
temp = new double[32];
Complex [] complex = new Complex[24];
```

The first step when we declare `double[] temp` simply insists that the variable `temp` can only contain an array of `double`s. Think of "array of `double`s" as itself a type , and this becomes exactly what you expect. The next step reserves 32 spaces to store the individual `double`s. The second declaration and instantiation reserves, in the same way, space to store

24 (references to) objects of type Complex, the class of Complex numbers we defined on page 11. Note that we get 24 slots, not 24 objects; the individual objects still have to be instantiated in the usual way. Here is an example to make the point First an array of complex numbers is declared and instantiated. Then within the loop, each individual complex number is instantiated and constructed. For simplicity I have given trivial values for the real and imaginary parts; note the **coercion** to provide the doubles needed by the constructor.

```
        Complex[] z = new Complex[50];
        for (int i = 0;i < z.length; i++) {
            z[i] = new Complex((double)i,(double)-i);
        }
```

4.3 Indexing arrays

The length of an array is always an int, and the first element of the array is indexed by 0. This means that complex[23] is the last element I can refer to. In other words, the last index is *one less* than the length of the array. Counting from zero may seem perverse at first, but our polynomial example shows it can be a natural choice.

In some programming languages, it is the responsibility of the user to ensure that complex[24] or the like, which doesn't exist, is not referred to.[2] Even JAVA can't check for such an error when the program is compiled because the index is usually computed when the program is running. However array access is monitored and any attempt to access an element which doesn't exist will be met with an error like:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 24
        at uk.ac.abdn.maths.mx3015.noteexamples.Array.main(Compiled Code)
```

We will meet Exceptions in Chapter 7; I've mentioned them now now so at least you can interpret this error message. It is a fatal error in the sense that it will completely stop the program's execution. Note also that, since this occurs long after the program has been compiled, there is no access to the original source code, so no way to report the line on which the error was made.

4.4 Initialisation

One way to instantiate an array is to declare it with an initialiser. We could have set up our coefficient array above very simply:

```
        int coefficients[] = {1,5,10,10,5,1};
```

Here is another example. We find the mean of a list of values. Note that at no point does the programmer have to count the number of data points. This means it is a trivial change to add another point. I've laid things out as I might if the dataset were significantly larger.

[2]For example neither C nor C++ check array bounds. Exploiting an unchecked array overflow is one of the most common ways of hacking on the web.

```
double data[] = {  1.73,  2.49, 17.32, 14.60, 64.7,
                  12.78, 14.63,  6.23, 49.61, 93.74};
double total = 0.0;
for (int i=0;i<data.length;i++) {
    total += data[i];
}
System.out.println("The mean is " + total/data.length);
```

4.2. Exercise. You are given an array `data` of `int`s, each of which contains one of the digits $0, 1, \ldots, 9$. Write code to count the number of occurrences of each individual digit, storing the results in an array `count`, so that `count[0]`,...,`count[9]` store the number of 0's,...,9's in the original `data`. Assume that `data` has already been initialised, but create the storage array. Then invent 16 values for your `data` and test what you have written.

Solution: This is available in the html version of these notes.

4.5 The Sieve of Eratosthenes

I'm going to show a typical use of arrays by describing a simple way of getting a list of all the prime numbers up to a given limit. The section begins our move from "five finger exercises" to "real" programming in that the analysis involves the combination of three elements:

- some simple mathematics to describe the problem and solution;
- an **algorithm** describing how we can do the calculation; and
- a JAVA class `Sieve` in which the algorithm is actually implemented.

Recall that a number is **prime** if given that $p|ab$ then either $p|a$ or $p|b$, or both. Primes have interested mathematicians for many years; they also form one basis of implementing the strong security that is now needed, for example on the Internet. It turns out to be surprisingly hard to determine precisely whether a number is prime; we return to this problem in Chapter **??**. For now we consider a simpler problem, where what is required is a list of all the primes up to a given n.

Mathematics To find this list of primes, start with all the numbers lined up in a row. I've omitted 0 and 1 because they are exceptions. So our list starts with 2. We know that 2 is prime, but given this prime, we also know that all the multiples of 2 are **composites**; in other words, are not prime. One way to think of this is by going through the list crossing out all multiples of 2; to actually do this we need to have a finite list of all the numbers up to a given maximum, say `Nmax`. I'm going to start by putting each number from my list into a box; as soon as I find the number is composite, the box is removed. So here is the situation before I start:

$$\boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{7}\ \boxed{8}\ \boxed{9}\ \boxed{10}\ \boxed{11}\ \boxed{12}\ \boxed{13}\ \boxed{14}\ \boxed{15}\ \boxed{16}\ \boxed{17}\ \boxed{18}\ \boxed{19}\ \boxed{20}$$
$$\boxed{21}\ \boxed{22}\ \boxed{23}\ \boxed{24}\ \boxed{25}\ \boxed{26}\ \boxed{27}\ \boxed{28}\ \boxed{29}\ \boxed{30}\ \boxed{31}\ \boxed{32}\ \boxed{33}\ \boxed{34}\ \boxed{35}\ \boxed{36}\ \boxed{37}\ \boxed{38}\ \boxed{39}\ \boxed{40}$$

Now let me mark, by "unboxing" them, every number which is a multiple of 2.

$$\begin{array}{cccccccccccccccccccc}
 & \boxed{2} & \boxed{3} & 4 & \boxed{5} & 6 & \boxed{7} & 8 & \boxed{9} & 10 & \boxed{11} & 12 & \boxed{13} & 14 & \boxed{15} & 16 & \boxed{17} & 18 & \boxed{19} & 20 \\
\boxed{21} & 22 & \boxed{23} & 24 & \boxed{25} & 26 & \boxed{27} & 28 & \boxed{29} & 30 & \boxed{31} & 32 & \boxed{33} & 34 & \boxed{35} & 36 & \boxed{37} & 38 & \boxed{39} & 40
\end{array}$$

We now know that the next boxed number in the list, in this case 3, is prime because we've "unboxed" the multiples of every number smaller than 3. So I can now mark all the multiples of 3. I could start with 6, but since that is already known to be composite, it is a waste of effort; instead I start with $3.3 = 9$. I'll mark the numbers we have *just* found to be composite by putting them in bold font.

$$\begin{array}{cccccccccccccccccccc}
 & \boxed{2} & \boxed{3} & 4 & \boxed{5} & 6 & \boxed{7} & 8 & \mathbf{9} & 10 & \boxed{11} & 12 & \boxed{13} & 14 & \mathbf{15} & 16 & \boxed{17} & 18 & \boxed{19} & 20 \\
\mathbf{21} & 22 & \boxed{23} & 24 & \boxed{25} & 26 & \mathbf{27} & 28 & \boxed{29} & 30 & \boxed{31} & 32 & \mathbf{33} & 34 & \boxed{35} & 36 & \boxed{37} & 38 & \mathbf{39} & 40
\end{array}$$

Lets continue in this way. Our next prime on the list is 5 and the first multiple of 5 we haven't marked is 5.5. So we start with 25 and mark multiples of 5.

$$\begin{array}{cccccccccccccccccccc}
 & \boxed{2} & \boxed{3} & 4 & \boxed{5} & 6 & \boxed{7} & 8 & 9 & 10 & \boxed{11} & 12 & \boxed{13} & 14 & 15 & 16 & \boxed{17} & 18 & \boxed{19} & 20 \\
21 & 22 & \boxed{23} & 24 & \mathbf{25} & 26 & 27 & 28 & \boxed{29} & 30 & \boxed{31} & 32 & 33 & 34 & \mathbf{35} & 36 & \boxed{37} & 38 & 39 & 40
\end{array}$$

A moments thought shows that a further step is not needed if we are only trying to find the primes up to 40. The next prime is 7 so the first unmarked multiple will be 49, too big to appear in out list. So the boxed numbers now are all the primes up to 40. This method is called a **prime sieve** because we start with all the numbers we are interested in, and "sieve out" the ones we know to be composite.

The Algorithm Having done a dry run and thought about the maths, its time to turn this into an algorithm — a systematic procedure or method which we can code. Here are the steps:

- Set up the storage which records whether or not a number is prime; we start at 2 rather than 1.
- Go to the smallest number (say n) in the list *which we haven't yet used* and mark all its multiples (*starting with* n^2) as composite. Then agree that this number has been used.
- When there is no more sieving to do, the remaining numbers are prime.

The second of these steps is the hard one; the parts in italics are added so the step works not just the first time, when $n = 2$, but also when $n = 3$, $n = 5$ and so on.

The Design Now comes the step of translating what we want to do into JAVA. It will come as no surprise that the storage is be an array; since we only wish to record whether or not a number is prime, an array of `boolean`s seems sensible. We clearly have to try to sieve for every $n \geq 2$, but give up immediately if n is not prime. The final step, once we have a usable n, is to actually do the sieve; in other words to mark multiples as composite. Here is the code, in the form of a JAVA class; you will see we have chosen to do all the work in the constructor.

```
public class Sieve {
    /** Here is where we store the status of a particular number. **/
    boolean[] prime;
    /** Only consider numbers which are no bigger than this. **/
    int sieveMax;
    /**
     * The constructor does much of the work, sieving every number up
     * to and including <code>sieveMax</code>.  After a sieve has been
     * constructed, we know the array <code>prime</code> has been set
     * correctly.
     **/
    public Sieve(int sieveMax) {
        this.sieveMax = sieveMax;
        // Although the array has been declared as a data member, it
        // must be instantiated before use.
        prime = new boolean[sieveMax+1];
        // Start by declaring that every number is prime
        for (int i = 0;i <= sieveMax;i++) {
            prime[i] = true;
        }
        // Then alter this for every number we know to be composite.
        prime[0] = false;
        prime[1] = false;
        int n = 2; // try to sieve with each number in turn
        // Only compute multiple from n*n upwards, so we only need try
        // numbers for which n*n is no more than sieveMax.
        int maxFactor = (int) Math.sqrt(sieveMax);
        while(n <= maxFactor) {
            if (prime[n]) {
                int k = n*n; // all multiples, k of n are compoite
                while (k <= sieveMax) {
                    prime[k] = false;
                    k+=n;
                }
            }
            n++;
        }
    }
}
```

4.3. Exercise. Write an `int nextPrime(int n)` method to add to the class, which returns the first prime in the list starting with n. Return 0 if there is no such prime.[3]

Solution: This is available in the html version of these notes.

[3] Of course there are an infinite number of primes; but in this class we only know the about primes smaller than `sieveMax`.

4.4. Exercise. A prime p is said to be a **twin prime** if both p and $p + 2$ are prime. Add an `int nextTwin(int n)` method which returns the first "twin" prime in the list starting with n. Return 0 if there is no such prime.[4]

Solution: This is available in the html version of these notes.

4.6 Multi-dimensional Arrays

We may also define multi-dimensional arrays. Here is an example in which we create an array with 3 rows, each of which has 4 entries. We choose to fill it with some rather simple initial values.

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

Clearly this is behaving very much like a 3×4 matrix. There is no requirement to fill the array by rows, as I have done, but you may find it helpful to think of array indexing as the same as that for matrices.

In fact there are no new principles involved in passing to two or more dimensions. We have simply created an array with three objects in it, called `arrayOne[0]`, `arrayOne[1]` and `arrayOne[2]`. Each of these objects is an integer array of size 4. In particular the third element of the first of these objects is called `arrayOne[0][2]`, as we would expect from the notation.

You can begin to see how this works if I repeat the above example but use the inbuilt `length` variable to tell me the sizes involved. Note that the number of entries in the first object (which we interpret as a row) is `arrayOne[0].length`.

```
int[][] arrayTwo = new int[3][4];
for (int i = 0; i < arrayTwo.length; i++) {
    for (int j = 0;j <arrayTwo[0].length;j++) {
        arrayTwo[i][j] = i+j;
    }
}
```

Thus the above code does exactly the same thing as in our first example.

To make the point even more accurately, I'm now going to generate a two dimensional array in which each row is of a different size.

```
int[][] arrayThree = new int[2][];
arrayThree[0] = new int[4];
arrayThree[1] = new int[7];
```

In the first line, we declare a two dimensional array of `int`s and instantiate the top level array. Then in the next two lines, we instantiate each of the lower level arrays, taking

[4]In contrast, it is not known at present whether there are an infinite number of twin primes.

advantage of the fact that, since the process happens separately, there is no need to make each row the same size.

These arguments show there is no reason to stick to just two indices; we can just as easily have arrays with 3 or even more indices. However the amount of storage necessary will grow quite rapidly with the number of indices.

4.5. Exercise. One way to describe Pascal's triangle is as a way of storing the Binomial coefficients. To remind you, the quantities $\binom{n}{k}$, or ${}_nC_k$, usually defined for $k \leq n$, are known as **binomial coefficients**. One definition of ${}_nC_k$ is as the number of ways of choosing k objects from a pool of n; an equivalent is as the coefficient of $x^k y^{n-k}$ in the expansion of $(x+y)^n$. It is usual to define ${}_0C_0$ as 1. The next few terms are as follows

n					${}_nC_k$				
0					1				
1				1		1			
2			1		2		1		
3		1		3		3		1	
4	1		4		6		4		1

In order to choose k things from n, we can first choose a distinguished element; then either choose k from the remaining $n-1$, or else choose $k-1$ from the remaining $n-1$ and include the distinguished element. Thus we have

$$ {}_nC_k = {}_{n-1}C_k + {}_{n-1}C_{k-1}. $$

Of course this doesn't work if $k = 0$ or $k = n$, but in each case the corresponding coefficient is 1. Otherwise, this formula enables us to work out the coefficients in one row from those in the previous row.

Show how to derive the triangle in JAVA, using the methods we've just discussed to create an array with just enough storage to hold the elements of the triangle, so row 1 has 1 entry, row 2 has 2 entries and so on.

Solution: This is available in the html version of these notes.

4.7 Arrays as Matrices

We conclude this Chapter with a more elaborate example in which we construct a `Matrix` class. One limitation of JAVA is the need, at this stage, to decide what sort of entries the matrix will have, even though almost all the code is totally indifferent to the type involved. We will fix on entries which are `double`s, probably the most useful choice. I'm going to keep the class simple, but it *will* prove useful to have the sizes available as instance variables as well as the actual entries. Here then is the start of the class.

```
public class Matrix {
    /** Store the number of rows in the matrix explicitly. **/
```

```
        int rows;
        /** And the number of columns in the matrix. **/
        int columns;
        /**
         * Store the actual entries; keep it private to stop anyone
         * working directly with this array.  Note that this is a
         * declaration; <code>entry</code> needs instantiation in a
         * constructor.
         **/
        private double[][] entry;
```

Before thinking of more general methods, we next need to write suitable constructors. The simplest constructor we need has to specify the size of the matrix. To save confusion about when it is to be done, I'm also going to insist that the array used internally to store the entries is instantiated at this stage. It is easy to forget to do it, and then try to store values in a non-existent array!

```
        /**
         * This constructor is responsible for setting sizes and
         * instantiating the data storage.
         **/
        Matrix(int rows, int columns) {
            this.rows = rows;
            this.columns = columns;
            entry = new double[rows][columns];
            // On instantiation, all the entries are set to zero.
        }
```

We also need a way of copying values into the storage. The method I now describe models one used by MAPLE in which the matrix entries are listed; in our case this means delivering them in a 1 dimensional array, with all the entries in the first row, followed by all in the second row and so on. Note the use of the *two* argument constructor in the call to `this(rows,columns)`. This is *much* better than writing out the code again and means that instantiation is always done in the same place.

```
        /**
         * A more capable constructor copies values into the
         * <code>entry</code> array.
         **/
        Matrix(int rows, int columns, double[] value) {
            this(rows, columns); // The 2-argument constructor
            for (int i = 0;i < rows; i++) {
                for (int j = 0; j < columns; j++) {
                    // The index of the <code>value</code> array shows why
                    // it is often easier to count from zero.
                    entry[i][j] = value[i*columns + j];
                }
            }
        }
```

Now we come to the class methods. Recall that the transpose b_{ij} of a matrix a_{ij} has $b_{ij} = a_{ji}$; in other words, the rows and columns are interchanged. It is a simple matter to code this. We first construct the new matrix, called `transpose`, that we are to return, and then copy data appropriately.

```
        Matrix transpose() {
            Matrix transpose = new Matrix(columns, rows);
            for (int i = 0; i <rows; i++) {
                for (int j = 0; j < columns; j++) {
                    transpose.entry[j][i] = entry[i][j];
                }
            }
            return transpose;
        }
```

A more interesting method implements the product rule. Note that we are *not* going to write a `static` method which multiplies two matrices together, rather we are writing a class method which (post)multiplies `this` matrix by an argument. Although I usually omit error checking code in examples, it seems vital here to check sizes before going ahead. But the rest of the code is simply implementing the formula

$$C = AB \qquad \text{if and only if} \qquad c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

where m for the number of columns of A, necessarily the same as the number of rows of B. I've even used the same indices in the same way in the code! The only other feature of the code is again the need to instantiate the new matrix, called `product` before calculating the values of its entries.

```
        Matrix multiply(Matrix m) {
            if (columns != m.rows) { // Size mismatch
                System.exit (-1);    // stop NOW!
            }
            Matrix product = new Matrix(rows,m.columns);
            for (int i = 0; i <product.rows; i++) {
                for (int j = 0; j < product.columns; j++) {
                    double value = 0.0;
                    for (int k = 0; k < columns; k++) {
                        value += entry[i][k]*m.entry[k][j];
                    }
                    product.entry[i][j] = value;
                }
            }
            return product;
        }
```

So far, we've seen two class methods, one of which had an argument. Thus we expected then to make use of the data available in the `entry` variable; we expect different objects to give different values. Some methods however are independent of any particular set of values. Here is an example; there is only one identity matrix of a given size. Hence the method is declared as `static`.

```
/**
 * It can be convenient to be able to build certain fixed
 * matrices, like the identity matrix (necessarily square)
 * easily.  These are static methods; we dont need to
 * instantiate a matrix before using it.
 **/
public static Matrix identity(int n) {
    Matrix identity = new Matrix(n,n);
    for (int i = 0; i < n; i++) {
        identity.entry[i][i] = 1;
    } // the other entries are automatically zero.
    return identity;
}
```

Finally here is some example code to exercise the class. It is not necessary to have a `main()` method in a class, but it can be convenient during development to test as you go. The matrix I construct is a simple one just there to show that everything works. Note too that we construct the identity matrix using a `static` method *before* any other objects of the `Matrix` class have been constructed.

```
public static void main(String argv[]) {
    System.out.println("The 2x2 identity matrix is:");
    Matrix.identity(2).display();
    double[] entries = {1.0/1,1.0/2,1.0/3,1.0/4,
                        1.0/2,1.0/3,1.0/4,1.0/5,
                        1.0/3,1.0/4,1.0/5,1.0/6};
    Matrix m = new Matrix(3,4,entries);
    System.out.println("The original matrix is:");
    m.display();
    System.out.println("The transpose is:");
    m.transpose().display();
    System.out.println("The product with the transpose is:");
    m.multiply(m.transpose()).display();
}
```

I've not shown you the `display()` method used above. You can write one yourself, or go to my original version.

Chapter 5

Java Background

We omit the part of the lecture which provided a brief view of computing over the years, and tried to explain where JAVA fits into the historical context. As a language, JAVA has the advantage that it is **platform independent** in that in principle the same code can run on many different machines. For example I have just written `Sieve.java` on the Sun in my office, which runs Solaris. I was able to run the same program immediately on my Windows 2000 laptop, with the same results. You can get more information from `http://java.sun.com`, the main Java site. In this Chapter we describe a little more of the background to the way JAVA works, and how you might go about using it.

5.1 Representing Data

At a very mechanistic level, essentially all (digital) computers manipulate strings of **bits** or **binary digits**. In other words they work with strings like

$$0110010010100101010111100100100001011010111010110010100001011 00$$

It is only at a higher level that such strings have a "meaning". The normal way this is done at present is to use a representation in which each character to be represented is assigned a different integer Thus "H" might be assigned to the decimal value 72, and then coded as the binary string "1001000".[1] This assignment was done in a standard way using the integers 0 to 127 and is known as ASCII — the American Standard Code for Information Interchange. As systems developed, it turned out there was not enough room in a 7 bit code, and the 8-bit (ie 0 - 255) "extended ASCII" was introduced with more symbols including for example the "235" character or £ sign. Even such a system has trouble with many of the worlds languages and JAVA uses a 16 bit representation of characters called **Unicode**.

Unicode gives a way of representing characters within a computer; so in principle the computer can then be programmed to process information.

5.1.1 Representing Integers

There are snags with such a system when working simply with numbers. We start by considering the simple case in which we want to manipulate integers. It would be complicated

[1] And indeed this is its ASCII value.

doing such a natural operation as addition when each decimal digit has an arbitrary code: extended ASCII uses "00110000" to represent "0", "00110001" to represent "1" and so on.

It is thus natural to have a different representation for integers when we wish to do arithmetic. JAVA has two main integer types, the `int` which uses 32 bits and the `long` which uses 64 bits. Rather than discussing them directly we illustrate the principles by considering integers with an 8 bit representation, giving the possibility of describing each of 0 .. 255. Arithmetic has problems however because of overflows; we can't compute 255 + 255 and expect to store than answer. Many languages, including JAVA cope with this by describing the answer as "Infinity"

Two's Complement Arithmetic

As soon as we wish to subtract, a second problem appears, since there is no room to represent the sign. One solution is to represent numbers in the range $-127 \ldots 127$ and reserve the top bit for the sign bit. A better solution uses what is called "two's complement" arithmetic. An abstract view of "-1" is as the number we add to 1 or "00000001" in order to get "0" or "00000000". Looked at in this way, there is an obvious candidate, namely "11111111", as you quickly verify. Here we have made good use of the finite length arithmetic to "lose" the awkward "carry" bit which tries to make the answer "257". Clearly then "-2" is simply "-1 + -1" and so on. In this way we have a representation which describes all numbers in the range $-128 \ldots 127$ and for which arithmetic works well. But the translation to decimal is less immediate; we have for example to interpret "11010000" as $-128 + 64 + 16$ or "-48". Clearly such a "two's-complement" representation extends to 32 bit or 64 bit integers and JAVA uses that representation.

