

# Programming in Java

A C Norman, Lent Term 2008

Part IA



# Contents

<b>1</b>	<b>Preface</b>	<b>7</b>
1.1	What is programming about? . . . . .	7
1.2	What about <i>good</i> programming? . . . . .	8
1.3	Ways to save time and effort . . . . .	9
1.3.1	Use existing resources . . . . .	9
1.3.2	Avoid dead-ends . . . . .	10
1.3.3	Create new re-usable resources . . . . .	10
1.3.4	Documentation and Test trails . . . . .	10
1.3.5	Do not make the same mistake twice . . . . .	10
1.4	Where does Java fit in? . . . . .	12
<b>2</b>	<b>General advice for novices</b>	<b>13</b>
<b>3</b>	<b>Introduction</b>	<b>15</b>
3.1	Introduction . . . . .	15
3.1.1	Books . . . . .	18
3.2	Practical work . . . . .	21
3.2.1	Exercises . . . . .	24
3.3	A Cook-book Kick-start . . . . .	28
3.3.1	Code Layout . . . . .	33
3.3.2	Emacs . . . . .	34
3.3.3	Drawing to a window: JApplets . . . . .	37
3.3.4	HTML and appletviewer . . . . .	42
3.3.5	Exercises . . . . .	43
<b>4</b>	<b>Basic use of Java</b>	<b>49</b>
4.1	Data types, constants and operations . . . . .	49
4.1.1	Reserved Words . . . . .	49
4.1.2	Basic Types . . . . .	51
4.1.3	Exercises . . . . .	65
4.2	Operators and expressions . . . . .	71

4.2.1	Exercises . . . . .	74
4.3	Control structures . . . . .	77
4.3.1	Exercises . . . . .	77
4.4	Control structures Part 2 . . . . .	82
4.4.1	Expression Statements . . . . .	82
4.4.2	Blocks . . . . .	82
4.4.3	Null statements . . . . .	83
4.4.4	if . . . . .	83
4.4.5	while, continue and break . . . . .	84
4.4.6	do . . . . .	84
4.4.7	for . . . . .	85
4.4.8	switch, case and default . . . . .	85
4.4.9	return . . . . .	87
4.4.10	try, catch and throw, finally . . . . .	87
4.4.11	assert . . . . .	88
4.4.12	Variable declarations . . . . .	88
4.4.13	Method definitions . . . . .	89
4.4.14	Exercises . . . . .	90
4.5	Java classes and packages . . . . .	98
4.5.1	Exercises . . . . .	108
4.6	Inheritance . . . . .	115
4.6.1	Inheritance and the standard libraries . . . . .	116
4.6.2	Name-spaces and classes . . . . .	120
4.6.3	Program development with classes . . . . .	125
4.7	Generics . . . . .	129
4.7.1	Exercises . . . . .	130
4.8	Important features of the class libraries . . . . .	139
4.8.1	File input and output . . . . .	140
4.8.2	Big integers . . . . .	147
4.8.3	Collections . . . . .	150
4.8.4	Simple use of Threads . . . . .	150
4.8.5	Network access . . . . .	153
4.8.6	Menus, scroll bars and dialog boxes . . . . .	155
4.8.7	Exercises . . . . .	160
<b>5</b>	<b>Designing and testing programs in Java</b>	<b>167</b>
5.1	Different sorts of programming tasks . . . . .	171
5.2	Analysis and description of the objective . . . . .	179
5.2.1	Important Questions . . . . .	179
5.2.2	Informal specifications . . . . .	180
5.2.3	Formal descriptions . . . . .	181

5.2.4	Executable specifications . . . . .	181
5.3	Ethical Considerations . . . . .	182
5.4	How much of the work has been done already? . . . . .	183
5.5	What skills and knowledge are available? . . . . .	185
5.6	Design of methods to achieve a goal . . . . .	186
5.6.1	Top-Down Design . . . . .	186
5.6.2	Bottom-Up Implementation . . . . .	189
5.6.3	Data Centred Programming . . . . .	190
5.6.4	Iterative Refinement . . . . .	190
5.6.5	Which of the above is best? . . . . .	191
5.7	How do we know it will work? . . . . .	191
5.8	While you are writing the program . . . . .	194
5.9	Documenting a program or project . . . . .	195
5.10	How do we know it does work? . . . . .	197
5.11	Is it efficient? . . . . .	200
5.12	