It is clear that this representation has advantages over the naive "sign bit" one: not only do you represent more numbers, and avoid ambiguity (since "0" and "-0" provide two separate representations for the same number); but also arithmetic can be implemented very simply directly on the bit representation.

5.1.2 Representing Real Numbers

Not all numbers are integers; we certainly want to be able to talk about number such as π. JAVA has (at least) two number representations which allow this, the `float` and the `double`.

Before we discuss the difference it is again necessary to say how numbers are actually stored. It will help the discussion to stay with decimal notation for the present, although our real interest is in binary. A decimal number can always be written as

$$\pm 10^c \times 0.d_1 d_2 d_3 \ldots$$

where the **exponent** c is chosen so that $d_1 \neq 0$. We refer to the number $0.d_1 d_2 d_3 \ldots$ as the **mantissa**. Our choice of exponent means that $1 \leq d_1 \leq 9$ since we cannot have $d_1 = 0$. By truncating the decimal expansion of the mantissa we get an approximation to the required number; indeed if we know the mantissa has at most 23 digits, we can list all possible values and choose the nearest to the one we want. If further restrict the exponent to (say) 8 digits the range of numbers is limited and there are varying size holes within our set of numbers, but we have reduced the representation to a fixed length one. Using one additional digit for the sign, means this representation describes numbers using a total of 32 digits.

There are two additional wrinkles when we move to the *binary* representation used in practice. The first bit is still used for the sign of the mantissa ("0" if it is positive, "1" if negative). The "exponent" k comes next stored in 8 bits with $0 \leq k \leq 255$. However rather than using one bit for the sign of the exponent, the actual exponent is taken as

$$2^c = 2^{k-127}$$

where 127 is known as the **offset**. Finally comes the mantissa, stored in binary. However, because of the way the mantissa is normalised, the first bit is guaranteed to be "1", so need not be stored.

The numbers just described are of type `float`. There is also a type `double` which uses 1 bit for the sign, 11 bits for the offset exponent and the remaining 52 for the mantissa. *You will find it is almost always essential to use the full precision of a double in numerical work.* We now summarise what we've seen about the numbers that occur in a computer

> The "integers", both positive and negative, behave perfectly normally until they suddenly stop. The "real" numbers also stop, in the sense that there are numbers which are too big to represent. However there are always gaps between adjacent "real" numbers, corresponding to the smallest increment of the mantissa. These gaps vary in size, but can't be made arbitrary small; in particular there is a smallest positive "real".

It is clear that these "numbers" — I'll drop the quotations marks from now on — may need a little care when being used. Here are some examples to illustrate this

```
public class Tenths {
    /** A simple test routine. **/
    public static void main(String argv[]) {
        float oneTenth = 0.1F;
        float sum = 0.0F;
        for (int i = 1;i<101;i++) {
            sum += oneTenth;
        }
        System.out.println("One hundred tenths is " + sum + ".");
    }
}
```

Our first example appears to behave inaccurately because the machine cannot represent the fraction 1/10 accurately. It is thus working with an approximation; adding up these approximations increases the error linearly. We necessarily accept such inaccuracies when using `floats` or `doubles`.

Now let us look at an extreme case. I'm going to try to add up an infinite series. First lets get the maths right.

5.1. Proposition. *The sum* $\sum \frac{1}{n}$ *is divergent.*

Proof. We estimate the partial sums:

$$\begin{aligned}
S_n &= \frac{1}{1} + \frac{1}{2} + \left(\frac{1}{3} + \frac{1}{4}\right) + \left(\frac{1}{5} + \cdots + \frac{1}{8}\right) + \left(\frac{1}{9} + \cdots + \frac{1}{15}\right) + \cdots + \frac{1}{n} \\
&> 1 + \frac{1}{2} + \frac{2}{4} + \frac{4}{8} > 2\frac{1}{2} \quad \text{if } n \geq 15 \\
&> 1 + \frac{1}{2} + \frac{2}{4} + \frac{4}{8} + \frac{8}{16} > 3 \quad \text{if } n \geq 31 \\
&\to \infty \quad \text{as} \quad n \to \infty.
\end{aligned}$$

□

I'm going to keep on adding terms of this series until the sum *doesn't change.* If you think about the geometric series

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n} + \ldots$$

which converges to 2, we see that the partial sum carries on increasing even when a series converges. Surely then the next program will never stop?

```
public class SumTest {
    /** Compute 1 + 1/2 + 1/3 + .... **/
    public static void main(String argv[]) {
        float sum = 0;
        float newSum = (float)1;
        int i = 1;
        while (newSum > sum) {
            sum = newSum;
            newSum+=1/((float)++i);
            }
        System.out.println("The sum is " + newSum +
                                    " obtained with " + i + " terms.");
    }
}
```

The reason for this odd behaviour is that, although we can reasonably accurately represent the new term, $1/n$, eventually this gets smaller than the smallest increment allowed in the mantissa; after all the exponent remains essentially fixed, since the sum is about 1; in this case the smallest allowed increment is about 2^{-24} and we eventually get below this. You can probably see why the maxim

Always work with terms of roughly similar sizes.

is quoted. A more careful analysis could easily make the sum larger than any (representable) constant.

We conclude this discussion of oddities by seeing what happens when we insist on doubling an integer. Here is the simple code

```
public class Overflow {
    public static void main(String argv[]) {
        int n = 0;
        int value = 1;
        while (value !=0) {
            value*=2;
            n++;
            System.out.print("The value is " + value);
            System.out.println(" doubling " + n + " times.");
        }
    }
}
```

You can see that in this case we actually reach zero by double!

5.2 How to Program

Theses examples should have convinced you that the task of manipulating numbers on a computer is not straightforward. And I'm sure you've extrapolated to other apparently simple tasks. Yet in Chapter 2 you started to come to grips with simple programs, and in the second Practical you will have met examples which suggest some of the power available from learning to program. In this section we step back and discuss the *process* of programming. I think you may be surprised where the emphasis is placed in what follows

> It is much easier to offer advice about programming than it is to follow such advice.

Let me review the steps you will take when presented with a problem. Necessarily this is a list, but in practice different stages will take place in parallel.

Analysis: you need to understand the problem; typically you need to know how the user is to interact with the program and how the program is expected to change some data. And the unspoken problem is how you then can react to subsequent *change*.

Algorithm Development: involves deciding not only how this is to be done, but also how much should be explicit. If you can satisfy the requirements without committing to a particular algorithm, you can subsequently develop a better algorithm and substitute it without causing problems. In any case *you* need to know what algorithm you are going to implement.

Program Design: even when you know the algorithm, many things still need to be settled. A popular methodology that led to languages like JAVA arises from the assumption that the program is modelling the world in some way; then different parts of the program, in our case different *classes* describe interacting parts of the world to be modelled.

Coding: Now you can get down to details. A very good strategy is to get a "bare bones" implementation and then add the detail. You may have great faith in your design, but you probably want to build it in such a way that you see *something* working early on, and then refine it to achieve your goal. The design will ensure you are making progress towards the goal — that you don't need to backtrack, but limited testing during coding will help you spot design problems very early on.

Testing: is the hard bit. Your design and coding may be perfect, but can the program cope with the real world? Does it behave gracefully with bad data? Does it degrade well under stress? Does it cover *all* the cases it should. Have you *checked* the denominator is non-zero before dividing by it, and taken sensible action otherwise?

Tidying: now it is working the time has come to clean up. If you have declared a variable and then not used it, get rid of it. If you don't, it will cause endless puzzlement later on. Is there a more obvious way to do something? Does you code *look* clear? If not, you need to rewrite.

Documenting: and even if you write perfect code —we all do don't we — you need to say what is happening. Not in mechanistic detail, but you need a discipline that (usually) requires you to indicate what variables are to be used for, what functions do and what classes do. Your aim is to write code that is so clear you need no documentation, and then document!

Securing: but there is little use in writing code if it then vanishes. Make sure that in future you can always get at the code, and that it won't be changed (or worse still) deleted. You should think of taking personal responsibility for this; you may not have a tape drive at home, but everyone has access to a floppy disc. In the end, if it disappears, it may well be *you* that is most affected.

Probably the single most important thing to believe is that you will make mistakes. At the point when you firmly believe this to be the case, there is hope. You will then be prepared to go to a great deal of trouble to try to avoid making mistakes. Arguably all the language features of JAVA are designed to help you in this process, at the expense of greater initial complication.

5.2.1 Documenting

Most of these issues will keep on coming back during the course, but already it is time to discuss documentation in more detail. I hope you have already seen that we add comments to our programs. An algorithm may be clear to you when you write it, but to assume you will never come back to it can be a mistake. Here are some points to bear in mind.

- You may have a new problem which is a little like one you have already solved. You need to understand the solution in order to change it.
- You may find someone else wants to solve a problem you have already solved. You are happy to give them the code, but don't have the time to explain too much.
- You may discover the problem was more relevant, generic or profitable than you first thought.
- You may simply wish to show that it is not *your* code that is in error.

Here is a quick review of comment notation and conventions.

Single line comments: for a brief comment on the same line as code, or perhaps for a single line which is just a comment, use the `//` symbol, which indicates that the remainder of the line is a comment. You may also wish to use this to temporarily disable a line when debugging; if an error message goes away when you remove a line, you begin to have an idea of what is causing the error!

Block comments: for a longer comment, start with `/*`, type your comment, possibly taking up a number of lines and then finish with `*/`. Everything between those symbols is ignored *even another `/*` symbol*. This gives a problem if you try to "nest" comments, perhaps to comment out a section for debugging purposes which contains a comment for documentation; the comment ends at the first `*/` symbol.

Javadoc Comments: these are discussed below; they are just block comment which start with `/**` rather than `/*`.

Logic Comments: This is trick to remove arbitrary sections of code and so avoid the problem described above. Here is an example;

```
java code here;
if (false) {
    as much code as you wish to remove
    /* including comments */
    and more code
    /*
        with more comments;
        all this will be ignored.
    */
    more java code here
    } // but this parenthesis must be at the same level as the
      // one it matches
and the java code continues;
```

Note that anything enclosed by the `if (false)` is never executed, since the boolean `false` is never `true`.

You may find this variety is helpful when you are developing and testing; it is often better to comment out temporary tests than to delete them. That way, you can more easily retrace your steps if there is a problem.

Javadoc

Fortunately JAVA has facilities for generating much documentation automatically. It does so in `html`[2] format and you need to use a web browser, probably Netscape Navigator or Internet Explorer, to read it. To see it in action, run JAVADOC. Like the JAVA command itself, it can be a lot of typing, so it is easiest using ANT. Here is how you would do it in the local environment, starting from your home filespace:

[2]HyperText Markup Language, the natural language of the web.

```
bash% cd mx3015
bash% ant docs
```

Then point your browser at the *file* `h:/mx3015/docs/index.html`[3]. You are likely to have to go through the "choose file" dialogue box. I hope you are impressed. As you would expect, the command above was designed to run JAVADOC on all your java files and put the output in the file structure below the `docs` directory at top level. Much of this is done in the file

`mx3015/build.xml`

which is updated as new package are added to the structure. You should not need to move or edit this file which will be replaced every time you reload `mx3015.jar`.

There are two crucial conventions;

- JAVADOC comments begin with `/**` and end with `*/`, so they have an extra `*` at the start.
- JAVADOC comments immediately precede the thing that they document.

Plain text works perfectly well, but because they are interpreted as `html`, you may also use standard HTML codes such as `<p>` or `<code> name </code>` if you choose. The next step is to try it for yourself and go to the source at `http://jsp.java.sun.com/products/jdk/javadoc/` for more information.

Of course JAVA itself uses JAVADOC to describe its own workings. To see the documentation try `file:///f:/jdk1.2.2/docs/api/index.html`. You will probably find that this becomes a crucial reference; you may want to bookmark it. This is a local copy and for copyright reasons, we cannot put it directly on the web. However if you are away from the University system but have access to the web, you should go to the source at `http://jsp.java.sun.com/products/jdk/javadoc/`.

5.2.2 Securing

Remember we are going to ask for finished code. It is *your* responsibility to secure it. You will find that the files are very small; a number will go on a floppy.

Here is one way in which you can use fairly standard JAVA facilities to secure your work. Before I describe it let me start with a caution. Backing up your files should be done regularly, but otherwise should involve no danger. However there is no point in doing a backup unless you expect one day to do a "restore" in which files are retrieved from the backup. Indeed you should check that you *can* restore files long before you *need* to restore them.

> **Caution** Restoring files can destroy valuable data. You should always restore slowly and carefully.

Let me spell it out, to try to help you avoid problems. If you restore a file, you are intentionally overwriting the current copy with the copy on backup. You can see how things can go wrong. It is just as easy to overwrite a good file — one which has been changed recently and which contains valuable data – as one you are trying to restore.

[3]Note that browsers can load files as well as web pages.

Here is a way to try to avoid those risks. Deliberately create an archive or backup area, and include a "restore" area within it. Always archive in the same way, and always restore initially within the separate "restore" area. And finally, take a fresh backup before doing a restore! Here is how you might do the first one; it starts with the assumption that you have just logged in, so are at top level in your home directory.

```
bash% cd mx3015
bash% mkdir archive
bash% cd archive
bash% jar cvf mybackup.jar -C .. src
bash% mkdir restore
bash% cd restore
bash% jar xvf ../mybackup.jar
bash% # Now check that everything worked ... and then
bash% cd ..
bash% rm -rf restore
bash% mv mybackup.jar mybackup-todays-date.jar
```

Here you have used the `jar` command to actually make the archive. The options used are `c`, because you wish to *create* a new archive, `v` in order to ask for *verbose* output to see what is happening and `f` in order to say that the next thing on the line is to be the name of the archive. The rest of the line has `-C` to *change* to a directory from which to start the backup, followed by the name of that directory (`..`), and finally the file or directory which is to be backup up, with a name specified relative to the directory you have just given.

In the example we back up the `src` directory which is in the directory "`..`". It makes sense to give the backup file a `jar` extension to remind you what it contains, although this isn't a requirement.

The next step is to check the backup by doing a restore in a new directory. You will now recognise the command we have used for distributing the files as a "restore" process. As long as all is well[4] the backup file can then be saved in a "permanent" way; I find it convenient to add the current date to the filename. It is up to you how often you wish to back up. For information, you can get 19 copies of `mx3015.jar` or four copies of a full backup of *all* the java files for this course — many more than you will have — onto a single floppy. If you *do* want a copy on floppy simple drag and drop using `NT Explorer`.

You *could* backup up directly using `NT EXplorer`, dragging and dropping the whole directory structure. I've suggested using `jar` instead because

- the structure is automatically compressed, saving space;
- a `jar` file can be read on any system that can run JAVA; and
- it is arguably easier to do things repeatably with keyboard than mouse.

You might not agree with this last comment. Ultimately I don't mind *how* you do a backup and restore, just as long as you can do it reliably.

[4]But make sure you do this at least once; the first time I did it, `jar` didn't restore in the way I had expected. I got a surprise which in other circumstances could have been a problem. As a result of my test I corrected the instructions above!

If you subsequently want to restore a single file, simply do the above procedure, check that the file you want is as you want it, and then copy it to its “proper” location. But be aware of the need for caution.

Chapter 6

Built In Classes

In this chapter we discuss some of the classes provided as standard by JAVA. You can find out the complete choice at `http://java.sun.com/products/jdk/1.2/docs/api/index.html`. When you start writing your own software projects you are likely to need a number of fundamental building blocks to get started. Of course you could write these yourself. However there is a vast library of "standard" classes which you, as a JAVA programmer, may use, packaged with the Java Developers Kit. Before you start writing simple utility classes of your own, it is a good idea to check the supplied classes just in case there is something already available you could use or build on. I introduced the `Point` class to help explain the idea of a class; in fact there are several different `Point` classes available in the `java.awt.geom` package.

6.1 Wrapper Classes

A theme throughout the course has been that "everything is an object", yet we started with primitive types, `int`s, `double`s and so on which didn't behave as objects. A pure approach would be to force an `int` to be an object, but an object takes *much* more storage than 32 bits. So for efficiency this isn't done. However each of the primitive types has a corresponding class, in which it *can* be **wrapped**, effectively making it into a class. The `Integer` class is such a wrapper class. It is in the package `java.lang` so is always available, and holds an `int`. Much more interesting are its methods: The advantage that

Type	Method name	Argument type	Function
	`Integer()`	`int` value	constructor
Integer	`valueOf()`	`String` s	converts from a string like "17"
`int`	`intValue()`		the underlying value
`int`	`parseInt()`	`String` s	parses the string to get an `int`.

Table 6.1: The `Integer` class and some of its its methods

the `Integer` class has over an `int` is one of uniformity; we saw in Section 3.4 that every class is a subclass of the `Object` class; so we have the methods that are inherited from this class. An important one is that certain types of container classes available in JAVA, like a

Hashtable or a Vector, are just designed to hold objects. We can now use them to hold integers, while still allowing them to work just for objects.

Each of the primitive types has a wrapper class although the more important ones are the Integer, the Long and the Double. We will see all of these in action when we come to look at the KeyboardInput class in the package uk.ac.abdn.amths.mx3015.util in Chapter 7.

6.2 String

Virtually all programs use the String class; it is an important, and very heavily optimised class. We've been using strings from day one; the String literal "Hello World" is an object of the String class, part of the java.lang package; thus always available. The class is final in the sense that you can't create subclasses of it; this make for greater efficiency. Another optimisation is that you can't change any part of a String, although you can look at the characters, test them and even copy from a section of it; this can be summed up by saying that the String is *read-only*. There are a variety of methods; the only way to see if a particular operation is possible is to look up the JAVADOC class documentation. Here is some nonsense string manipulation to give you an example of some of the methods.

```
String line = "   a = b + c;   // This could be a line of java.";
line = line.trim();
int i = line.indexOf("//",0); // comment starts at position i
String codeOnly = line.substring(0,i);
System.out.println(" The Code is :" + codeOnly);
```

We start by creating a String simply by writing it out; formally we are using a very helpful constructor at this stage. On the next line, we get a brand new string, obtained by removing the spaces at the start of the first one using the trim() method.. Note that we *don't* edit the existing string; the old version is still there, although since we've assigned the new one to line there is no way to retrieve it. This is what I mean by saying that strings are read-only; they can't be edited. On the next line we look for a given substring — where the comment starts — and find its position. This index is used to create a new string with the substring() method, which uses our previously calculated index to pick out just the code part of the line. Finally in the argument of the System.out.print() method we build a string to print by joining or **concatenating** the quoted string with the string stored in codeOnly using the + operator.

```
int k = line.length();
String s = String.valueOf(k);
```

These two steps continue the example; we find the length of the string, and convert that integer, using a static method, into a String.

As you see nothing clever happens at each stage, but the effect mounts up. Note also that using + to join or concatenate two Strings works in general, not just when we are using a System.out.print() statement.

6.2.1 Comparing Strings

We saw in Section 5.1 that individual characters were represented using **Unicode**, so each character was represented internally using a 16 bit number. These codes then enable us to to impose an “alphabetical” order on characters chosen to be useful to humans. Extending this ordering to strings is then quite simple. Indeed comparing strings, perhaps to sort a list of strings into alphabetical order, is a very common and useful operation. However care is needed here. Two different comparison methods are used in the next fragment of code, which continues the above examples.

```
String code = "a = b + c;    ";
String samething = code;
boolean firstAttempt = (code == codeOnly);
boolean secondAttempt = (code.compareTo(codeOnly) == 0);
boolean thirdAttempt = (code == samething);
```

In the first attempt, we use the `==` method. This will be only true if the strings are actually the same; if both variables are pointing to the *same* string in memory. This is not the case here, and the value of `firstAttempt` is `false`. Instead we need the `compareTo()` method from the string class which returns an `int`. The sign is determined by looking at the first character where the two strings differ, and is based on the lexicographical ordering being used. In other words, `compareTo()` compares the strings, character by character; thus `secondAttempt` is `true` and shows how other string comparisons should be done. The only situation in which direct comparison will give the expected result is illustrated with `thirdAttempt`, which *is* true, although not very useful.

6.3 StringBuffer

I noted that there are no methods in the `String` class which allow a string to be modified. There is a related class, the `StringBuffer` class, optimised in a different way, which is designed precisely for this to happen easily.[1] Methods in the `StringBuffer` class include `insert()`, `append()` and even a `setCharAt()` method. A `StringBuffer` can either be constructed to be empty, or it can be initialised with the same sequence of characters as the String object given as an argument. We use this version in the example that follows:

```
StringBuffer sbuf = new
    StringBuffer( String.valueOf( 54321.6789 ) );
sbuf.reverse();
sbuf.insert( 4, " xxx " );
sbuf.append( " yyy " );
System.out.println( sbuf );
```

Here, a string was created representing the number 54321.6789. This was used to initialise the `StringBuffer sbuf`. The buffer was then reversed, had something inserted and something appended. You might like to check that you expect `9876 xxx .12345 yyy` to be printed out.

[1] A buffer is often used to mean a temporary storage area, which can, within limits, grow as required. You may have noticed that emacs refers to its working copy of text as being in a buffer; this is the same usage as JAVA for the same sort of reason.

Here is another quick example in which we manipulate individual **characters** a datatype we make little use of in this course. A Caeser cypher rewrites a message “shifting” each letter in the alphabet by a fixed amount; in this case 13 letters.

```
public class Caeser {
    public static void main(String[] args) {
        StringBuffer message = new StringBuffer("attack at dawn");
        char current;
        for( int i = 0; i<message.length(); i++ ) {
            current = (char)(message.charAt(i) + 13);
            message.setCharAt(i,current > ’z’?
                              (char)(current-26):current);
        }
        System.out.println( "Transmit the message " + message );
    }
}
```

The output is

```
Transmit the message nggnpx-ng-qnja
```

The example demonstrates that, in contrast with the behaviour for `Strings`, the `message` object itself was changed.

6.4 Vector

As you progress in JAVA, you will begin to feel the need for so-called **container** classes which hold collections of things and allow you to get at them in a useful way. We’ve seen one such already; an array of `Objects` could be used to store anything we want. Indeed using the wrapper classes discussed in Section 6.1 we can put *anything* into an array.

We now discuss the `Vector` class. This is a container class which is similar to an array but proves significantly easier to use in practice. This use of the word ’vector” is fairly specific to JAVA, and is not very closely related to the way mathematicians use the word.

In brief a `Vector` in the package `java.util` is what might be called an *extensible array*. It is not necessary to get the capacity right initially; items can be added or removed and the `Vector` will grow or shrink to accommodate them. Like an array, it contains components that can be accessed using an integer index. It can only store `Objects`; you will see in the example below that this really isn’t a restriction.

We give a minimal implementation of a class designed to hold a polygonal path. It uses the `Point` class we’ve already seen, and stores the polygonal path as a list of `Points` in an instance variable of class `Vector`.

```
import java.util.Vector;
import java.util.Iterator;
public class PolyLine {
    Vector points;
    PolyLine() {
        points = new Vector();
```

```
            }
            void addPoint(Point p) {
                points.addElement(p);
            }
```

This is one way to add elements to a Vector. You can find a number of other methods in the documentation. As an example of how to retrieve things, we give a toString() method below. It runs through the points in points adding them to the output String. In fact we do so twice. The first time illustrates a very general way of “traversing” or iterating” through the elements of any collection class. An Iterator guarantees to do so, although not necessarily in a particular order. This in fact makes it an unsuitable choice in this application, so we give a more routine way of retrieving the elements in points. When retrieved, using the elementAt() method, the resulting Object then has to be cast to the right type; in our case a Point. If this cast is invalid, a RuntimeException will be thrown.

```
            public String toString() {
                StringBuffer sb = new StringBuffer();
                // This may not get the right order
                Iterator p = points.iterator();
                while (p.hasNext()){
                    sb.append((Point) p.next());
                }
                //but the obvious way will
                sb = new StringBuffer();
                for (int i = 0; i < points.size(); i++) {
                    sb.append((Point) points.elementAt(i));
                }
                return sb.toString();
            }
```

Finally comes a main() to exercise the class; a rather uninteresting collection of points – in fact collinear — is created and the resulting PolyLine displayed.

```
            public static void main(String[] args) {
                PolyLine pl = new PolyLine();
                for (int i = 0; i < 100; i++) {
                    pl.addPoint(new Point(i,-i));
                }
                System.out.println(pl);
            }
        }
```

Chapter 7

Exceptions and File Handling

7.1 Exceptions

A computer program can't necessarily guarantee to get everything right if it interacts with its environment. Here are some examples of things that may go wrong

- a 'divide by zero" error, in which you compute `a/b` and `b` turns out to be zero;
- you try to construct a triangle with three collinear point;
- you try to access a file which isn't there; or
- you try to read an array element which isn't there.

In all of these cases, the result may not be what you intended. Of course if you are programming defensively you should check each such operation. But what are you supposed to do if it doesn't work?

In the "triangle" example, you may decide to check for collinearity in the constructor. You may know you *can't* accept such triangles. But what do you do if you get presented with such three collinear points as arguments to the constructor? Of course you can (and should) try hard not to get into the situation. But it would be good to catch any errors that have been overlooked.

In old-style programming, "difficult" methods (or subroutines) had a way of commenting on how well they had done; a variable called perhaps `status` which could describe both success and various types of error. This worked well as long as programs weren't very heavily nested. But what if you are 20 methods deep — method `method1` has called `method2` has called ... has called `method20`, which has found an error. Suppose you can only deal with it at top level - in `method1`. You need a way of throwing the error all the way back to `method1`. In other words, `method19` has to look at the status return from `method20` and set its own status based on that; its caller, `method18` then has to process this status return and so on.

Rather than doing this every time, a mechanism developed called an **exception** which JAVA implements. The syntax is as follows:

```
// At the start of something that may go wrong
try {
        <commands that may go wrong>
    } catch (Exception e) {
        <things to do with the Exception, called e>
    } finally {
        <things you want done anyway>
    }
```

The whole "try" block is executed, but if an "exception" is encountered, the rest of the block is skipped and control passes to the "catch" block, which is supposed to cope with it. As you may expect, an Exception is a special type of JAVA object. In case there is something you absolutely *must* do whatever happens, control goes through the `finally` block— however the previous code was traversed. It follows from the general rules that the argument of the catch clause can also be any class that extends an `Exception`.

Here is an example:

```
package uk.ac.abdn.maths.mx3015.files;
import java.io.InputStreamReader;
import java.io.BufferedReader;
public class ReadInt{
    /** This isn't a real class, just an example; so we might as well
        let all the action occur in the main method. */
    public static void main(String argv[]) {
        BufferedReader b = new BufferedReader(new
            InputStreamReader(System.in));
        try{
            System.out.print("Enter an integer to be read: ");
            String s = b.readLine();
            int i = Integer.parseInt(s);
            System.out.println("The integer " + i + " was input.");
        }
        catch (Exception e) {
            System.err.println(e); // print the exception.
        }
    }
}
```

Here is a brief discussion of this example:

- A `BufferedReader` is the thing you read from — you can see it is built in a fairly complicated way. Note the explicit import of the two classes we use. You could import `java.io.*` giving access to the all the JAVA input - output facilities, but it is considered better style to be explicit about the classes you use.

- Note what is happening; you read a line to get a string, and then extract an integer.

Now lets try to do the same task reading from a file. The syntax is almost the same

```
package uk.ac.abdn.maths.mx3015.files;
import java.io.FileReader;
import java.io.BufferedReader;
public class FromFile{
    public static void main(String argv[]) {
        try{
            BufferedReader in =
                new BufferedReader
                    (new FileReader ("numbers.dat"));
            String s = in.readLine();
            int i = Integer.parseInt(s);
            System.out.println("The integer " + i + " was read.");
            in.close();
        }
        catch (Exception e) {
            System.err.println(e); // Print the exception to warn.
        }
    }
}
```

There are some interesting differences.

- a `FileReader` extends a `InputStreamReader` which is or abstract model of a stream of symbols - most easily though of as a keyboard;
- whereas before we may hit a NumberFormatException, now we can also run foul of a FileNotFoundException; thus it might be quite sensible to catch the latter and pop up a file chooser; and
- in practice in simple programs we usually don't actually do much when we catch the exception.

The aim of the `try` block is both to protect the rest of the program and also to draw the attention of you — the programmer — to the "difficult" bits. One consequence is that any variable which is only set within a `try` block is regarded as unsafe, and so will be treated as "uninitialised" outside the block. Sometimes this causes "try creep" as the "try" block is expanded more and more; often the variable is initialised to a default value *before* the `try` block.

You are at liberty to create your own subclasses of exceptions. Lets start with a simple example. In the Circle class we first met in Section 3.2 let me suppose we want a constructor in which the radius is the sum of the co-ordinates of the centre.[1] If misused, this could allow us to construct a circle with a negative radius. We could protect the constructor with a purpose-built exception as follows.

```
Circle(Point p) throws NegativeRadiusException {
    this(p,0);
    double r = p.x + p.y;
```

[1] I can't think of a good reason why, except to provide this example!

```
            if ( r < 0) throw new NegativeRadiusException(r);
            radius = r;
        }
```

Now we need to build our own NegativeRadiusException. This is a good example when it is useful to have more than one class in a file. To keep the code short, I won't actually do anything to correct the situation.

```
        class NegativeRadiusException extends Exception {
            // Just need a constructor.
            NegativeRadiusException(double r) {
                System.out.println("Potential radius " + r
                                   + " can't be negative.");
            }
        }
```

And now we can use this constructor. As I've noted, it can only be used in a try block. In this case we catch the exception immediately.

```
        try {
            Circle circle = new Circle(p);
        } catch (NegativeRadiusException e) {
            // In this example do nothing.
        }
```

7.1. Exercise. Write a java class to represent a cubical tank of a given length, breadth and height, to which liquid may be added. You should be able to construct a tank with given dimensions, although you may assume the dimensions are sensible (positive). Write a toString() method, and methods to get the current volume of liquid in the tank, to get the *depth* of the liquid in the tank, and to add a given volume of liquid to the tank. This last should throw a TankOverflowException if it would cause the tank to overflow and the liquid should not be added. You should write such an Exception as a nested class. Then exercise these methods in a main method. You need do nothing when you catch the Exception.

Solution: This is available in the html version of these notes.

Here is quite a long example which tries to illustrate how exceptions might be used. I've kept it simple, and avoided Exceptions arising from input by using the KeyboardInput class in the uk.ac.abdn.maths.mx3015.util package. You may wish now to look at it and see how it was implemented to do this; in fact it simply sets default values and then catches any exceptions.

The general principle with exceptions is that, within a method, you can throw and catch as you like, and even re-throw Exceptions that have been caught. However anything not caught by the end of the method must be declared at the start.

This is the "right" way to do things; such an Exception is called a **checked exception** precisely because they should either be declared or checked for (ie caught). But the designers of JAVA were people who live in the real world and allowed a less exacting form of Exception, called an **unchecked exception**. You have already met such an unchecked, or

RunTimeExceptions, such as illegal array access. These simply raise an exception when they occur — but it would be intolerable if every array access had to be protected by a `try - catch` block.

It is hard to give a brief example to show how things should work. Here is an attempt, supposed to simulate processing with many nested classes; hence the name `Cascade`.

```
package uk.ac.abdn.maths.mx3015.files;
import uk.ac.abdn.maths.mx3015.util.KeyboardInput;
public class Cascade{
    Cascade() {
        KeyboardInput in = new KeyboardInput();
        boolean wanted = true;
        FirstCascade fc = null;
        while (wanted) {
            System.out.print("Give an integer less than 5-> ");
            int i = in.readInteger();
            try {
                fc = new FirstCascade(i);
                wanted = false;
            } catch (Wobbly w) {
                wanted = true;
            }
        }
        System.out.print("Thank you");
        if (fc.setupGood) {
            System.out.print(" very much");
        }
        System.out.println(".");
    }
    public static void main(String argv[]) {
        Cascade c = new Cascade();
    }
    class FirstCascade {
        boolean setupGood = false;
        FirstCascade(int i) throws Wobbly {
            System.out.println("Initialising FirstCascade");
            try {
                SecondCascade sc = new SecondCascade(i);
                setupGood=true;
            } catch (Grumble g) {
                setupGood=false;
            } finally {
                System.out.println("FirstCascade done");
            }
        }
    }
```

```
            class SecondCascade {
                SecondCascade(int i) throws Wobbly, Grumble{
                    if (i > 9) {
                        throw new Wobbly ("Please do as I ask!");
                    }
                    if (i > 4 ) {
                        throw new Grumble("I'll accept that.");
                    }
                }

            }
            class Wobbly extends Exception {
                public Wobbly(String s) {
                    System.out.println("Wobbly: " + s);
                }
            }
            class Grumble extends Exception {
                public Grumble(String s) {
                    System.out.println("Grumble: " + s);
                }
            }
        }
```

Notice that, as with the `NegativeRadiusException` and `TankOverrflowException`, we can choose class names to try to help document; in the above case we picked `Wobbly` and `Grumble`.

7.2 File Output

It is presumably clear that there are many different methods of reading data from a file; the choice very much depends on the task in hand. You can find out more by going to the Java documentation. In the same way there are many ways of writing *to* a file. Here is the most “natural” one; again it has to be done within a “try” block for the similar reasons to those that applied to input; we can’t be sure that the file will be there, that it will be writable, that there will be enough disc space to complete the write etc. And again it is a combination of methods, designed to pick up various useful characteristics.

```
        package uk.ac.abdn.maths.mx3015.files;
        import java.io.FileOutputStream;
        import java.io.BufferedOutputStream;
        import java.io.PrintStream;
        public class ToFile{
            public static void main(String argv[]) {
                try{
                    PrintStream  out =
                        new PrintStream (
                    new BufferedOutputStream (
                        new FileOutputStream ("output.dat")));
            out.println(17);
            out.println(7.132);
            out.println("This is a string");
            for (int i = 0; i < 10 ; i++ ){
                out.print(i + " ");
            }
            out.print("\n");
            //out.flush();
            out.close();
        }
        catch (Exception e) {
            System.err.println(e); // Print the exception to warn.
        }
    }
}
```

This is mostly straightforward; some comments:

- A `BufferedOutputStream` doesn't write immediately. Instead it holds on to the data until there is enough to make it worth while "talking" to the file. You can *force* a write with the `flush` method. In fact it is not necessary above because the `flush` method is called automatically before the file is closed.
- Everything behaves as you are used to because the `print` and `println` methods are now familiar. You can add "escape" characters to strings to insert non - printing characters. Thus `\t` adds a "tab" character, while `\n` add a newline. Thus `out.print("\n")` used above is simply another — and less straightforward — way of writing `out.println()` to get a new line.
- As you expect, you can give the file name in a string variable, rather than explicitly. Indeed it would normally be gathered using some form of dialogue box.

7.3 Reading Data

The whole point of computing is to process information; in other words, to read information from the environment, process it in some way, and as a result, affect the environment. So far this has typically involved reading data from the keyboard, or even from the program itself, and writing data to the screen. We have just seen how to replace this data flow with a much more general model, in which both input and output are replaced by *files*. In Section 7.2 we found it was possible to write the output file rather freely — essentially we could write whatever we wanted there. Out final step on the way to flexible input and output is to look again at input, where, at present, we can only read one item on each line.

Of course, if we read the line as a `String`, we can clearly read all the data available. If that `String` is really a `double`, then we already saw in Section 7.1 a way of parsing the string to obtain the double. The only problem we have working more generally is thus that of splitting a string up into lots of substrings, each of which can be converted to an `int` or a `double` etc.

Java has a special class designed to do exactly this job, namely the `StringTokenizer` class. Its job is to chop up strings and return parts of them on demand. By default, it splits strings at spaces, but additional constructors allow you to specify what the "token separator" should be, and even whether or not to include the separator with the token. Here is an example; it reads data from a single line, but there is no difficulty extending it to read data, as required, from a whole file. As you will find in the documentation, a call to `in.ready()` checks that a line is available to read. Note the `import` statement; here I've been very explicit about what to use in the `java.util` class.

```
package uk.ac.abdn.maths.mx3015.files;
import java.io.FileReader;
import java.io.BufferedReader;
import java.util.StringTokenizer;
public class ReadLines{
    public static void main(String argv[]) {
        try{
            BufferedReader in =
                new BufferedReader
                    (new FileReader ("numbers3.dat"));
            // Don't bother checking in.ready() here?
            String line = in.readLine();
            StringTokenizer st = new StringTokenizer(line);
            System.out.print("Read ");
            for (int i = 0;i < 3;i++) {
                String s = st.nextToken();
                double d = Double.parseDouble(s);
                System.out.print(d + " ");
            }
            System.out.println();
            in.close();
        }
        catch (Exception e) {
            System.err.println(e); // Print the exception to warn.
        }
    }
}
```

Chapter 8

Solving Equations

In this chapter we investigate methods for finding roots or equivalently of solving an equation. Recall that a point a is a **root** of a function f if $f(a) = 0$. Here are some examples:

- solve the equation $x^2 = 2$;
- for a given b, solve the equation $bx \cos x = \sin x$;
- solve the equation $x = \exp(-3x)$; and
- solve the equation $x^2 - 20000x + 1 = 0$.

8.1 Quadratics

Surely there is nothing hard about solving a quadratic equation? Lets start with the formula: the roots of $ax^2 + bx + c = 0$ are obtained by completing the square: we get roots

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Provided a, b and c are real, the action is in the **discriminant** Δ given by $\Delta = b^2 - 4ac$; we have real roots if $\Delta \geq 0$ and complex ones otherwise. However even when the roots are real, the story isn't over. Consider the following

8.1. Exercise. Solve the equation $x^2 - 20000x + 1 = 0$.

The quadratic formula gives us roots

$$\frac{20000 \pm \sqrt{4 \times 10^{10} - 4}}{2} = 10^5 \pm 10^5 \sqrt{1 - 10^{-10}}.$$

Using the binomial theorem for $(1 - x)^{1/2}$ on the square root gives approximate roots

$$10^5 \pm 10^5(1 - 10^{-10}/2) = 10^5 \pm (10^5 - \tfrac{1}{2}10^{-5})$$

where the next term in the expansion after the one involving 10^{-10} is of the order of 10^{-20}. One root is clear; it is $2.10^5 - \frac{1}{2}10^{-5}$. We know this value with very great accuracy since the next term is about 10^{-15}. The ratio of the correction term to the root itself is 10^{-20}; so small that even if the root is a `double`, the correction term has no effect.

Unfortunately, computing the second root using the formula gives the value $\frac{1}{2}10^{-5}$. We ended up cancelling the "dominant" terms. In this case, the second order of 10^{-15} is only down a factor of 10^{-10} on the value we were going to report. In contrast to the other root this one is (relatively) much less accurate.

There is another way in which we can avoid subtracting two terms of almost the same size. Recall that $(x-\alpha)(x-\beta) = x^2 - (\alpha+\beta)x + \alpha\beta$; in other words, we can read off the *symmetric functions* of the roots, in this case the sum and the product, directly from the equation. Looking at our equation we see that the sum of the roots is 2.10^5 while the product is 1. If the smaller root is β, we see using the "product of the roots" formula that

$$\beta = \frac{1}{2.10^5 - \frac{1}{2}10^{-5}} = \frac{1}{2}10^{-5}\left(1 - \frac{1}{4}10^{-10}\right)^{-1} = \frac{1}{2}10^{-5}\left(1 + \frac{1}{4}10^{-10}\right). \qquad (8.1)$$

In this way we get the same relative accuracy for the smaller root as for the larger root.

This is a general maxim in numerical analysis.

> Do all you can to avoid the loss of significant figures which occurs when subtracting two terms of almost the same size.

The point is that there are often different ways of achieving the same final result. As above, one way may preserve many more useful figures than another.

8.1.1 A Quadratic Class

It would be natural to incorporate these ideas in a class for quadratics equations which "knows" its roots. Here is one way to implement it.

```java
package uk.ac.abdn.maths.mx3015.formulae;

import java.io.*;
import java.text.NumberFormat;
import java.util.StringTokenizer;

/**
 * This class instantiates a quadratic written as ax^2 + bx + c.
 *
 * @version $Revision: 3.1 $
 * @author  Ian Craw
 */
public class Quadratic {

    /** The coefficient of x^2. */
    private double a;

    /** The coefficient of x. */
    private double b;

    /** The constant term. */
    private double c;

    /** Declare whether we have real or complex roots */
    public boolean hasRealRoots;

    /**
     * The root with the larger modulus - only useful if
     * <code>hasRealroots</code> is true.
     **/
    public double root1;

    /**
     * The root with the smaller modulus - only useful if
     * <code>hasRealroots</code> is true.
     **/
    public double root2;

    /**
     * The real part of a pair of complex roots - only useful if
     * <code>hasRealroots</code> is false, so the Quadratic has
     * complex roots
    **/
    public double realPart;

    /**
     * The imaginary part of a pair of complex roots - only useful if
     * <code>hasRealroots</code> is false, so the Quadratic has
     * complex roots
    **/
    public double imaginaryPart;

    /**
     * The constructor is responsible for checking we <em>have</em> a
     * quadratic.  It initialises the constructor and then calls
     * <code>solve</code> which solves the quadratic using the formula
     * and sets the roots appropriately.
     **/
    public Quadratic (double a, double b, double c) {
        if (a == 0) {
        }
        this.a = a;
        this.b = b;
        this.c = c;
        solve();
    }
```

```
/**
 * Obtain either two real rools or the real and imaginary parts of
 * the two complex roots, necesarily conjugate since the
 * coefficients are real.  Sets  the boolean
 * <code>hasRealRoots</code> and </em>exactly</em> two of the four
 * instance variables which store the roots.
 **/
public void solve() {
    double discriminant = b*b - 4*a*c;
    // Code needs writing
}
/** A convenience method to display Quadratics. */
public String toString() {
    return (a + " x^2 + " + b + " x + " + c);
}
/**
 * Purely for testing - print the roots.  Note that "System.out"
 * is a perfectly sensible PrintStream.
 **/
public void printRoots(PrintStream out) {
    // The next three lines use more advanced language features.
    // to control how much space each entry takes up.
    NumberFormat nf = NumberFormat.getInstance();
    nf.setMaximumFractionDigits(2);
    nf.setMinimumFractionDigits(2);
    StringBuffer sb = new StringBuffer();
    sb.append("The quadratic " + this) ;
    if (hasRealRoots) {
        sb.append(" has real roots " + nf.format(root1)
                    + " and " + nf.format(root2) + ".");
    } else {
        sb.append(" has complex roots " + nf.format(realPart)
                    + " +/- i " + nf.format(imaginaryPart) + ".");
    }
    out.println(sb);
}

/** Create some quadratics to test the class. */
public static void main(String argv[]) {
    Quadratic p = new Quadratic(1.0,5.0,6.0);
    p.printRoots(System.out);
    p = new Quadratic(2.0,-4.0,4.0);
    p.printRoots(System.out);
    p = new Quadratic(1,-2.0,1.0);
```

```
        p.printRoots(System.out);
        p = new Quadratic(1.0,10000.0,1.0);
        p.printRoots(System.out);
        p = new Quadratic(1.0,-10000.0,1.0);
        p.printRoots(System.out);
    }
    /** Purpose built exception class just to report the error. **/
    class IllFormedQuadraticException extends RuntimeException {
        // Just need a constructor.
        IllFormedQuadraticException() {
            System.out.println("The quadratic term must be non-zero.");        }
    }
}
```

One point to note is the use both of `toString()` method, and the more verbose and customised `printRoots()`. In general we would expect to work directly on the data output format , as we've done here, rather than relying completely on the `toString()` method.

8.1.2 Complex Coefficients

Of course it may be convenient to do this in general, even when the coefficients are *complex*. Which means the above method, in which we assume that the discriminant is real, breaks down. We can still push ahead if we have a method `Complex.sqrt()` which works in general; in other words if we have a working `Complex` class. You are invited to try this version for yourself.

8.1.3 Absolute and Relative Errors

Suppose the exact value of some quantity is T and that A is an approximation to this value. There are two standard ways in which to measure the 'error' in using A as an approximation to T:

- Absolute Error $\quad E = |A - T|$;
- Relative Error $\quad E = \left|\dfrac{A - T}{T}\right|$.

Percentage error is relative error times 100.

For numbers close to 1 both measures are roughly the same. For T very large or very close to zero they are very different. In a computing situation you will almost always be concerned with relative error, since (real) numbers are represented in such a way that the relative error between the "actual" number and the closets "representable" number is roughly constant; for a `double` this is about 10^{-15} as your work computing $\sin(x)$ will have shown.

Of course in a given real situation you need a feel for which type of error is the important one. There are many situations in which your main concern is the percentage error. If you order 1000 kilos of sand and only receive 999.9 kilos then you might not complain too loudly. On the other hand, if the two bits of the channel tunnel had been out in direction by the same relative error, in a 40km tunnel the result would have been a 4m inaccuracy, probably extremely expensive.

8.2 Iterative Methods

We have seen that, even when there is a formula to solve an equation, we necessarily end up with an *approximation* to the required solution. Since this is inherent in working numerically, there is no reason to prefer "exact" methods to those which involve successive approximations to a root. All we require is that the root can be evaluated as accurately as is needed.

8.2.1 The Bisection Method

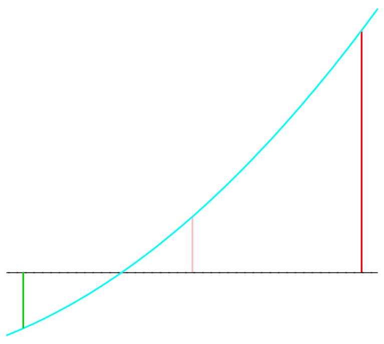
Figure 8.1: The mid point becomes the new end-point.

We start with a very simple method, based on the Intermediate Value Theorem: if we have a *continuous* function which is positive at one point and negative at another, somewhere between them it must vanish or have a root[1].

To use this result, assume we are able to **bracket** the root; in other words, we can find a pair of points a and b with $f(a) < 0$ and $f(b) > 0$ or vice-versa. We can then investigate the sign of $f((a+b)/2)$ and use this mid-point to replace either a or b; the choice being made so the new pair of points still brackets the root. This is shown in Fig 8.1.

We can express this algorithm in terms of iteration as follows: assume that a_n and b_n bracket the root as above and that $a_n < b_n$.

- let $p_n = (a_n + b_n)/2$;
- if p_n is a good enough approximation to the root, stop;
- if continuing, and $f(a_n)$ and $f(p_n)$ have the same sign, let $a_{n+1} = p_n$ and $b_{n+1} = b_n$; otherwise let $b_{n+1} = p_n$ and $a_{n+1} = a_n$.

Note than a_n forms an increasing sequence which is bounded above; that b_n forms a decreasing sequence which is bounded below, and that

$$|b_n - a_n| = \frac{1}{2^n}|b_0 - a_0|.$$

so our sequence of approximate roots p_n converges; indeed we know the maximum error in terms of the starting positions $|b_0 - a_0|$ and the number n of successive approximations. If we seek an error of at most ϵ, we must have

$$\frac{1}{2^n}|b_0 - a_0| < \epsilon \quad \text{so} \quad n > \ln_2\left(\frac{|b_0 - a_0|}{\epsilon}\right).$$

and we can calculate in advance the minimum number of steps needed for any required accuracy.

[1]See for example MA2001 notes, Theorem 4.30

8.2.2 False Position

The bisection method is both simple and guaranteed. But it is slow; it makes no assumptions at all about the behaviour of the function, and so its guaranteed rate of convergence is also the best possible. In contrast the **reguli falsi** or **method of false position** gives fewer guarantees, but can give an acceptable approximation *very* much more quickly.

The improvement is illustrated in Fig 8.2 using the same function as before. Geometrically the change is simple. We know two points $(a, f(a))$ and $(b, f(b))$ on the graph; while there is no guarantee, let us assume the function is a straight line between these two points, and find where the line crosses the axis. Then take this as the next approximation.

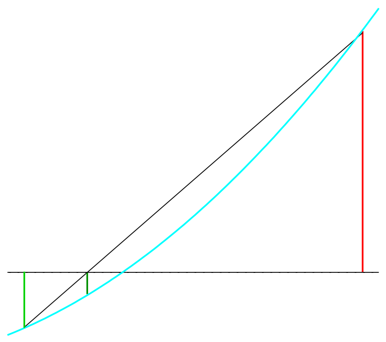
Figure 8.2: Reguli falsi: assume the function is linear to get the next approximation.

The example is instructive. It is clear that we have a good approximation, and we can continue to bracket the root as we move. However it is equally clear that we can't continue treating the distance between the two brackets as a measure of the error; in the example shown, this difference stays large.

At a root α of f, we have $f(\alpha) = 0$, and so, assuming f is continuous, we know that f is small *near* α. We now assume the converse; that if $f(\beta)$ is small, then β is a good approximation to the root α. Thus we assume that an approximate root is an approximation to the actual root.

This is an *assumption*, and things may go wrong. If f is very flat in this area, F may not actually cross the axis for some time. And if f has a local minimum near β, but doesn't change sign near β, we have been completely misled; there is no root near β. We could be formal and put a condition on f that f' doesn't get too small near β, but this can sometimes be hard to verify. Certainly it is no substitute for having a reasonable idea of the behaviour of the function and so knowing that there are no catches.

However using the assumption that an approximate root is an approximation to the actual root, our previous iteration scheme can be used with minor changes. Note that the equation of the line joining the two points we consider is

$$\frac{y - f(a)}{f(b) - f(a)} = \frac{x - a}{b - a}.$$

and putting $y = 0$ gives the updating rule below.

- let $p_n = a_n - f(a_n)\dfrac{b_n - a_n}{f(b_n) - f(a_n)}$;
- if p_n is a good enough approximation to the root, stop;
- if continuing, and $f(a_n)$ and $f(p_n)$ have the same sign, let $a_{n+1} = p_n$ and $b_{n+1} = b_n$; otherwise let $b_{n+1} = p_n$ and $a_{n+1} = a_n$.

However we can no longer guarantee the speed of convergence even though we expect it to be better.

8.2.3 Secant Method

What happens if we abandon the requirement to bracket the root? Geometrically we show the situation in Fig 8.3. The points A and B are the original ones bracketing the root, and the point C is derived form the chord AB as before. We have shown, in D the next iteration using the *reguli falsi* noting that the other end of the "bracket", namely A isn't contributing much to the convergence. However the most recent two points, namely B and C *are* fairly close to the required root, and if this pair is used instead of the bracketing pair AC in the next step, we move to E which looks as though it is a very significant improvement. This is the **secant** method; we still use a chord on the curve to estimate the next approximation to the root, but now the chord is drawn between the *last* two points on the curve, rather than the most recent bracketing pair.

This leads to the following algorithm

- let $p_n = a_n - f(a)\dfrac{a_{n+1} - a_n}{f(a_{n+1}) - f(a_n)}$;
- if p_n is a good enough approximation to the root, stop;
- if continuing, let a_n be a_{n+1} and let $a_{n+1} = p_n$.

Note that, just as before, we need *two* points to start the process. There is no requirement that successive approximations bracket the root, but obtaining such a pair initially is at least evidence that we are near a root!

Although we have derived this method in a fairly *ad hoc* way, it turns out that the secant method is one of the best methods to use for "difficult' roots providing we can abandon the guarantee of the bisection method. We discuss this further after describing one more method, which is essentially the secant method taken to the limit in the sense of the calculus.

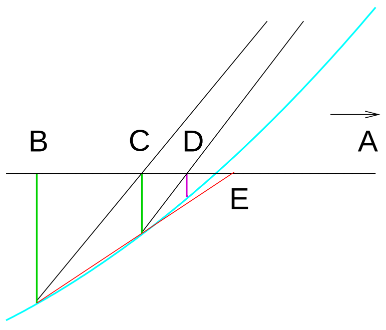

Figure 8.3: The Secant Method: extrapolate using the last two points found on the curve.

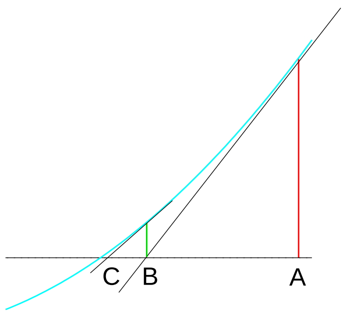

Figure 8.4: Newtons Method: the tangent helps find where the curve cuts the axis.

8.2.4 Newton's Method

Newton's method can be described briefly as approximating the graph of f near a point a by the tangent to f at $(a, f(a))$. The geometrical justification is shown in Fig 8.4. Starting with the approximation A to a root the next two steps go to B and C by drawing tangents appropriately. This method is faster than anything we have considered up until now, but

- it requires more of the function, namely that it be *differentiable* and that we can evaluate both f and f' at any point; and
- it is arguably significantly less stable than earlier methods.

Suppose we have an approximation a_0 to a root of f and that the true root is $a = a_0 + h$, so $f(a_0 + h) = 0$. By Taylor's theorem[2] we have

$$0 = f(a_0 + h) = f(a_0) + hf'(a_0) + \frac{h^2}{2!}f''(a_0) + \quad \text{terms of order at least } h^3$$

If a_0 is a good approximation to a, the error h is small; we assume this in what follows. Ignoring terms involving h^2 and above gives the approximation

$$0 \sim f(a_0) + hf'(a_0) \quad \text{and hence} \quad h \sim -\frac{f(a_0)}{f'(a_0)}.$$

Thus from a good approximation a_0 to the root a, we obtain a better approximation $a_1 = a_0 + h$. Applying this method repeatedly, gives a sequence of approximations $a_0, a_1, \ldots, a_n, \ldots$ where

$$a_{n+1} = a_n - \frac{f(a_n)}{f'(a_n)}. \tag{8.2}$$

This formula clearly gives trouble when $f'(a_n)$ is small; but in the derivation this is exactly the case when it may not be appropriate to ignore $h^2 f''(a_n)$ compared with $hf'(a_n)$.

And it is now clear that the secant method can be regarded as an approximation to Newton's method in which the derivative is replace by a Newton quotient.

Speed of Convergence

We can analyse the speed at which Newton's method converges. As above assume that a is a root of f and that $a_n \to a$ as $n \to \infty$, where a_n is given by equation 8.2. Let $\epsilon_n = a_n - a$, so $a_n = \epsilon_n + a$. Then from our iteration scheme, equation 8.2, we have

$$\begin{aligned}
a + \epsilon_{n+1} &= a + \epsilon_n - \frac{f(a+\epsilon_n)}{f'(a+\epsilon_n)}, \\
\epsilon_{n+1} &= \epsilon_n - \frac{f(a) + \epsilon_n f'(a) + \frac{\epsilon_n^2}{2!}f''(a) + \ldots}{f'(a) + \epsilon_n f''(a) + \ldots}, \\
&\sim \frac{\epsilon_n^2 f''(a) - \frac{\epsilon_n^2}{2!}f''(a)}{f'(a) + \epsilon_n f''(a)}, \\
\epsilon_{n+1} &\sim \epsilon_n^2 \left[\frac{f''(a)}{2f'(a)}\right]
\end{aligned}$$

and the process converges *quadratically* in the sense that the error at a given stage is the *square* of the error at the preceding stage. In contrast, for the bisection method, writing ϵ_n for the maximum possible error at that stage, we have *linear* convergence, since $\epsilon_{n+1} = \epsilon_n/2$.

[2]See for example MA2001 notes, Theorem 5.6

8.2.5 Common Sense

Any of the above methods can be expected to work. As suggested, Newton's method is likely to be the fastest, the bisection method is the simplest and least can go wrong with it, while the secant method has a degree of robustness by avoiding derivatives which means it is good in "difficult" situations. But there is no substitute for thinking about the problem first, and using understanding to help these methods. One of our original examples was that of solving the equation $b\cos(x) = \sin(x)$ for varying values of b. One such graph (with $b = 1.3$) is shown in Fig. 8.5, together with the two components parts. In this form it appears to be a relatively complicated situation.

Compare that with the next graphs, in Fig. 8.6. We have simply noted that the equation becomes much simpler to understand when divided by $\cos(x)$; we are seeking the intersection of $y = bx$ and $y = \tan(x)$. The behaviour and root locations become obvious!

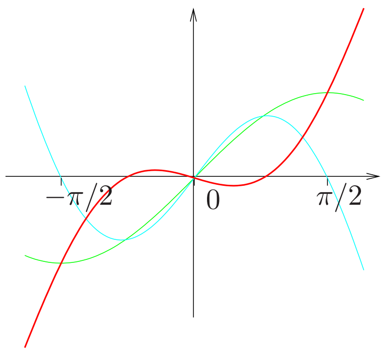

Figure 8.5: Graph of $y = b\cos(x) - \sin(x)$ together with $y = \sin(x)$ and $y = b\cos(x)$ with $b = 1.3$.

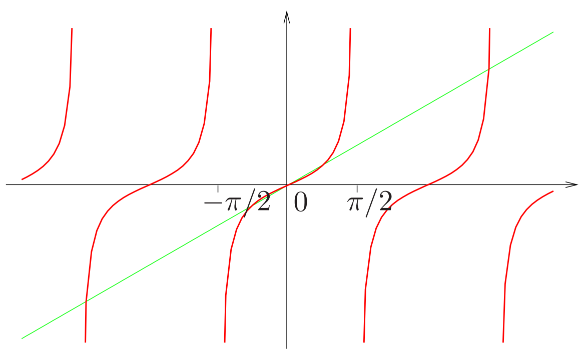

Figure 8.6: The graphs of $y = \tan(x)$ and $y = bx$ with $b = 1.3$.

As a second example, we return to the quadratic

$$x^2 - 20000x + 1 = 0$$

which we solved carefully using the quadratic formula. Recall that if the roots are α and β we have $\alpha + \beta = 20000$ and $\alpha\beta = 1$. Thus one root is necessary close to zero. We can then find it from the equation! Neglecting x^2 gives a first approximation $x = \frac{1}{2}10^{-5}$ immediately. rewriting the equation as $20000x = (1 + x^2)$ and using our first approximation gives

$$x = \frac{1}{20000}\left(1 + \frac{1}{4}10^{-10}\right) = \frac{10^{-5}}{2}\left(1 + \frac{1}{4}10^{-10}\right).$$

which is exactly the value we derived in 8.1.

And here is another example where it is more profitable to think than dash in with a general purpose method. Try it and see how you get on.

8.2. Example. A straight metal rail is 2km long and is firmly anchored at each end. An additional length of 20cm is welded into the middle of the rail so that it assume the shape of an arc of a vertical circle. How high above the ground is the top of the arc?

8.3 Implementation

In this section we describe how the algorithms of the previous section can be turned into working JAVA code. You have seen that JAVA is a flexible language; that there are usually many ways of doing things. So none of the suggestions in this section are *right*. However they give you an idea about *one* way to do things. And bear in mind that the methods suggested are ones which scale well. They will work for the short stand-alone programs that you write as practice, but will also stand the strain of much larger programs. Experience suggests that even short programs become much longer during their lifetime, so it is good practice to write robustly from the beginning.

8.3.1 Beginnings

So lets set out the basics. We are going to write a JAVA class to find roots of functions. I'll call it `Solver`. It will have a `main` method to help us test, and it will be in a package in the standard way.

```
package uk.ac.abdn.maths.mx3015.solve;
public class Solver {
    public static void main(String[] argv) {
    }
}
```

This is a minimal class; it can (and should) be tested to see that it runs. To remind you, the `(String[] argv)` argument is the argument required of a `main` method and puts anything on the command line after `Solver` into the variable `argv`. You can see an example of its use in `Echo.java` in the `samples` package.

The aim of this class is to find the root of a given function f using the various algorithms we discussed in Section 8.2. Given that we are working in JAVA, it looks like we need a method `bisectionMethod` (say) which will do this. Here we meet out first choice. Should this be a static method — something like `Math.sqrt` — or should it be an instance method? There is no universal answer; it depends on how `Solver` is to be used. In the end it comes down to whether we might want more than one `Solver` object around at the same time, and if so whether out method needs to manipulate variables associated with each object.

So we are forced to think about *objects*. I suggest it may be useful to have different objects associated with different functions and that our object can store information about the performance of the method. That way we may be able to understand and compare different iterative methods, which arguably makes sense here.

8.3.2 Constructors

This decision gives a clue about appropriate constructors. Each `Solver` is going to be associated with a particular function and will be responsible for find roots of that function. It looks as though we should store the function as a member variable which is initialised by the constructor. We expect to have functions available which we pass to this class to find appropriate roots, or "solve". Such functions will be objects; we've already seen examples such as the `Quadratic` and `Sine` classes.

We might choose to have different constructors for each such class, but this is not general. Indeed one point of iterative methods is that they use very little information about the function; just the ability to evaluate it at a point. This is precisely the reason we built the `Evaluable` interface, which can be used in argument lists just like a class. We are thus led to a constructor like

```
public Solver(Evaluable f) {
    this.f = f;
}
```

where we assume the member variable which holds the function is called `f`. You may prefer to call the constructor argument `f` as well and use the assignment `this.f = f;` instead.

At this stage we can test again that all is well and to do so we need an `Evaluable`. Since we will need one to do the final testing anyway, it is probably as well to sort this out now. Clearly one needs to create specific functions with which to test. Let me suppose we have one, say `ExampleFunction`; mine has a `toString()` method. Then the testing could be as simple as adding

```
ExampleFunction f = new ExampleFunction();
Solver solver = new Solver(f);
```

to the `main` method. When we have more in place, there may be a need for several different sorts of functions to help testing, but for the time being we just need enough to keep developing.

8.3.3 Instance Variables

We've already decided we need to have `f` as an instance variable. Are there any others? At this stage the main point to note is that new ones can be added as the need arises. Recall also the desirability of keeping instance variables private if at all possible. One we expect to use is some form of "tolerance" or acceptability; say `double tolerance`. If we make this private, we *may* need to write

```
public void setTolerance(double tol);
public double getTolerance();
```

but both are easy and can be written when necessary. Another variable can keep track of the number of times the function has been called. We are likely to need a `boolean success` which records the outcome, since we have already seen that some of the methods can fail. Another strategy would have been to throw exceptions when things go wrong; here we take the view that all our methods will complete "successfully".

This last choice means we may need to know why things went wrong if `success` is `false` after calling the method. Even with the bisection method, different things can go wrong

- the initial limits may not bracket a root;
- the function may have been called too many times; or
- we may not actually find a root.[3]

[3]Are you sure that this can't happen? What if you tried $\sin(1/x)$ on $[e^{-12}, 1]$?

One way to record such an outcome is by having a variable which records the outcome, perhaps called failReason. A first look suggests that this should be a String variable recording what went wrong. In fact subsequent use of the variable is much easier if it is an int, which takes different values for different types of failure. We could set fail = 1 if the initial limits did not bracket a root etc. Simply using these numbers is very prone to error, but using **symbolic constants**, usually written in capitals, is a way round this difficulty. These are naturally static variables, since we just need one copy however many objects there are. And because we wish to set them and then leave them unchanged, we declare them to be final which is JAVA's way of saying they should be constant. Thus somewhere in the declaration of class variables we would have

```
static final int NOT_BRACKETED = 1;
static final int TOO_MANY_EVALUATIONS = 2;
static final int NO_ROOT = 3;
```

They would then be used is situations like

```
if (iterationCount > MAXCOUNT ) {
    failReason = TOO_MANY_EVALUATIONS;
    success = false;
    return FAIL;
}
```

Remember that this is an example; it won't work for you unless you have declared and manipulated the appropriate variables. The variable FAIL is discussed below.

To sum up, there are good reasons for using both *static* and *instance* variables. We can add more as necessary.

8.3.4 Methods

Now we can get down to work. What is the **signature** of the method we will call bisectionMethod? In other words, what should its arguments be, and what should its return type be? I suggest the return type is easy; we will use it to return the root that has been found, so it should a be double. The argument list is hard to decide on. We *don't* need to specify the function whose root we are trying to find in the argument list, because it is an instance variable.

It is fairly clear that we need the two initial points, which I'll call them lowerLimit and upperLimit to start the method off. There are a number of other things we might wish to give the method, such as the maximum number of iterations allowed, the error tolerance and so on. Our choice of constructor was motivated by the possibility of running different root finding methods on the same function. This probably means we should *only* have method arguments which are specific to a given root, and we are led to the signature

```
public double bisectionMethod(double lowerLimit,
                              double upperLimit) {
```

A problem we need to face is that a method *must* always have a return type; we have to report a root even if we can't find one! We choose in this case to return the value 0.0; but to "explain" why within the program, we should have a static final variable called FAIL or something, which holds the value 0.0. This perhaps clears up an unexplained line in

Section 8.3.3. Probably the next step is to test again, with our method simply returning either `FAIL` or more interestingly

```
        return  (upperLimit + lowerLimit)/2;
```

Once again we find that the need to test intermediate states drives the `main` method; it looks as though it will be ready long before `bisectionMethod` where all the work is to be done.

8.3.5 Bisection

We now discuss the actual implementation of the method. At each step, either `upperLimit` or `lowerLimit` is reset, and the algorithm runs until they are sufficiently close. This suggests a loop as

```
        while ((upperLimit - lowerLimit) > epsilon) {
            // Calculate the midpoint
            // Decide which endpoint to reset
            // Update and check the number of iterations
        }
```

and this then gives a good handle on what initialisation to do to support the loop. Recall that because our function implements the `Evaluable` interface, we compute $f(x)$ by calling `f.at(x)` provided that `x` is a `double`. Note also that it could be expensive to evaluate the function; if your logic needs to use the value of f at a given point more than once, you should store the result of the first evaluation rather than re-computing. Also, remember to check for the various conditions that could cause a fail, and set the variables appropriately.

8.3.6 Other Methods

Having got a basic structure, little needs to be changed subsequently. We do however have a more complicated test of convergence. If the new point to be calculated is `xNew` (say) then convergence is determined by how big `f.at(xNew)` is; the crucial control structure is like

```
        while (Math.abs(midValue) > tolerance) {
            // Update the end points
            // etc
        }
```

One other difficulty arises when we write `newtonsMethod`, since we need the derivative of f as well as the ability to evaluate f. However, provided our example function implements the `Differentiable` interface, and so provides `at(double x)` and `dash(double x)` methods we have no problems. The derivative is available as `((Differentiable)f)`, since we must explicitly *cast* `f` back to being `Differentiable`. And if this method is attempted with a function which does not implement that interface, a `ClassCastException` will occur at run time.

Chapter 9

Numerical Integration

Integration proves so useful in practice that it is natural to ask how to integrate functions which you "can't" integrate in the sense of finding an antiderivative. Since the (definite) integral has a perfectly valid interpretation in terms of area, it is natural to use this to approximate definite integrals. Our aim is this section is to explore in a certain amount of detail how we can derive numerical integration, or **quadrature** routines which are automatic and yet relatively simple. Our problem can be stated as:

> Evaluate $\int_a^b f(x)\,dx$ with no knowledge of the function f, except that it is "relatively smooth".

There is no useful description that is more accurate, and in general there are functions which will defeat *any* algorithm. Geometrically, we interpret the integral as the area under the curve. One natural method then is "counting squares"; and already the difficulties become clear!

One way to be more precise is to say that f should be well approximated by its Taylor series; in other words, that f should behave like an approximating polynomial. In these circumstances we can hope to get an indication of the accuracy of the method. However a polynomial has neither horizontal nor a vertical asymptotes; yet many of the functions we wish to manipulate do have such features. So we can expect methods to do badly even when theory suggest they should do well!

9.1 Elementary Quadrature

Our first method is a very simply extension of the "counting squares' idea. We divide the interval $[a, b]$ into n equally spaced subintervals, each of length h, so $nh = b - a$. This distance h is fundamental in what we do; it is usually known as the **step length**. We the label end points of these subintervals by

$$a = x_0 < x_1 < x_2, \dots, x_n < x_n = b,$$

so $x_{i+1} - x_i = h$. Assume now we *only* know the values of f at these end points, say $y_i = f(x_i)$. One way to approximate the area under the curve is to add up the areas of the rectangles

$$A = h\sum_{i=0}^{n-1} y_i = h\sum_{i=0}^{n-1} f(a+ih), \tag{9.1}$$

and this is clearly capable of calculation on a computer. It is known as the **Rectangle Rule** for numerical integration.

Here is another description of this process:

- first approximate the function on one of the subintervals of length h;
- calculate the area under the curve obtained by replacing the function by its approximation; and
- add up all the areas obtained in this way.

We then recognise our first variant as that obtained by approximating f by a *constant*; an approximation of order 0 and adding up the areas of the resulting rectangles.

Clearly this can be generalised. In the spirit of Taylor's theorem, we can approximate f by a linear function, by a quadratic function and so on. The approximation of order one treats f as linear, and the resulting approximation is a trapezium. Calculating the area is elementary; it is $h(y_1 - y_0)/2$.

Our next step is to approximate f as quadratic, of the form $f(x) = ax^2 + bx + c$. We choose a, b and c so the values y_0, y_1 and y_2 are correct and integrate the resulting function. This is straightforward but not immediate. In order to simplify the manipulation we use the fact that the area will be the same wherever we place the origin. Here is the result.

9.1. Proposition. *Let f be a quadratic function which passes through the three points $(-h, y_{-1})$, $(0, y_0)$ and (h, y_1). Then*

$$\int_{-h}^{h} f(x)\,dx = \frac{h}{3}\left[y_{-1} + 4y_0 + y_1\right].$$

Proof. This is a simple calculation; the hard bit is in setting it up as above. Assume that $f(x) = ax^2 + bx + c$. Then since it passes through $(0, y_0)$ we have $y_0 = f(0) = c$. It passes through the other two given points, so

$$\begin{aligned} y_{-1} &= ah^2 - bh + c, \\ y_1 &= ah^2 + bh + c. \end{aligned}$$

Subtracting these shows that $2bh = y_1 - y_{-1}$, giving b. Adding, we have

$$2ah^2 = y_1 + y_{-1} - 2y_0, \qquad a = \frac{1}{2h^2}\Big(y_1 + 2y_0 + y_{-1}\Big),$$

on using the value of c we have just obtained. We now simply integrate and use these values.

$$\begin{aligned} \int_{-h}^{h} f(x)\,dx &= \int_{-h}^{h} (ax^2 + bx + c)\,dx = \left[\frac{ax^3}{3} + \frac{bx^2}{2} + cx\right]_{-h}^{h} \\ &= \frac{a}{3}2h^3 + c.2h = \frac{h}{3}\left[y_h + y_{-h} - 2y_0 + 6y_0\right], \end{aligned}$$

and this gives the result claimed. □

An interesting fact emerges from the proof; note that even had we used a *cubic* approximation, the final result would have been no better, because terms of odd degree, like the x term above, have no contribution. In other words, to do better we would need an order 4 approximation. In practice is is normal to stop with the linear approximation, or **Trapezium rule** or the quadric approximation, leading to **Simpson's Rule**. A summary of the results we have derived is given in Table 9.1.

Order	Rule Name	Area of Approximation	Error
0	Rectangle	hy_0	$\frac{h^2}{2}y'(c)$
1	Trapezium	$\frac{h}{2}(y_0+y_1)$	$-\frac{1}{12}h^3y''(c)$
2	Simpson	$\frac{h}{3}(y_0+4y_1+y_2)$	$-\frac{1}{90}h^5y^{(v)}(c)$

Table 9.1: Integrating by approximations of different orders.

We make not attempt to justify the "Error" column. It is simply there to show that much more can be done with a more complete analysis, using Taylor's Theorem with an appropriate form of the remainder.

These formulae have assumed we use a single approximation for f throughout the range of integration, but our original setup split it into n parts. Returning to that scenario leads to the following improvement of Equation 9.1.

$$A = \frac{h}{2}\Big[(y_0+y_1)+(y_1+y_2)+\cdots+(y_{n-1}+y_n)\Big] = \frac{h}{2}\Big[y_0+y_n+2\sum_{i=1}^{n-1}y_i\Big] \tag{9.2}$$

This can be described as

$$\frac{h}{2}\times\big(\text{first} + \text{last} + \text{twice the sum of the rest}\big)$$

and this is the form in which the **Trapezium Rule** is normally quoted.

In the same spirit we can derive **Simpson's Rule**. Here it is convenient to assume there are an *even* number of subintervals, or equivalently that the end points are y_0 and y_{2n}. Then

$$\begin{aligned} A &= \frac{h}{3}\Big[(y_0+4y_1+y_2)+(y_2+4y_3+y_4)+\cdots+(y_{2n-2}+4y_{2n-1}+y_{2n})\Big] \\ &= \frac{h}{2}\Big[y_0+y_{2n}+2\sum_{i=1}^{n-1}y_{2i}+4\sum_{i=1}^{n}y_{2i-1}\Big]. \end{aligned}$$

This can be described as

$$\frac{h}{3}\times\big(\text{first} + \text{last} + \text{twice the sum of the evens} + \text{four} \times \text{the sum of the odds}\big)$$

which is the form in which **Simpson's Rule** is normally quoted.

9.2 Adaptive Quadrature

All of the previous section was initially developed even before the invention of hand calculators. Typically there would be some preliminary investigation to decide on the step length and then the calculation would proceed. With computer assistance we can afford to be more systematic. Here is a typical algorithm.

```
Set the step length appropriately;
Calculate a first approximation to the area;
do {
    halve the step length;
    calculate a new approximation to the area;
} while (difference between old
         and new areas is not small enough);
```

Such an algorithm begins to feel like Newton's method in that we have a way of examining how well we are doing. Halving the step length is sensible, since we can in principle make use of all the old function evaluations when we do our new calculation of area. We show how this works with the Trapezium Rule, but even then the general case is notationally a little complicated. To simplify, we look at the transition from 3 to 5 ordinates. We have a first approximation A_1 to the area of the form $A_1 = \frac{H}{2}\big(Y_0 + 2Y_1 + Y_2\big)$ and a second approximation with $h = H/2$ and ordinates $y_0 = Y_0$, $y_2 = Y_1$, $y_4 = Y_2$ and new ordinates y_1 and y_3. Then

$$\begin{aligned} A_2 &= \frac{h}{2}\big(y_0 + y_4 + 2(y_1 + y_2 + y_3)\big) = \frac{h}{2}\big(y_0 + y_4 + 2y_2\big) + h\big(y_1 + y_3)\big) \\ &= \tfrac{1}{2}A_1 + h \times \text{sum of the new ordinates.} \end{aligned}$$

In general, we have

$$A_{n+1} = \tfrac{1}{2}A_n + \text{new step length} \times \text{sum of the new ordinates.}$$

which gives a very simple updating formula.

There is an additional twist which gives improved results. Assume we are evaluating a given integral, whose true value is A and whose approximate values, obtained using 2^n subdivisions of the interval of integration, are A_n. We write H for the step length with 0 subdivisions and h for the step length with 1 subdivision, so as before, $H = 2h$. Then because we know the error term starts off with terms of order 3, we have

$$\begin{aligned} A &= \frac{H}{2}(y_0 + y_2) + C^{(0)}H^3 = A_0 + C^{(0)}8h^3 \\ A &= \frac{h}{2}(y_0 + 2y_1 + y_2) + 2C^{(1)}h^3 = A_1 + 2C^{(1)}h^3 \end{aligned}$$

accurate up to terms involving h^4. Note that the second of the two estimates accumulates an error of $C^{(1)}h^3$ on *each* of the two subintervals.

Now we make the (not implausible) assumption that all the error constants are the same, so in particular $C^{(0)} = C^{(1)}$. Then

$$4A - A = 3A = 4A_1 - A_0 \quad \text{and} \quad A = \frac{4A_1 - A_0}{4 - 1}$$

In general, we have $A = (4A_n - A_{n-1})/(4-1)$, and we deduce the value of the actual integral A from the two most recent estimates. This is known as **Richardson extrapolation**.

You can see how the formula works by thinking out what would be happening in a different case. Had the formula been $(4A_1 + A_0)/(4+1)$, we would have been averaging the two approximations, with the average weighted more towards the most recent. In particular, this would have been an interpolation between two known values. The effect of the correct formula is take the new value, and then move even further away from the old one; hence it is an *extrapolation*.

9.3 Romberg's Method

The theoretical basis for Richardson extrapolation is not clear unless our function is indeed a polynomial. In this section we discuss an extension of Richardson extrapolation, known as **Romberg's Method** which seems to work assuming only that the second derivative of the function to be integrated is bounded.

We construct a triangle as follows

$$\begin{array}{lllll}
A_{0,0} & & & & \\
A_{0,1} & A_{1,0} & & & \\
A_{0,2} & A_{1,1} & A_{2,0} & & \\
A_{0,3} & A_{1,2} & A_{2,1} & A_{3,0} & \\
\cdots & \cdots & \cdots & \cdots & \cdots
\end{array}$$

The first column consists of trapezium estimates, where the estimate $A_{0,n}$ has step length 2^{-n} times that of $A_{0,0}$. Now construct successive Richardson extrapolates $A_{1,1}$, $A_{1,2}$ and so on, noting that we can't start until the second row, since we need *two* values in the previous column to be available.

In general we define

$$A_{n,k} = \frac{1}{4^n - 1}\left(4^n A_{n-1,k+1} - A_{n-1,k}\right)$$

In other words, we use two successive Richardson extrapolations to get a "second" order extrapolation and so on. Note the factor 4^n indicating that the contribution of the previous best estimate is weighted less and less as we get higher and higher order extrapolations. The process stops when two successive estimates on the diagonal agree.

The algorithm is clear, although the justification is quite hard. However the improvement can be spectacular; I've just seen an example when 17 function evaluations using Romberg's method gave greater accuracy than 16385 evaluations using the adaptive trapezium rule.

9.4 Other Methods

There are many other ways of integrating. One such, known as **Gauss's Method** is very effective. It abandons our assumption that the ordinates shall be equally spaced, and instead uses the freedom to choose them "optimally spaced" in the sense that they are chosen to minimise the expected error between the actual integral and the integral of the resulting polynomial approximation.

We also note here a method which is very different in spirit, and which has only become popular recently with the availability of enormous computing power. Suppose we have a circle inside and just touching a square. If we choose a point at random within the square, the probability that it lands inside the circle will be $\pi r^2/4r^2$. Looking at this backwards, if we can *measure* this probability, then that will give the area of the circle. This method is known as **Monte Carlo** integration because random numbers are used to obtain frequencies which then approximate the probabilities. We discuss this method in greater detail in Section 10.8.

9.5 Implementation

In thinking about the design of a class you need some idea of how it will be used. In writing an `Integrator` class, I'm thinking of using it to compare some of the methods we've just discussed. So it seems natural to have a different `Integrator` object for each definite integral to be evaluated. As with the `Solver` classes, we are trying to work with a general function; simply having available the `at(double x)` method, so we may as well make use again of the `Evaluable` interface. These decisions fix the bare bones of the class.

```
package uk.ac.abdn.maths.mx3015.integ;
import uk.ac.abdn.maths.mx3015.util.Evaluable;
public class Integrator {
    Evaluable f;
    double lowerLimit;
    double upperLimit;
    public Integrator(Evaluable f,
                      double lowerLimit,
                      double upperLimit) {
        this.f = f;
        this.lowerLimit = lowerLimit;
        this.upperLimit = upperLimit;
    }
}
```

If you add an empty `main` method you can test this and confirm that all is well. The next step would be to exercise the constructor, which means we need an `Evaluable`. I've taken advantage of the new package to write a different `ExampleFunction` class. The first change is that it no longer claims to implement the `Differentiable` interface; there is no need, since at no point do we rely on access to the derivative. You can also more clearly see why we want an instance or member of a class this time, because in addition to the rather pointless `name` variable, we have the instance variable `callCount`; different instances of the class will have different values of this counter even though the return value of their `at` method is the same.

You can always change the return value in order to integrate a different function. In "real life" you are expecting to get the `Evaluable` delivered from some other part of the application; this bit just has to return an approximation to the integral.

```
package uk.ac.abdn.maths.mx3015.integ;
import uk.ac.abdn.maths.mx3015.util.Evaluable;
public class ExampleFunction implements Evaluable {
    public String name;
    int callCount = 0;
    public ExampleFunction(String myname) {
        name = myname;
        // callCount will be zero automatically
    }
    public void resetCount() {
        callCount = 0;
    }
    public double at (double x) {
        return Math.exp(x) - 4*x*x*x;
    }
    public String toString(){
        return name;
    }
```

With this in hand we can create a simple test harness in the `main` method which just creates an Integrator. As usual we create a variable name of the appropriate type and then create the object itself by calling the constructor using the `new` operator. And again as usual, we choose to do these two steps in one. So the start of our `main` method becomes

```
public static void main(String argv[]) {
    ExampleFunction f = new
        ExampleFunction("the test function Math.exp(x) + 4*x*x*x");
    Integrator integral = new Integrator(f,0.0,1.0);
    // ...
}
```

Finally we come to the implementation of the methods to do the actual approximate integration. The method `trapeziumRule` should be one which uses the Trapezium rule to integrate. We don't need to pass the function or the end points to it because they are already available within the class. So we simply have to pass the step size, or some surrogate. I've chosen the *number* of steps to take rather than the steplength, since there are fewer possibilities of being given a stupid value. And the return value is clear; we simply want the value of the integral. This time I've not chosen to put a limit on the number of evaluations to use or otherwise check if the answer is sensible. Clearly such code could be added, and would probably create the need for other instance variables, such as `success`. In the simple case then the signature of the method is

```
double trapeziumRule(int steps) {
```

and outside the class if our Integrator is called `integral`, we would refer to the value as `integral.trapeziumRule(n)` where `n` was the number of steps to be used.

9.6 Common Sense

Like almost all applications to numerical analysis, it is essential to look at the functions involved *before* applying such package routines. Be aware that difficulties for quadrature routines occur when the function starts *not* behaving like a polynomial. A simple example is the problem of evaluating

$$\int_0^1 \sqrt{x}\cos x\, dx.$$

I *think* this can't be done analytically. Certainly it looks a mess; we really have no tools if "integration by parts" fails. Numerically it is a mess but for very different reasons. The numeric problem occurs at 0, where the function behaves like $\sqrt{x}$, which has the y axis as a vertical asymptote. Elsewhere the function is well behaved and accuracy will come very quickly.

An obvious remedy is to combine the two approaches, using each where it is strong. This is the technique known as **removing the singularity**. In this case, why not evaluate

$$\int_0^1 \left(\sqrt{x}\cos x - \sqrt{x}\right)\, dx$$

instead? The two functions differ just by $\sqrt{x}$ which *can* be integrated analytically so we can easily calculate the difference between the integrals. Yet in its second form the integral gives *no* trouble numerically.

As a second example consider the following two integrals

$$I_1 = \int_{0.01}^1 \frac{dx}{\mathrm{e}^x - 1}, \qquad I_2 = \int_{0.01}^1 \left(\frac{1}{\mathrm{e}^x - 1} - \frac{1}{x}\right)\, dx.$$

Recalling the Taylor expansion for e^x about 0, we expect the denominator to behave a bit like x; in other words we expect the function to blow up as we approach zero. Again we can cope with the "difficult" bit of the function analytically, and the second function integrates easily. In my testing, the first integral took 125 *times* as many function evaluations as the second. I can think of few other ways to get such a dramatic run-time speedup.

Chapter 10

Simulation and Random Numbers

10.1 Simulation

The advent of powerful computers on almost every desk is in the process of fundamentally changing the way we think about many problems. One of the more important *uses* of mathematics was in modelling complex situations in order to make predictions. Typically the situation would be simplified and abstracted; this model was then taken as the basis for performing calculations and the results were mapped back to the original problem domain.

10.1. Example. You are offered the chance to play the following game. You and your opponent each flip two £1 coins and put them down on the bar.[1] If you each show the same number of heads and tails (both of you have two heads, two tails or each have one head and one tail) , your opponent gives you both of his coins; otherwise you give him *one* of yours.

Should you play?

You accept and find it isn't too bad a bet. You are then offered an "improvement" in that the game is played with three coins and again you take all the coins on winning. Should you go along with this and so make more money?

It isn't too hard to do this using traditional probability calculations. However the alternative now is to simulate the game on a computer and simply look at the outcome. In other words, to take the description of probability as "relative frequency" literally and perform many trials in order to estimate the expected outcome. Although "lazy" this approach has the advantage that no analysis of solution mechanisms is required. You still have to understand the problem, but you can use computation as a substitute for the mathematical analysis of the solutions.

Here is a much less artificial example from the area of digital image processing. The colour of a "skin" pixel — a pixel where the image is, for example, part of a person's cheek seems to be fairly characteristic. Indeed a good model, in an appropriate colour space, consists of a finite number of Gaussian variables; the number in the "mixture" and the individual means and variances are then found by "training" with examples. Given such a trained model, one can simulate the occurrence of skin pixels; then calculate the probability that a given position in colour space is generated by this model; and finally classify new

[1] You wouldn't even think of playing such a stupid game if you weren't in a bar!

image pixels as skin or not, depending on whether they have a reasonable probability of being generated.

The “game” example above involved a very explicit element of chance; a coin was flipped. But a large amount of our understanding of the world, like the “skin” example is a combination of explicit natural laws and such chance events, used to decide between outcomes when the choice appears not to be deterministic. The way we model such chance events on the computer is through the use of **random numbers**; numbers which are chosen “at random”

10.2 What is a Random Number?

How does a computer, which is a totally deterministic device, make ‘random’ choices? What is a random choice for that matter? A classic reference here is Knuth (1981); much of what follows is taken from there.

We will see later that most of the situations that require random choices to be made can be handled once we have “random numbers” available. So, what is a random number? You will realise that this is a silly question as soon as you rephrase it in the form “is 2 a random number?”. Randomness is not a property of individual numbers. More properly it is a property of infinite sequences of numbers.

I am now going to try to clarify what we mean by a random sequence of numbers. To keep things definite I will assume that we are talking about real numbers, say `doubles`, in the range $[0, 1)$.

What we are eventually aiming to define and construct is what is known as a random number generator. This is a method or black box, which, when given an input value **seed** (usually an integer), produces as output an infinite random sequence of real numbers $\{x_k\}$ with $x_k \in [0, 1)$.

$$\texttt{seed} \quad \rightarrow \quad \boxed{\text{Math.random()}} \quad \rightarrow \quad \{x_k\}$$

Furthermore, we require that different seeds produce different sequences and that, in some sense, ’related’ seeds do not produce related sequences.

As far as I am aware, nobody has ever given an entirely convincing definition of the term “random sequence”. On the other hand, everybody has a common-sense idea of what it means. We say things like: “the sequence should have no pattern or structure”. More directly we might say that knowing $x_1, \ldots, x_n$ tells us nothing about $x_{n+1}, \ldots$.

One of the simplest attempts at a mathematical definition goes as follows.

A sequence $\{\mathbf{p_k}\}$ in $[0, 1)^n$ is **uniformly distributed** on $[0, 1)^n$ if, for any box (or Cartesian product of intervals)

$$A = [l_1, r_1] \times [l_2, r_2] \times \cdots \times [l_n, r_n]$$

in $[0, 1)^n$,

$$\lim_{k\to\infty} \frac{|\{\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_k}\} \cap A|}{k} = |A|$$

where $|A|$ is the volume of A. This is saying that a “fair” or “correct” proportion of the sequence happens to fall in the box A; if A has one tenth of the volume of the unit n - cell, then one tenth of the random sequence should lie in A.

A sequence $\{x_k\}$ on $[0,1)$ is n-**distributed** on $[0,1)$ if the sequence $\{\mathbf{p_k}\}$ given by

$$\mathbf{p_k} = (x_{kn}, x_{kn+1}, \dots, x_{kn+n-1}) \in [0,1)^n$$

is uniformly distributed on $[0,1)^n$.

We will require that a random sequence in $[0,1)$ be ∞-distributed on $[0,1)$, i.e. k-distributed for all k.

This turns out to be an inadequate definition of randomness, but it is good enough for all practical purposes. In particular, an ∞-distributed sequence will pass all the standard statistical tests for randomness. (If you want to be thoroughly perverse you could argue that the fact that it passes all such test is itself evidence of a certain non-randomness!)

10.3 Generating Random Numbers

Like most computer languages, JAVA has a software 'random number generators'. These do not generate random sequences in the above sense (almost by definition, no algorithmic process can possibly do so). Instead, they produce what are called *pseudo-random numbers*. These are sequences of numbers that are sufficiently 'random' to pass all current statistical tests for randomness in 'realistic' cases.

Some quotations are in order here:

> "A random sequence is a vague notion embodying the idea of a sequence in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests, traditional with statisticians and depending somewhat on the use to which the sequence is put." (Lehmer 1951).
>
> "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." (Von Neumann 1951).
>
> and finally to make the point about (Knuth 1981)
>
> "If you think you're a really good programmer,...read [Knuth's] Art of Computer Programming....You should definitely send me a resume if you can read the whole thing." (Bill Gates, as quoted on the Amazon web site!)

There are many algorithms available (see Knuth (1981) for some horror stories) but most languages now use what are called linear congruential generators.

A **Linear Congruential Generator** is determined by three positive integers a, b, m. We call a the **multiplier**, b the **increment** and m the **modulus**. It produces its pseudo-random sequence as follows:

- generate a sequence of integers $\{n_k\}$ for $(0 \le n_k < m)$ as follows:
  - specify n_0 - the **seed**; and
  - generate the rest by defining $n_{k+1} = an_k + b \pmod{m}$.
- then let $x_k = \frac{n_k}{m} \in [0,1)$.

This looks ridiculous. The x's are rational and there are only m values available for them, namely k/m for $k = 0, \dots, m-1$. However, in practical cases, m is usually something like

$$2^{32} = 4,294,967,296$$

and a is chosen carefully to give good results. The choice of a is critical, the choice of b less important.

It can be shown that the following conditions should be satisfied if we are to have a good generator:

- The modulus m should be large. This is usually not a problem because values like the one given above are very convenient for computer arithmetic.
- If, as often happens, m is a power of 2 then pick a such that $a \bmod 8 = 5$. This will have the effect of maximising the period of the necessarily cyclic sequence produced. The choice of value for a is the most delicate part of the whole design. Any suggested value for a should be submitted to a thing called the 'spectral test' before being used.
- The increment b should be coprime to m. A value like $b = 1$ is perfectly acceptable.

An acceptable choice of values would be:

$$m = 2^{32} \qquad a = 1664525 \qquad b = 1.$$

The theory behind these conditions is discussed in much more detail in Knuth (1981, Section 3.2.1).

I'm quite serious about all the reservations. Knuth's book just cited was well known long before the development of JAVA. Indeed the JAVA documentation quotes the exact section of Knuth's book that I've just quoted. Yet in the description of the `nextDouble` method of the class `java.util.Random`, we find the comment:-

> [In early versions of Java, the result was incorrectly calculated as:
>
> ```
> return (((long)next(27) << 27) + next(27))
>           / (double)(1L << 54);
> ```
>
> This might seem to be equivalent, if not better, but in fact it introduced a large non-uniformity because of the bias in the rounding of floating-point numbers: it was three times as likely that the low-order bit of the significand would be 0 than that it would be 1! This non-uniformity probably doesn't matter much in practice, but we strive for perfection.]

10.4 Specialised Random Numbers

There are many 'random processes' that are useful in various kinds of simulation. Most, if not all, of these can be generated once you have available a random number generator of the kind discussed above. I'm going to discuss briefly how to do this, but first we review the facilities within JAVA.

In fact JAVA takes the same view, offering a "no frills" random number generator in the method `Math.random()` (in the `java.lang` package, so always available in the same way as `Math.sqrt()`) and a more extensive `Random` class.[2] As usual the documentation that comes with the API is thorough, and provides the last word. You can locate it simply by

[2]In effect `Math.random()` is just a simple interface to the more elaborate class.

searching for `Random`, but it may be quicker to remember that it is one of the classes in the `java.util` package.

Each separate instance of the class gives a different random number generator. If it is a “true” random number generator, then it shouldn’t matter whether you have one generator used all the time or you use several separate generators. However it may be conceptually easier to think of (say) two coins as each having their own random number generator. One of the constructors has no arguments, so after a call to

```
Random rand = new Random();
```

it can be used as required. Some of the methods are given in Table 10.4 which gives all that is usually required.

Method	Description
`double nextDouble()`	returns a `double` “random” value *uniformly* distributed in the interval $[0, 1]$;
`int nextInt(int n)`	returns an `int` *uniformly* distributed between 0 and $n - 1$ inclusive;
`double nextGaussian()`	returns a `double` which is *normally* distributed with mean 0.0 and standard deviation 1.0.

Table 10.1: Some methods in the `java.util.Random` class.

Sometimes a random number generator can be *too* random. It often happens when debugging that we want to work with a *fixed* random sequence. Of course this doesn’t make sense, but you can see the advantages when debugging of knowing that the same thing happens each time. For this reason there is a constructor in which the seed can be set explicitly thus:-

```
public Random(long seed)
```

You can see from the original description of a random number generator that provided we start with the same seed, we always get the *same* sequence of random numbers. In fact the no argument constructor is defined as follows:-

```
public Random() {
    this(System.currentTimeMillis());
}
```

in other words, it simply takes the current time as a seed. This differs between runs, so different pseudo-random sequences are delivered. Often successive runs will have closely related seeds; our requirement that similar seeds should not produce related random sequences is thus important here.

10.4.1 Generating Random Reals

I suppose throughout this, that we work with the simple `Math.random()` method which returns `doubles` uniformly distributed between 0 and 1.

It is common to want a stream of random numbers uniformly distributed between, say, A and B — rather than between 0 and 1. This is easily arranged:

```
double randomAB(double a, double b) {
    return a + (b-a)*Math.random();
}
```

i.e. just scale and shift. This does not affect the randomness in any way. Let me give a partial argument to justify that claim. We know that `Math.random()` generates a sequence $\{x_k\} \subset [0, 1)$ which is uniformly distributed. We want to show that the sequence produced by `randomAB(X,Y)` has the same kind of property with respect to the interval $[X, Y)$. It is easiest to say this in probability language. Let $[u, v)$ be an interval in $[X, Y)$. We want the probability that a number produced by `randomAB(X,Y)` falls in this range to be $(v - u)/(Y - X)$ — i.e. the subset gets precisely its fair share of elements. Now

$$\begin{aligned} u \le \text{xrand}(X, Y) < v \quad &\iff \quad u \le X + (Y - X) * \texttt{Math.random()} < v \\ &\iff \quad \frac{u - X}{Y - X} \le \texttt{Math.random()} < \frac{v - X}{Y - X} \end{aligned}$$

Now, you can easily check that this range for `Math.random()` lies in $[0, 1)$. The length of the range is

$$\frac{v - X}{Y - X} - \frac{u - X}{Y - X} = \frac{v - u}{Y - X}$$

So the probability that `Math.random()` falls in this range is, by the uniform distribution of `Math.random()`, $(v - u)/(Y - X)$. So this is the probability that `randomAB(X,Y)` gives a value in $[u, v)$. This is the value that we wanted.

10.4.2 Generating Random Integers

Suppose we want to generate a random stream of integers in the range $n, \ldots, m$. The method to do this could be written as:

```
static int randomInt(int n, int m) {
    return (int) Math.floor(n + (m-n+1)*Math.random());
}
```

This works because:

- the result is certainly an integer in the range $n, \ldots, m$, since `Math.random()` < 1.
- To check randomness we just have to be sure that each integer is equally likely to be chosen at each stage. But k is chosen if

  $$k \le n + (m - n + 1) * \texttt{Math.random()} < k + 1,$$

  which can be rewritten as

  $$\frac{k - n}{m - n + 1} \le \texttt{Math.random()} < \frac{k + 1 - n}{m - n + 1}.$$

The length of this range, which lies in $[0, 1)$, is independent of k (subtract and see). So, by the definition of uniform distribution for `Math.random()`, all values of k are equally likely to be chosen.

If you feel energetic you can also prove that the output of `randomInt(int n, int m)` has ‘higher order randomness’ in the sense that, for example, all pairs (i, j) with $n \leq i, j \leq m$ are equally likely to be chosen.

10.5 Choosing ‘with probability p’

Very often random algorithms require you to ‘choose this object with probability p’ or ‘take this option with probability p’. What does this mean and how do we do it? The basic meaning is that in a large number of such decisions you will perform the action a proportion p of the time. But we’re not doing it lots of times, we’re just doing it once. Well, in that case you have to make a random decision as to whether or not to perform the action and the decision must be biased so as to give the correct proportion ‘in the long run’.

The simplest case is that of choosing with probability 1/2. This means that we are equally likely to choose or not choose. The traditional method is to flip a coin. If we had to make a choice with probability 1/6 we might decide to throw a dice and take the action if it shows a six.

The general case can be handled by means of a standard random number generator. Suppose we want to take an option ‘with probability p’. Then we use our random number generator to generate a random real in $[0, 1)$. If this number is less than p we take the option, otherwise we do not. This is correct, in terms of probability, because $[0, p)$ is proportion p of $[0, 1)$.

```
static boolean choose(double p) {
    // Returns true with "probability" p
    return (Math.random() < p);
}
```

So, for example, flipping an unbiased coin can be simulated by:

```
static boolean flip() {
    // Returns true if the coin came down "heads"
    return (Math.random() < 0.5);
}
```

It is fairly easy to extend this to making a choice with given probabilities between a finite number of things. For example suppose we have choices C_1, C_2 and C_3 which have to be made with probabilities p_1 p_2 and p_3. Assume the choices are both exclusive and exhaustive, so we must make *exactly* one of them. This means that $p_1 + p_2 + p_3 = 1$. The algorithm, in pseudocode is then

```
double p = Math.random();
if (p < $p_1$) {
    choose $C_1$
} else if (p < $p_1$ + $p_2$) {
    choose $C_2$
} else {
    choose $C_3$
}
```

The reason I've used **pseudocode** here is to make the shape of the algorithm clear. I've used correct JAVA to show the control structure, but to avoid getting bogged down in the (type-safe) details of exactly what the choices are, and how they should be returned, I've simply written "choose C_1" and so on. I hope the fact that I returned to mathematical notation, together with the lack of semicolons alerted you to this.

10.6 Choosing without Replacement

We now have a collection of things and we wish to choose *all* of them; we are more concerned about the order in which they come out, rather than in which are chosen. Or we want to choose part of a collection of things. The crucial point is that once something has been chosen, it is no longer available to be chosen again; hence the notion of sampling **without replacement**. We assume the objects are labelled by integers in some way; typically they will be held in an array say `things[]`. We then choose an integer `i` between `0` and `things.length`; this corresponds to choosing `thing[i]`.

We could repeat the random choice, but may end up picking `i` again. We could maintain a list of things we've chosen and only accept the choice if it is a new one. If we only want to choose k of the n items and k is very small compared with n this may be sensible, but in general, and certainly when we wish to choose all the items, this method looks silly. Instead we shorten the effective length of the array so we always make an allowable choice. Here is the step in which we have `k` items left to choose from:

- choose one thing (say `thing[i]`) from the first `k` elements of the array;
- then copy `thing[k-1]` to `thing[i]`;
- reset `k` to `k-1`, so next time we can choose any of the first `k-1` elements.

Of course we *might* have chosen the last element of the array at random. In that case the copy does nothing useful, but it certainly does no harm.

This method works sensibly even if we only want to choose k of the n items; however the overhead of creating the array may outweigh the cost of repeated choices if $k << n$. Finally note that there is no need to choose the last item from the array since there is no choice left!

10.7 Conveyor Belt Sampling

It is often the case that the set to be sampled from is far too big to be held in memory. We have to be able to make our selection simply by looking at one item of the set at a time. The obvious analogy is that of a quality control inspector standing alongside a conveyor belt and having to choose, let's say, 10 items from every 1000 that pass him. He has to be able to do this without backing up the conveyor belt. Here is the algorithm written as a static method:

```
/** Choose k samples from n. **/
static int [] choose(int k,int n) {
    int [] choice = new int[k];
    int j = 0; // the number we've chosen
    int seen = 0; // the number we've examined
    while (j < k) {
        if (Math.random()*(n-seen) < k-j) {
            choice[j++] = seen;
        }
        seen++;
    }
    return choice;
}
```

In this it is assumed that the items to be sampled are being counted and the "choice" is the item numbers to be sampled. In the above we return the list of choices as an array, but it would be easy to "use" the indices as they are produced.

The logic of this method is that if, at the ith item, we have so far chosen j items then we have $k-j$ items left to choose from the remaining $n-i+1$ items (including the ith). So it seems reasonable to choose the ith item with probability

$$\frac{k-j}{n-i+1}$$

and that's what the algorithm does, using the variable `seen` instead of $i-1$. That seems too obvious to be true!

This is a simple routine; the real problem lies in trying to justify its claim to produce a random sample. Indeed, it is not obvious that the routine will succeed in getting its full sample before it runs out of input!

We cannot end up with more than k items because we stop the loop if we have got k. It is thus enough to show that we cannot end up with *fewer* than k. But at any given choice the probability can be written as

$$p = \frac{\text{Number remaining to be chosen}}{\text{Total number remaining}},$$

and this becomes, and stays 1 precisely at the point where we need to take all the remaining items to make sure we get enough.

Let's now look at the claim that it produces *random* samples. Consider samples of size k. There are $\binom{n}{k}$ of these in all and we want each of them to be an equally likely selection.

Let me start by looking at the simple case $k=2$. What is the probability of our choosing the subset $\{x_a, x_b\}$ from $(x_1, x_2, \dots, x_n)$?

We must *not* pick $x_1, x_2, \dots, x_{a-1}$, we *must* pick x_a, we must *not* pick $x_{a+1}, \dots, x_{b-1}$, we *must* pick x_b and we must *not* pick $x_{b+1}, \dots, x_n$. Turn all that into a probability calculation and you get the probability

$$p = \prod_{i=1}^{i=a-1} \left(\frac{n-i-1}{n-i+1}\right) \cdot \frac{2}{n-a+1} \cdot \prod_{i=a+1}^{b-1} \left(\frac{n-i}{n-i+1}\right) \cdot \frac{1}{n-b+1} \cdot \prod_{i=b+1}^{n} \left(\frac{n-i+1}{n-i+1}\right).$$

This looks horrible at first sight, but closer examination reveals all.

The denominators run in sequence from n down to 1. So the overall denominator is $n!$. We have a 2 and a 1 on the top (at the chosen points) and that gives us a factor 2!. Finally, the rest of the numerator is

$$(n-2)(n-3)\cdots(n-a)\cdot(n-a-1)(n-a-2)\cdots(n-b+1)\cdot(n-b)\cdots 1 = (n-2)!$$

Notice that the sequence steps over the joins properly. This gives us the factor $(n-2)!$. So the probability is

$$p = \frac{2!\,(n-2)!}{n!} = \binom{n}{2}^{-1} \quad \text{as required.}$$

The general argument goes in just the same way. Suppose we specify a subset of k elements. Then we work out the probability of choosing this subset, as above. We find that the denominators run from n down to 1 once more — giving the $n!$. The 'chosen' elements give us the factor $1.2.3\ldots k = k!$. There remain $n-k$ numerators. They are all different, the largest is $n-k$ and the smallest is 1. So they must be precisely all the numbers from 1 to $n-k$ and so give the factor $(n-k)!$. Thus the probability of choosing this subset is

$$p = \frac{k!\,(n-k)!}{n!} = \binom{n}{k}^{-1}.$$

So all subsets of a given size are equally likely to be chosen, and we definitely have a random sample.

10.8 Monte Carlo Integration

We are now going to turn things on their heads. Given that we *have* a random number generator, can we use what is in effect a guarantee of randomness to *calculate* areas. I'll describe the result in two dimensions, but it works generally. Assume we have a bounded set A whose area we wish to calculate. First construct a set R, typically a rectangle, with $R \supseteq A$ whose area is easy to calculate. In addition you need two things

- a random point $\mathbf{x}$ which is uniformly distributed in R; and
- a boolean function (membership) which is true precisely when $\mathbf{x} \in A$.

It is now easy to find the area of A; simply generate random points in R and count the proportion which land within A. The process is known as **Monte Carlo integration** because of its chance element. It can be shown that the error after n points have been chosen is proportional to $1/\sqrt{n}$, so taking four times as many points at random doubles the accuracy. This means it is slow, and until recently, was infeasibly so, but now it is the method of choice for complicated shapes for which the membership function is possible but which have a complicated shape.

10.8.1 The Volume of a Sphere

Lets apply this principle. I'll compute the volume of a sphere of radius 1 because it is a very simple exercise. The class is specialised to this end; I make no attempt to cope with spheres of different radius etc. So the setup is straightforward.

```
public class Sphere {
    /** The number of attepts */
    private final int attempts;
    /** The number of successes */
    private int hitCount;
    /** Simply set up the counters. **/
    public Sphere(int n) {
        attempts = n;
        hitCount = 0;
    }
    /** True if the chosen point lie inside the sphere. **/
    private boolean inside(double x, double y, double z) {
        return (x*x + y*y + z*z <=1);
    }
```

Note that the number of attempts *is* an instance variable so we can compute the volume from data stored within the class. Our sphere of radius 1 necessarily lies in a cube of side 2, so the outer volume is 8 units; thus it is necessary to generate points randomly within this volume. Here is the rest of the class.

```
public double estimate() {
    double x;
    double y;
    double z;
    double totalVolume = 2.0*2.0*2.0;
    for (int i = 0;i < attempts;i++) {
        x = 2*Math.random()-1;
        y = 2*Math.random()-1;
        z = 2*Math.random()-1;
        if (inside(x,y,z)) {
            hitCount++;
        }
    }
    return (totalVolume*hitCount/attempts);
}

/** A simple test routine. **/
public static void main(String[] argv) {
    int total = 100000;
    Sphere s = new Sphere(total);
    System.out.println("Approximation is: " + s.estimate());
```

```
        System.out.println("True volume is: " + 4*Math.PI/3);
    }
```

10.9 Simulation

In this section we discuss in detail a simulation in which you are interested in the answer to a concrete problem.

10.9.1 A Meeting

10.2. Example. A boy and a girl agree to meet at the Union sometime between 6:00 pm and 7:00 pm one evening. Neither is sure that the other will turn up, and so each decides only to wait for a fixed time before leaving again. The boy decides to wait for 15 minutes, while the girl is only willing to wait for 10 minutes. They each come on public transport and observation suggests that their arrival time are likely to be *uniformly* distributed between 6:00pm and 6:59pm.

Given that they both turn up, what is the probability that they meet?

As discussed before, we always have the option of doing such a question analytically; here we are instead going to simulate the events. We have two separate processes going on; one for each participant. After we know each arrival time, we can work out whether they meet or not; one person has to arrive both *after* the other arrived and *before* the other has left.

This is an ideal situation to use JAVA's need for objects. We create a class and construct two objects which are instances of the class; one to be the boy and one the girl. I'll call the class `Meeting`. Already we expect to have class variables `arrive` and `leave`. Here is the start of the actual class:

```
package uk.ac.abdn.maths.mx3015.random;
import java.util.Random;
public class Meeting {
    /** A shared random number generator. **/
    static Random r;
    /** The numbers of minutes after 6:00 when arriving. **/
    private int arrive;
    /** The numbers of minutes after 6:00 when leaving. **/
    private int leave;
    /** The number of minutes we are prepared to wait for. **/
    private int waitFor;
    /** The name of the person who is waiting. **/
    private String name;
```

There is no need to have a separate random number generator for each object. So the class variable `r` could have been a static variable. However we choose to have one each; the constructor is then straightforward.

```
Meeting(int w, String name) {
    if (r == null) {
            r = new Random();
        }
        this.name = name;
        waitFor = w;
    }
```

Next we need to set up useful methods. Two separate things are going on here; one is the actual arrival; the other is the check as to whether a meeting occurred. Given our flexible random number generator, the first is easy:

```
    /** A new occasion; resets arival and departure times. **/
    void arrives() {
        arrive = r.nextInt(60);
        leave = arrive + waitFor;
        if (leave > 59) {
            leave = 59;
        }
    }
```

while the second, checking whether they meet, means we need another person (another object of class `Meeting`) to exist as well.

```
    /** Returns true if they meet; false otherwise. **/
    boolean meets(Meeting m) {
        return ((arrive <=m.arrive) && (leave >= m.arrive)
                || (m.arrive <= arrive) && (m.leave >= arrive));
    }
```

So far this has been a typical JAVA example. A number of small methods, each of which is easy to verify. Now comes the testing. It is convenient to write a `toString` method; then the `main` method seems to make a great deal of sense?

```
    /** Just here to simplify the main method. **/
    public String toString() {
        return (name + " arrived at 6." + arrive +
                " and left at 6." + leave + ".\n");
    }

    /** A simple test routine. **/
    public static void main(String argv[]) {
        Meeting girl = new Meeting(10,"Alice");
        Meeting boy  = new Meeting(15,"Bob  ");
        boy.arrives();
        girl.arrives();
        String may = (boy.meets(girl))?"":"don't ";
        System.out.println(girl + boy.toString() +
                           "They " + may + "meet.");
        int meetings = 0;
        final int trials = 100000;
```

```
            for (int i=0;i<trials;i++) {
                boy.arrives();
                girl.arrives();
                if (boy.meets(girl)) {
                    meetings++;
                }
            }
            System.out.println("On average they meet " + 100*meetings/trials
                                   + "% of the time.");
        }
```

It turns out that under the conditions given, they meet about 38% of the time. Try it for yourself, and decide if it is easier than doing the analysis.

10.9.2 A Queue

Our second example looks at the behaviour of a queue, and is drawn from practical work set to Statistics students. I'm not trying to describe the interesting statistics here; rather to pick a relevant example which generates a larger collection of objects than we have seen so far.

The notion of queueing is familiar to us all. In this description, I'm going to give names to all the things involved in the process. A **queue** consists of a **server**, who performs some service for each **client** in turn. The service takes a certain time to perform; this time is governed by a **distribution**, which describes how this service time varies. The clients arrive for service at times which are also governed by some distribution . If the server is occupied with a client, then all subsequent clients have to wait, and present themselves in turn for service. The way in which this is usually implemented, in which clients stand one behind each other, and in front of the server, gives rise to the "usual" notion of a queue; from our viewpoint it only describes part of the process.

This description is the first step towards identifying objects for our simulation. It has been suggested that one way to "automate" the process of finding objects is to take such a description and underline all the nouns; these are then the objects! While this is too simple, the words in bold font in the previous paragraph *will* form a useful set of objects.

Lets start with a distribution. The only point of a distribution is to describe the time interval between consecutive events. The problem is one of simulating a single server queue. A service is provided which takes a certain time to complete. Requests are queued until they can be fulfilled, and as additional requests occur they are added to the queue. The aim of the simulation is to look at the length of queue that forms.

The time to perform each service is variable. We simulate it by assigning each service time "at random". In the same way, the time between new requests for service is to be "random". We don't necessarily assume any connection between the service time and the gap between arrivals.

We concentrate initially on the service time; what does it mean to say it is "random"? Clearly a service time has to be positive. One obvious possibility then is that the service time should be chosen randomly from (say) $[0, 1)$. In other words, that it should be uniformly distributed in the interval $[0, 1)$. Many other interpretations are possible. For example there may be be just two types of service, which take time 0.3 and 0.7 and which are equally likely to occur. This gives a different distribution of service times which we could equally easily simulate.

The gap between arrival times can be treated in the same way. In one situation these gaps may be uniformly distributed; in another arrivals might cluster, with (say) a 30% chance that they come in pairs. We need the concept of a **distribution** of arrival times from which samples are chosen at random. The particular distribution is then chosen to suit the problem being simulated.

Here is a class which implements a **deterministic distribution**; ie one in which each sample from the distribution is the same.

```
public class DeterministicDistribution extends Distribution {

    /** Store the fixed interval between events. **/
    private double interval;

    /**
     * Creates a new <code>DeterministicDistribution</code> instance.
     *
     * @param interval the time between events.
     */
    DeterministicDistribution(double interval) {
        this.interval = interval;
    }

    /**
     * This is the method we need to extend a Distribution.
     *
     * @return the next sample from the distribution.
     */
    double sample() {
        return interval;
    }
}
```

This code is sufficiently short that there are few problems. But already we are looking ahead; a `Distribution` itself is abstract and very simple:

```
public abstract class Distribution {
    /**
     * The only method samples from the distribution.
     * @return the next sample.
     */
    abstract double sample();
}
```

We can now go to work knowing we have a distribution available to test with. The advantage of *this* one is that there is no random element; we know what to expect. This is a great advantage during testing!

Our next object is equally simple. All a Client needs to know about is the time when it arrived, or started to request service. So the pattern is almost the same, with an 1-argument constructor and an accessor.[3]

```
/** A client waits in a queue to receive service. */
public class SClient {
    /** When the client requests service. */
    double arrivalTime;
    /** Constructor requires an arrival time. */
    SClient(double t) {
        arrivalTime = t;
    }
    /** Standard accessor. */
    double getArrivalTime(){
        return arrivalTime;
    }
}
```

Now lets move to the Queue itself. A queue is controlled by a distribution of arrival times, and is responsible for making clients available. We add two extras: the queue shuts after a given time, and is able to say how many clients remain to be processed.

```
public class SQueue {
    /** The distribution that governs service times. */
    Distribution arrivalTime;
    /** Maintain our version of the clock. */
    double timeNow = 0.0;
    /** Only serve until this time. */
    final double queueShut = 50.0;
    /** Queue governed by distribtion of arrival times. **/
    SQueue(Distribution d) {
        this.arrivalTime = d;
    }
    /** This is the crucial external interface. */
    SClient getNext() {
        timeNow += arrivalTime.sample();
        if (timeNow >queueShut ) {
            return null;
        }
        return new SClient(timeNow);
    }
    /** How many clients are there in the queue now? */
    int waiting() {
        int waiting = 0;
```

[3]A Client may be interested in more information, such as when service became available, and was completed. I'm describing a simple SClient version here; a more general version, namely Client with these refinements is available with the code.

```
                while (getNext() != null) {
                    waiting++;
                }
                return waiting;
            }
        }
```

Notice how little interaction there has been between our objects so far. Each object needs the others to exist, but is able to let them behave as they want to. It perhaps isn't obvious that the SQueue should maintain its own notion of time. However it is the distribution of arrival times which drive the whole process. Note also that getNext() deliberately returns a null when the queue is shut. This provides a way of communicating this fact to the server, who would otherwise need to ask the queue if it was empty before requesting the next client.

We are almost there now; the remaining crucial class is the Server which is really at the heart of the process. The server is also governed by a distribution, this time of service times rather than arrival times, and so the class looks quite like the Queue class, even to having its own notion of time.

The main job of a server is to serve; this is done in the serve() method, whose main job is to work out whether it waits for the next client or vice-versa. Here is the class:

```
        /** A server serves Client objects taken from the queue */
        public class SServer {
            /** This governs the service times for clients. */
            Distribution serviceTime;
            /** Maintain our version of the clock. */
            double timeNow = 0.0;
            /** The time when the server shuts. **/
            final double closingTime = 50.0;
            /** QueueServer governed by distribtion of service times. **/
            SServer(Distribution d) {
                serviceTime = d;
            }
            /** Say if there is more action. **/
            boolean isOpen() {
                return (timeNow < closingTime);
            }
            /** The serve method is where it all happens. **/
            void serve(SQueue queue){
                SClient client = queue.getNext();
                if (client == null) { // No more clients to be served.
                    timeNow = closingTime;
                } else { // Try to serve the next client.
                    if (client.getArrivalTime() > timeNow) {
                         // we were waiting for a client
                        timeNow = client.getArrivalTime();
                    }    // Otherwise the client was queueing.
                timeNow +=serviceTime.sample();
            }
        }
    }
```

At this stage everything can be tested. All the basic components are there and they work together as they will in a final setup. So emphasis moves from the underlying structure, to the reason for doing the simulation in the first place. Our basic distribution is simple, but the problem only becomes interesting when the distributions become more complicated. We can create more complicated distributions with *no* change to what we have already. As long as our new distributions extend the `Distribution` class, we can use them. Here is an example of one such extension.

```
public class ExponentialDistribution extends Distribution {
    /**
     * The fixed mean of the Distribution
     */
    private double mean;

    /**
     * The random generator of doubles used to determine the next
     * sample.
     */
    private Random r;

    /**
     * Creates a new <code>ExponentialDistribution</code> instance.
     *
     * @param m the mean of the distribution.
     */
    ExponentialDistribution(double m) {
        mean = m;
        r = new Random();
    }

    /**
     * Create random samples from an exponential distribution with
     * given mean.
     *
     * @return the next sample from the distribution.
     */
    double sample() {
        return -Math.log(r.nextDouble()) * mean;
    }
}
```

In practice there can be many such different extensions; you will see another one used in the final test of these classes.

I'm going to show one more idea. The aim of all this simulation is to see what happens "on average". The queue will be allowed to work itself out many different times, and the number of clients waiting for service at the end of the time will be noted. This can easily be done in a main() method, but there is an advantage in separating out this recording code from the code needed to create the test situation. Here then is an Observer class designed *just* to gather statistics. You will see its main job is just to note() what happens, as you expect of an Observer!

```
public class Observer {
    /** The number of simulations that have been run. **/
    private int simulationCount;
    /**
     * The number of simulations in which no Clients were waiting when
     * the server closes.
     **/
    private int zeroCount;
    /** Store the number of clients waiting. **/
    private int sum;
    /** And accumulate more statistical data. **/
    private long sumSq;
    /** Explicitly zero everything; this would happen anyway. **/
    Observer() {
        simulationCount = 0;
        zeroCount = 0;
        sum = 0;
        sumSq = 0;
    }
    /**
     * Interest is in the queue size at the end of each run. *
     * @param qSize the final queue size to be noted.
     */
    void note(int qSize) {
        simulationCount++;
        if (qSize == 0) {
            zeroCount++;
        }
        sum += qSize;
        sumSq += (qSize * qSize);
    }
    /**
     * Convenience method to help reporting.
     * @return the percentage of the time the queue is empty at the
     * end of the simulation.
     */
    double getZeroPerCent() {
            return zeroCount * 100.0 / simulationCount;
        }
        /**
         * Report the mean number waiting at the end of a run. *
         * @return the mean.
         */
        double getMean() {
            return sum / (double) simulationCount;
        }
        /**
         * Report the standard deviation of the remainder.
         * @return the standard deviation.
         */
        double getSD() {
            double variance = (sumSq - (sum * sum / (double) simulationCount))
                / (simulationCount - 1.0);
            return (Math.sqrt(variance));
        }
    }
```

Finally we can put all this together. We simply let the queue run a large number of times, note the result and then print out the answers.

```
    public class SQueueTest {
        /** A simple test routine. **/
        public static void main(String argv[]) {
            Observer observer = new Observer();
            for (int i = 0;i < 10000;i++) {
                SQueue queue   = new SQueue (new
                    ExponentialDistribution(2.0));
                SServer server = new SServer(new
                    GaussianDistribution(1.5,0.23));
                while (server.isOpen()) {
                    server.serve(queue);
                }
                observer.note(queue.waiting());
            }
            System.out.print("The server shut with an average of ");
            System.out.println(observer.getMean() + " clients waiting.");
            System.out.print("The standard deviation of these data was ");
            System.out.println(observer.getSD() + ".");
            System.out.print("The queue was cleared ");
            System.out.println(observer.getZeroPerCent() +"% of the time.");
        }
    }
```

The interest in this example is the way that the JAVA classes model the actual situation, This leads to a great deal of independence between the classes. In contrast a naive implementation could easily have a single class containing everything.

Chapter 11

Cryptography: Java for Secure Communication

If you have ever used the web to buy anything, or even given confidential information to a browser you are probably aware that it claims to do *secure* communication with the server. In other words, the data passed to the server is coded, or *encrypted* in such a way that an eavesdropper or snooper; someone who is able to monitor the data passing over the communication line (in our case, an Ethernet) is not able to read what is being sent. In contrast, normal web pages are sent in **plaintext**; in other words, the content can be read by anyone.

The underlying technology uses some interesting mathematics from a branch which was regarded as very pure until these applications were discovered, namely number theory. I'm using "pure mathematics" here as a contrast to "applied mathematics"; traditionally pure mathematics was investigated simply from intellectual curiosity. It is one of the surprises of modern computing that it has been able to make serious use of theorems in number theory developed originally solely "because they were there".

In this chapter we describe the background to **cryptography**, fill in some needed background in number theory, show how it is relevant, and finally discuss how this may all be implemented in JAVA. For a more details of the number theory described here, look at Burton (1995); I've tried to keep my notation in line with this account. For a more advanced treatment, with details of primality testing, a discussion of the timings of the various algorithms involved, and a more abstract approach to the underlying algebra, take a look at Motwani & Raghavan (1995, Chapter 14).

11.1 Introduction

Cryptography can be thought of as the ability to write a message in such a way that it remains confidential; ensuring that only the intended recipients are able to read it. This is essential when the Internet is used for business applications. However cryptography can also be used for:

Authentication: to ensure that the receiver of a message can verify it's origin;

Integrity: to ensure the message has not been tampered with; and

Non-repudiation: to ensure that the originator cannot later deny that she sent the message.

Cryptography is a large subject and it has a long, fascinating and colourful history. We concentrate here on a single application of cryptography, that of **digital signatures**.

11.1.1 Ciphers and Keys

The word “cryptography” is derived from the Greek as “hidden writing”. In cryptography, the word **cipher** is used to mean the cryptographic algorithm or function function which **encrypts** or **decrypts** a message. The message before encryption is known as **plaintext** and the result of encrypting the plaintext by applying the cipher is the **ciphertext**. Often there are two ciphers; one for encryption and one for decryption. In modern cryptography these functions typically take 2 arguments; the plaintext or ciphertext and a **key**. The security of these cryptosystems lies entirely in the security of the key. Anyone who knows the decryption key can read the ciphertext.

As an example, consider the 27 symbols consisting of the letters in alphabetical order, followed by a space. They are to be arranged cyclically, so that the next letter after “A” is “B”, while the next letter after “Z” is “space”, the next letter after “space” is “A” and so on. Using this arrangement, we obtain the well known **Caesar cypher** (we implemented one in Section 6.3) in which each letter in the message is replaced by the one which occurs k places further on in this list. Here k is the key. Thus a message which is generated using the key $k = 3$ and whose ciphertext is `KHOOZCZRUOG` can be decrypted knowing this key; the decryption algorithm is the same as the encryption algorithm, using the key $k = -3$.

In some systems, known as **symmetric cryptosystems** the encryption and decryption keys are the same. With these systems, the sender and recipients must agree on a key before the message is sent, and then both use it. This means that the key must be communicated somehow beforehand and therefore stands a chance of being intercepted and hence the security of the message compromised.

In **Public Key** cryptosystems, the encryption and decryption keys are different *and* the decryption key cannot reasonably be derived from the encryption key. This allows the encryption key to be widely distributed, that is, made public; the encryption key is known as the **public key**, while the decryption key is known as the **private key**.

If now Alice wishes to send a confidential document to Bob[1], she obtains Bob’s public key and uses it to encrypt the document to be sent. Bob then decrypts the ciphertext with his private key. It is useful to avoid the terms ’encryption key’ and ’decryption key’ when discussing public key cryptosystems because in some algorithms including the popular RSA cipher, which we will be looking at later, both the private key and the public key may be used for encryption. We thus say “encrypt the message with the private-key” rather than “encrypt the message with the decryption key” which appears to be confusing.

11.1.2 Digital Signatures

Handwritten signatures are immensely important for document interchange.

- The presence of a signature on a document convinces the recipient that the signatory signed the document.

[1]The use of the names Alice, Bob and even Eve (an eavesdropper) is traditional in cryptography.

- The signature is proof that the signatory and no-one else signed the document.
- The signature is not reusable.
- A signed document is unalterable. That is, any evidence of tampering with the document renders the document invalid.
- The signatory cannot later claim that he did not sign the document. ie. the signature cannot be repudiated.

Of course, handwritten signatures are not entirely secure. However the risks of compromise are relatively small. This, together with the ease by which documents can be signed continues to make the hand-written signature the standard (western) way in which paper documents are authenticated.

It is much harder to 'sign' electronic documents or computer files. You all know how easy it is to copy files and change the content of a file. It is also generally *very* easy to change the time-stamp on computer files to make it appear that a file existed before the date that it was really created. If business is to be conducted using computer files, email etc it is essential that some digital equivalent of a handwritten signature can be applied to e-documents.

11.1.3 Implementing Digital Signatures

It is in fact fairly easy to implement digital signatures using a public key cryptosystem. To sign a document, Alice simply encrypts it with her private key. Anyone can decrypt the document, using Alice's *public* key. And Alice's public key will not decrypt documents originating from Eve, even if she claims they come from Alice. Thus a document from Alice is verified as originating with her.

Alice has no need to encrypt the whole of her document to prove it originated with her. Instead she encrypts what is known as a **hash** of the document. This hash or **message digest** is a large number (but very much smaller than the message itself) that is derived from the whole message. The **hash function** which produces the hash, has the property that any change in the message (however small) produces a change in the hash, while the chance of different documents producing the same hash is vanishingly small. In effect then the hash is unique to a plaintext. So Alice computes the hash of her document and sends it with the document as a 'signature'. When Bob receives the document he first decrypts the hash using Alice's public key. His next step is to generate his own version of the hash using the same hash function. If this turns out to be the same as the one he decrypted, he can be sure that

- the document originated with Alice; and
- the version he received is the one that Alice signed.

In other words the use of such a hash allows the same type of authentication and non repudiation that a paper signature does.

11.2 Mathematical background

Before going further, we look at the process of encryption from a more mathematical viewpoint. As usual, let $\mathbb{Z}_n$ consist of the integers $\{0, 1, \ldots, n-1\}$. We can use **modular arithmetic** to get the operations of addition and multiplication defined on this set. Thus for $a, b \in \mathbb{Z}_n$, we define $a+b$, sometimes written as $a +_n b$ to be the non-negative remainder when $a+b$ is divided by n. The crucial point is that $a +_n b$ then also lies in $\mathbb{Z}_n$

This is a familiar operation in JAVA; after all, if $m = ln + r$, where $0 \le r < n$, then l is just `m/n`, the value when one `int` is divided by the other, while the remainder r is what we have learned to write as `m%n`. Thus given that a and b are `int`s, $a +_n b$ is simply `(a+b)%n`. In the same way we can define ab, or $a \times_n b$, using modular arithmetic to reduce the answer to lie in $\mathbb{Z}_n$.

The number n will be fixed in all of this; I will eventually call it the encoding or **enciphering modulus**; for now just call it the underlying **modulus**.

It is convenient to have a notation which allows us to pass from $\mathbb{Z}$ to $\mathbb{Z}_n$. I will write $m \equiv r \pmod{n}$ if $0 \le r < n$ and there is some l such that $m = ln + r$. This is sometimes described by saying that r is the **residue** of m mod n; the notation is even abused by writing $m \pmod{n}$ when we really intend to write r.

Modular arithmetic not only gives answers in $\mathbb{Z}_n$, but it behaves sensibly.

11.1. Proposition. *Let $a \equiv a' \pmod{n}$ and $b \equiv b' \pmod{n}$. Then*

$$\begin{aligned} a +_n b &\equiv a' +_n b' \pmod{n} \\ a \times_n b &\equiv a' \times_n b' \pmod{n} \end{aligned}$$

In other words, it doesn't matter whether we remove multiple of the modulus before or after doing modular arithmetic.

Proof. There are numbers k, l such that $a = kn + a'$, $b = ln + b'$. Now just calculate. □

11.2.1 Some Group Theory

A knowledge of group theory is not a prerequisite for this course, and nor should it be. However a brief acquaintance is helpful in what follows. All I actually need is the concept and one elementary counting result.

A (finite) group G is a set G together with a binary operation $\circ : G \times G \to G$ which satisfies the following:

Associativity: if g, h and $k \in G$, then $g \circ (h \circ k) = (g \circ h) \circ k$. [2]

Identity: there is a distinguished element $e \in G$ such that $e \circ g = g \circ e = g$ for every $g \in G$.

Inverse: Given $g \in G$ there is an element $g^{-1} \in G$ such that $g \circ g^{-1} = g^{-1} \circ g = e$.

This formality doesn't say why these rules have been singled out. The motivation is to describe a structure consisting of a set with a single well-behaved rule for combining two elements. The operation is written mysteriously as $\circ$ because real examples cover both

[2]We don't require that $g \circ h) = h \circ g$, although all the examples we meet later satisfy the additional **abelian** condition.

addition, where $\circ$ is actually $+$, and multiplication, where $\circ$ is written as $\times$ or even omitted, simply writing the symbols next to each other. The "distinguished element" will usually be written as 0 (if $\circ$ is addition) or 1 (if $\circ$ is multiplication). The last rule then says that every element has a negative (if $\circ$ is addition) or an inverse (if $\circ$ is multiplication). In what follows, we meet an example of each.

In the rest of this section I'll use multiplicative notation and write composition as juxtaposition. In particular we write $g \circ g = g.g = g^2$ and so on. If G is a finite group, the **order** of G is simply the number of elements in G.

Let $g \in G$ and consider the sequence of elements $g, g^2, g^3, \ldots$. Since G is finite, this must eventually repeat, so there are some indices r and s with $r < s$ such that $g^r = g^s$. Then since g has inverse g^{-1}, we have $e = g^{s-r}$. The **order** of $g \in G$ is the smallest k such that $g^k = e$ and our remark above shows that every element *has* an order.

11.2. Proposition. *Let G be a finite group, and let $g \in G$. Then the order of g divides the order of G.*

Proof. I'm not going to prove this in detail, but the idea is very simply. Consider the set $\{e, g, g^2, \ldots, g^{k-1}\}$, where k is the order of g. These are all distinct. If this fills up the whole of G, the order of G must be k and we are finished. Otherwise there is some $h \in G$ which hasn't yet been counted. The crucial technical step is to show that $\{h, gh, g^2h, \ldots, g^{k-1}h\}$ are all different, and that *none* of them has been counted so far.

If this gives the whole of G, then G has order $2k$ and we are done; otherwise, in the same way, we can find another disjoint collection of k elements and so on. In any case, the order of G is always a multiple of k, as required. □

11.2.2 The Group $\mathbb{Z}_n$

The set $\mathbb{Z}_n$, together with the operation $+_n$ forms such a group, known as the additive group of integers $\mod n$. From now on I write $+$ rather than $+_n$. The extra content in saying this is a group is that:

- addition is well behaved in that it is associative;
- there is an "identity" element, namely 0 with the property that $0 + a = a$ for all $a \in \mathbb{Z}_n$; and
- each $a \in \mathbb{Z}_n$ has an additive inverse, ie an element $-a$, such that $a + (-a) = 0$.

All these properties are very obvious from the corresponding property in $\mathbb{Z}$. Of course, in its usual meaning $-a \notin \mathbb{Z}_n$ unless $a = 0$. However if $a \in \mathbb{Z}_n$ and $a \neq 0$, then $n - a \in \mathbb{Z}_n$ and acts as an additive inverse. Equivalently, $-a \pmod n$ is the required element.

Define a map $E_k : \mathbb{Z}_n \to \mathbb{Z}_n$ by "addition mod n" so that $E_k(m) = m + k$. Note that we can always "undo" the effect of this map; indeed the inverse, or "undoing" map is E_{-k} since $(m + k) + (-k) = m$. Formally this is the statement that E_k is *bijective*. We now recognise the Caesar cipher example on page 128 as using such a map, working in $\mathbb{Z}_{27}$. To deal with the full message, we first write it as a sequence M_i of elements in $\mathbb{Z}_n$ for $i = 1 \ldots l$. The ciphertext is then the sequence $E_k(M_i)$ $i = 1 \ldots l$; indeed I've chose the notation E for the map to remind you that this is the *enciphering* map. We can describe this cipher as depending on the **key** k; deciphering is exactly the same as enciphering, except that the key $-k \in \mathbb{Z}_n$ is used.

Of course the Caesar cipher is not very secure; deciphering is trivial if you know the key. Even if you don't, it is not too hard to try all possible keys. However the behaviour is typical; indeed we will see that there is a natural group which is a subset of $\mathbb{Z}_n$, and a fairly obvious bijection on this group which provides us with an "industrial strength" cipher. To develop this, we need to look at some number theory.

11.2.3 Greatest Common Divisors

Much number theory is concerned with expressing a number in terms of smaller factors. If a, $b \in \mathbb{Z}$ we say that a **divides** b, or a is a **divisor** or **factor** of b and write $a|b$ if there is some $c \in \mathbb{Z}$ such that $b = ac$. Of special interest are numbers with no non-trivial factors. Recall that an integer $p > 1$ is **prime** if its only positive divisors are 1 and p. A positive integer that is not prime is **composite**. A fundamental result is that any integer can be represented uniquely (up to the order of the factors) as a product of powers of primes.

We say that integers a and b have a **common divisor** c if c is a divisor if both a and b. Given a, $b \in \mathbb{Z}$ we say that c is the **greatest common divisor** or **highest common factor** of a and b, and write $c = \gcd(a, b)$ if

- c is a common divisor of a and b; and
- whenever d is a common divisor of a and b then $d|c$.

We have already met an algorithm to compute the greatest common divisor of two integers in Section 2.5.3, namely the **Euclidean algorithm**. In fact we need a more detailed version as follows.

11.3. Theorem (Extended Euclidean Algorithm). *For $m > n > 0$ there are integers c and d such that*

$$\gcd(m, n) = cm + dn.$$

Proof. The proof is constructive. We write $r_0 = m$, $r_1 = n$, and in general, write $r_{i+2} = r_i \% r_{i+1}$ and $q_{i+2} = r_i / r_{i+1}$, where this is integer division. The process stops when some $r_i = 0$; putting everything back together gives values for c and d. □

Lest this seem too glib, here is JAVA code to find the coefficients, giving a more accurate way to describe the proof.

```
/**
 * Returns the coefficent "c" in the expression g = c*m + d*n.  If
 * g = c0*m + d0*n = c1*n + d1*m%n, we can obtain new expressions
 * for c0 and d0, using the fact that m = n*(m/n) + m%n.
 * Re-arranging to eliminate m%n, we see that g = d1*m + (c1
 * -d1*(m%n))*n; thus c0=d1 and d0 = c1 -d1*(m/n).  This is the basis
 * of the recursive formulae in both getC and getD below.
 **/
public static long getC(long m, long n) {
        if (m == 0) return 0;
        if (n == 0) return 1;
        return getD(n,m%n);
    }
    /** Returns the coefficent "d" in the expression  g = c*m + d*n. **/
    public static long getD(long m, long n) {
        if (m == 0) return 1;
        if (n == 0) return 0;
        return getC(n,m%n) -getD(n,m%n)*(m/n);
    }
```

Given a, $b \in \mathbb{Z}$ with $\gcd(a,b) = 1$, we say that a and b are **coprime**, or that a is **coprime to** b.

11.4. Proposition. *Let $a \in \mathbb{Z}_n$ be coprime to n. Then a has a multiplicative inverse in $\mathbb{Z}_n$. In other words there is some $b \in \mathbb{Z}_n$ such that $a.b = 1$ in $\mathbb{Z}_n$.*

Proof. By our assumption, $\gcd(a,n) = 1$, so by the extended Euclidean algorithm, there are integers c, d such that $1 = ac + nd$. Thus

$$ac = 1 - nd \equiv 1 \pmod{n}.$$

Using Proposition 11.1, we see that a has multiplicative inverse $b = c \pmod{n}$. □

11.2.4 The Group $\mathbb{Z}_n^*$

The multiplicative equivalent of the group $\mathbb{Z}_n$ is a little less obvious. Let

$$\mathbb{Z}_n^* = \{l \in \mathbb{Z}_n : \gcd(l,n) = 1\}$$

11.5. Proposition. *The set $\mathbb{Z}_n^*$ with the usual multiplication $\times_n$ is a group.*

Proof. We start by showing that if $a, b \in \mathbb{Z}_n^*$, then so is ab. If this is not so, and $\gcd(ab,n) > 1$, there is some prime p such that $p|ab$ and $p|n$. Then either $p|a$ and so $\gcd(a,n) \geq p$ or $p|b$ and again we get a contradiction. Since multiplication is associative in $\mathbb{Z}$, it remains so in $\mathbb{Z}_n^*$. The identity element is of course 1, and it remains to show that given $a \in \mathbb{Z}_n^*$ the multiplicative inverse b obtained in Proposition 11.4 also lies in $\mathbb{Z}_n^*$. We have $ab = 1 + ln$ for some l. If $r|b$ and $r|n$, then we see that $r|1$. Thus $r = 1$ and $\gcd(b,n) = 1$. □

Let $\phi(n)$ be the number of elements in $\mathbb{Z}_n^*$. This function is know as the **Euler phi-function**; we need some of its properties.

In one case it is easy to calculate: if p is prime, then $\mathbb{Z}_p^*$ simply consists of the numbers $1, 2, \ldots, p-1$, since each of them is coprime to p. Thus $\phi(p) = p - 1$. Note also that $\phi(p) = p - 1$ is *only* true if p is prime.

This observation can be combined with our work on group theory (Proposition 11.2 to obtain an old result of Fermat:

11.6. Theorem (Fermat's Little Theorem). *Let p be prime and let $a \in \mathbb{Z}_p^*$. Then $a^{p-1} \equiv 1 \pmod{p}$.*

Proof. Since $a \in \mathbb{Z}_p^*$, the order of a divides the order of $\mathbb{Z}_p^*$. Thus in $\mathbb{Z}_p^*$, necessarily $a^{p-1} = 1$, which is the required conclusion. $\square$

11.7. Example. We can put these ideas together to obtain a bijective map $\mathbb{Z}_p^* \to \mathbb{Z}_p^*$ which is rather like the one we used to generate the Caesar cipher. I will use that language to describe our new map. Let M be a message in $\mathbb{Z}_p^*$, let k be a key chosen so that $\gcd(k, p-1) = 1$, and define $E_k(M) = M^k \pmod p$. The restriction on the key that it has no common factor with $p - 1$ is not great; certainly we can satisfy it by choosing a *prime* key.

Using this assumption and the Euclidean algorithm, there are numbers j and l such that $kj + l(p-1) = 1$. Then, doing arithmetic in $\mathbb{Z}_p^*$, we have

$$E_j(E_k(M)) \equiv (M^k)^j \equiv M^{kj} \equiv M^{1-(p-1)l} \equiv M$$

since by Fermat's theorem $M^{(p-1)l} = 1^l = 1$. Thus E_k is an invertible map, with inverse E_j. In other words, the **deciphering** map is of the same form as the enciphering map.

Of course this cipher is of no more practical use than the Caesar cipher, in that it would be relatively easy to deduce the inverse map. However we are surprisingly close to one that is *very* hard to break.

11.3 The RSA Algorithm

We have almost all the material assembled to put together what appears to be a very secure public-key cryptosystem. This was first described publicly by Rivest, Shamir and Adelman in 1977[3] and so is known as the RSA cryptosystem. The company they founded (`http://www.rsasecurity.com`) is now one of the dominant ones in computer security and their algorithm is probably the most widely used public key algorithm.

Instead of working in $\mathbb{Z}_p^*$ as above we work more generally in $\mathbb{Z}_n^*$ where n is no longer prime. In fact we proved more than Theorem 11.6; the same argument gives more, although the result has to be stated in terms of Euler's phi-function:

11.8. Theorem (Euler). *For any n and $a \in \mathbb{Z}_n^*$, $a^{\phi(n)} = 1 \pmod n$.*

Another result of Euler's is that his ϕ is multiplicative.

11.9. Proposition. *If $\gcd(a, b) = 1$ then $\phi(ab) = \phi(a)\phi(b)$.*

Proof. Not easy. You can find details in Burton (1995, Theorem 7.2). $\square$

In particular, if p and q are distinct primes and $n = pq$ then $\phi(n) = (p-1)(q-1)$. We use this n as an enciphering modulus to produce an encoding function $E = E_k$ in almost exactly the same way as for Example 11.7. Given a message M, which we assume initially satisfies $\gcd(M, n) = 1$ so that $M \in \mathbb{Z}_n^*$, we choose a key k such that $\gcd(k, \phi(n)) = \gcd(k, (p-1)(q-1)) = 1$. Again this is not hard; one choice would be any prime between (say) p and $\phi(n)$. Finally we define our enciphering function by

$$E_k(M) = M^k \in \mathbb{Z}_n^*.$$

[3]There is evidence that the same system was in use by GCHQ from the late '60s.

The extended Euclidean algorithm gives integers j and l such that $kj + l\phi(n) = 1$. Then arguing as before, we have

$$E_j(E_k(M)) \equiv M^{jk} \equiv M^{1-\phi(n)l} \equiv M \pmod{\phi(n)}$$

on using Euler's theorem, rather than Fermat's theorem. Our restriction that $\gcd(M, n) = 1$ is purely technical; the fact that E_j acts as a deciphering key for E_k remains true in any case.

The difference between this and the previous version is fundamental. It is perfectly possible to publish both enciphering modulus n, the key k and even the *method* being used without compromising the cipher. In other words, this is a public-key cryptosystem. To compute the decoding modulus, it is necessary to apply the Euclidean algorithm to k and $\phi(n)$. Knowing n itself essentially gives no information about $\phi(n)$ unless n can be expressed as a product of the two primes from which it was built. It is not hard to show that knowing n and $\phi(n)$ make it very easy to derive p and q. In other words, the computation of $\phi(n)$ when $n = pq$ is essentially equivalent to expressing n as a product of its prime factors. This is believed to be a *very* hard problem if n is reasonably large.

11.4 Prime Numbers

We said very little so far about how to *implement* some of these techniques we discussed, concentrating instead on showing that they could be useful. We now take a closer look at implementation. Our crucial need is to be able to obtain a large supply of primes numbers. Specifically we need to find large primes p and q so we can use $n = pq$ as our enciphering modulus. There is always a balance in cryptography between convenience and security. The larger the prime, the harder it is to factorise the enciphering modulus, but the longer manipulations such as coding and decoding take. But an example may be interesting. In 1977 a message was coded using a 129 digit enciphering modulus (or $129\log_2(10) \sim 430$ bits), the product of two similar size primes, and the scientific community was challenged to decrypt the message.[4] At the time, using the technology available then, it was estimated that it would take 40 quadrillion years to break using brute force techniques. In fact it was decrypted 17 years later in 1994, although the effort; 600 volunteers, 1600 computers and 8 months, was still very large. Nevertheless the increase in machine speed and availability of both machines and networking had not been predicted.

This discussion shows why currently recommendations for a "safe" size of prime to use in encryption is for perhaps 512, 1024 or 2048 bits. In what follows I will work with 512-bit primes. In any case, n is be *very* much larger than anything we can manipulate using a `long` and our first problem will be manipulating such numbers. Fortunately JAVA has a `BigInteger` class designed for this situation; and objects of this class are essentially unlimited in length.

11.4.1 Availability

If primes are rare, we may not have a useful scheme. Factorisation of n could be achieved simply by trying all the primes up to $\sqrt{n}$. In fact this isn't the case; there are far, far too

[4]The message was "The magic words are squeamish ossifrage"; an ossifrage is a kind of hawk!

many to make this possible. Let $\pi(x)$ be the number of primes $p \leq x$. Then the **prime number theorem** states that

$$\pi(x) \approx x \log x.$$

We can thus estimate how many 512 bit primes there are: the theorem gives

$$2(2^{513} - 1)/\ln(2^{513} - 1) - 2^{512}/\ln(2^{512}) \approx 3 \times 10^{152}.$$

where the factor of 2 is inserted because we don't look for large even primes! Clearly there are plenty of primes of a given size.

In fact, there are so many to choose from that we can take a prime at random from all those of 512 bits in length and be confident that no-one else will ever guess which one we chose. Even though everyone who creates a key-pair needs 3 unique prime numbers and even though each user of public-key cryptography may need key-pairs for different purposes, there remain so many primes that the chances of two such chosen at random being the same are essentially zero.

11.5 Calculating Prime Numbers

We now consider the question of how to find prime numbers. The prime number theorem shows that if we pick an integer n at random, it will be prime with probability $\approx \dfrac{1}{\ln(n)}$. So even if $n \sim 2^{512}$, our random n is prime with probability $\dfrac{1}{512\ln(2)} \sim 0.0028$. This is perfectly reasonable; in principle then we can find primes quickly.

So the important question is how easy is it to tell if a number n is prime. It is certainly infeasible to check the primality of n directly from the definition. We've seen an elementary method of finding primes, namely the prime sieve, in Section 4.5, and we will make use of it shortly, but this method would also take infeasibly long and use an infeasible amount of storage. However, using our sieve to generate a list of *small* primes, we can quickly decide that many such **prime candidates** n are composite. We simply check in turn whether each small prime is a factor of n. And any prime candidate which is accepted by such a filter is fairly likely to be prime.[5]

Note the result of such a test on a prime candidate n: either

- we demonstrate that n is composite; or
- it remains unknown whether or not n is prime.

We will call this a **one-sided** test for primality. We are about to meet more such tests. Recall Fermat's Little Theorem (11.6). If n is prime, and we choose any a with $1 < a < n-1$ then $a^{n-1} \equiv 1 \pmod{n}$. Conversely, if we ever find such an a for which $a^{n-1} \not\equiv 1 \pmod{n}$, we have demonstrated that n is composite.

We call a number n which passes this "Fermat's Little Theorem (FLT)" test for a given base a, but which is not actually a prime, a **pseudoprime** with respect to the base a. Such pseudoprimes are very rare; *very* much rarer than primes themselves. Of course we can

[5]It is said that using just the primes $p < 256$ eliminates 80% of all prime candidates.

apply the FLT test again using a different a; each such test passed successfully increase our confidence that n is prime. However there *are* numbers, called **Carmichael numbers** which are composite, but which satisfy the FLT test for *every* base a.[6] Indeed fairly recently it has been confirmed that there are an infinite number of Carmichael numbers.

11.5.1 The Rabin-Miller Test

There is a stronger version of the FLT test which has a useful bound describing when it fails, called the **Rabin-Miller** test (Rabin 1980). There is no loss in assuming that our prime candidate n is an odd number. Thus $n - 1 = 2^k m$ for some $k > 0$ and odd integer m. As with the test based on FLT, choose a base a with $1 < a < n - 1$ and calculate the following sequence modulo n:

$$a^m, a^{2m}, a^{4m}, \dots, a^{2^k m},$$

where each term is the square of the previous one, and in all there are $k + 1$ terms in the sequence. We say that n passes the Rabin-Miller test for the base a if *either* the first term in this sequence is 1, *or* 1 occurs later in this sequence and is immediately preceded by -1. If the test fails, we know that n is composite, and describe the base a as a **witness** to the compositeness of n. The crucial extra information we can derive is that if (odd) n *is* composite, then at least 3/4 of the numbers a with $1 < a < n - 1$ are witness to this fact.

We can convert this result into a **probabilistic primality test** as follows:

> Choose k independent integers $a_1, \dots a_k$ which $1 < a_r < n - 1$. Then if (odd) n passes the Rabin-Miller test for each base a_r, the probability that n is composite is no more than $1/4^k$.

In other words, we will never be *certain* that n is prime, but by performing enough Rabin-Miller tests, we make the probability of n being composite as small as we choose.

11.5.2 Real World Prime Generation

Here is the resulting procedure to generate primes — or strictly, numbers that have a very low probability of being composite:

- generate a random string of 512 bits;
- set both the high bit and the low bit to 1, thus ensuring that the corresponding number n is indeed 512-bits in length and is odd;
- check n for divisibility for all primes $p < 2000$; and
- choose 100 (say) independent integers $a_1, \dots a_{100}$ for which $1 < a_r < n - 1$ for each r and apply the Rabin-Miller test for each such base.

A number which passes all of these tests has a probability $< 4^{-100}$ of being composite; it is "probably prime". That all goes pretty quickly. The divisibility test is useful to weed out most of the composites before starting the more lengthy computations involved with the Rabin-Miller test.

[6]The smallest is 56; another is 1729.

11.6 BigInteger

We now have all the mathematical tools we need and can turn to implementation. I have already noted the `BigInteger` class, which is in the `java.math` package. In fact that provides all we need because it was written with cryptography in mind. To quote the first paragraph of the API documentation:

> Immutable arbitrary-precision integers. All operations behave as if BigIntegers were represented in two's-complement notation (like Java's primitive integer types). BigInteger provides analogues to all of Java's primitive integer operators, and all relevant methods from java.lang.Math. Additionally, BigInteger provides operations for modular arithmetic, GCD calculation, primality testing, prime generation, bit manipulation, and a few other miscellaneous operations.

In Section 3.1.5, we noted that JAVA does *not* permit operator overloading, so if m and n are both `BigInteger`s we *cannot* write $m + n$ for their sum; instead we must use the `BigInteger add(BigInteger)` method which returns a `new BigInteger` which is the sum of `this` BigInteger and the argument. Other arithmetic and operations and tests have to be treated similarly, using for example the `boolean equals(BigInteger)` method. A useful method to handle inequalities is the `int compareTo(BigInteger n)` which returns 1, 0 or -1 as `this` BigInteger is numerically less than, equal to, or greater than n.

I won't give a full discussion of the `PrimeGenerator` class, designed to generate large primes, although it is included in the latest distribution. However, I think it may be instructive to see parts of it. Specifically I'll show how to implement the Rabin-Miller test itself. Recall that in Section 11.5.1 we started with a prime candidate n, and needed to write $n = 2^k m$ where m is odd. I suggest such a decomposition is a good candidate for being done in a helper class; one which appears in the same file as the main (public) class, but which isn't declared to be public. Here is a possible implementation of the class, showing how to use various `BigInteger` methods.

```
/**
 * A utility class designed simply to hold the decomposition of
 * n-1 as the product of 2^k and m, where m is odd.  Here n is a
 * candidate for being prime (so is odd!).
 **/
class Decomposition {
    /** The number we decompose. **/
    private BigInteger nMinusOne;
    /** The index of the largest power of two dividing nMinusOne. **/
    private int k;
    /** The remainder nMinusOne/(2^k) **/
    private BigInteger m;
    /**
     * The constructor actually performs the decomposition of its
     * argument, obtaining numbers k and m such that
     * nMinusOne = 2^k*m
     **/
```

```
        Decomposition(BigInteger nMinusOne) {
            this.nMinusOne = nMinusOne;
            k = 0;
            m = nMinusOne;
            BigInteger two  = BigInteger.valueOf(2);
            while (m.getLowestSetBit() > 0) { // So m is even
                m = m.divide(two);
                k++;
            }
        }
        public BigInteger getM()  {
            return this.m;
        }
        public int getK()  {
            return this.k;
        }
    }
```

The main interest is the fact that the calculations take place in the constructor; the values obtained are then stored and are the available to the accessor methods. This is a very good example of when we do *not* want mutator methods; the variables are effectively "read - only".

I hope you find the next code fragment, the Rabin - Miller test, itself, to be a relatively accurate translation of its mathematical description above.

```
    /**
     * Apply the Rabin - Miller test with a single base to the
     * candidate <code>n</code>.  Because we are going to do this a
     * number of times, we precompute the decomposition of the prime
     * candidate externally and pass it as an argument to the method.
     * The second argument should actually be be n-1, which, we note,
     * is congruent to -1 (mod n).
    **/
    boolean probPrime( BigInteger n, BigInteger minusOne,
                       Decomposition decomposition) {
        BigInteger a;  // the base
        // Choose a random base a with a < p.
        do {
            a = makeRandom512();
        } while( a.compareTo(n) >= 0 );
        // z will sucessively take the values
        // a^m,a^{2m}, a^{4m}, a^{8m} etc, all calculated mod (n).
        BigInteger z = a.modPow(decomposition.getM(),n);
        if (z.equals(one)) {
            //  This is the easy case; the first term in the sequence
            //  is correct, so we pass the test.
            return true;
        }
        // Go through continually squaring z until we reach n-1.
        for (int i = 0; i < decomposition.getK();i++) {
            BigInteger zSquared = (z.multiply(z)).mod(n);
            if (zSquared.equals(one) && z.equals(minusOne)) {
                // We've passed the hard version of the Rabin Miller test.
                return true;
            }
            z = zSquared;
        }
        return false;
    }
```

This pass just does a single test, but multiple tests are easily driven from a method which calls the above method as follows.

```
    boolean probPrime(BigInteger n, int maxTests ) {
        BigInteger nMinusOne = n.subtract(one);
        // Note we could also call this minusOne, since
        // p-1 is congruent to -1 (mod p)
        Decomposition decomposition = new Decomposition(nMinusOne);
        for( int i = 0; i < maxTests; i++ ) {
            if (!probPrime(n,nMinusOne,decomposition)) {
                return false;
            }
        }
        return true;
    }
```

In use we first construct 512 bit candidates as described above and eliminate those which are composite with small prime factors using a sieve. The prime candidates are then subjected to the Rabin - Miller test with 100 different bases; even with this much work, a new prime is produced in a few seconds.

Bibliography

Bishop, J. & Bishop, N. (2000), *Java Gently for Engineers & Scientists*, Addison Wesley.
*http://www.booksites.net/bishop-jges

Burton, D. M. (1995), *Elementary Number Theory*, fourth edn, McGraw Hill.

Conte, S. D. & de Boor, C. (1980), *Elementary Numerical Analysis*, McGraw-Hill.

Davies, R. (1999), *Introductory Java for Scientists and Engineers*, Addison-Wesley.
*http://www.jscieng.co.uk

Deitel, H. M. & Deitel, P. J. (2002), *Java How to Program*, fourth edn, Prentice Hall.

Eckel, B. (1998), *Thinking in Java*, Prentice Hall PTR.
*http://www.bruceeckel.com

Faires, J. D. & Burden, R. (1998), *Numerical Methods*, second edn, Brooks/Cole.

Horton, I. (1999), *Beginning Java 2*, Wrox Press.
*http://www.wrox.com

Knuth, D. E. (1981), *Seminumerical Algorithms*, Vol. 2 of *The Art of Computer Programming*, second edn, Addison-Wesley.

Motwani, R. & Raghavan, P. (1995), *Randomized Algorthms*, Cambridge.

Press, W. H., Flannery, B. P., Teukolsky, S. A. & Vetterling, W. T. (1992), *Numerical recipes in C. The Art of Scientific Computing*, second edn, Cambridge University Press.

Rabin, M. O. (1980), 'Probabalistic algorithm for testing primality', *Journal of Number Theory* **12**, 128–138.

Winder, R. & Roberts, G. (2000), *Developing Java Software*, second edn, Wiley.
*http://www.dcs.kcl.ac.uk/DevJavaSoft

Index Entries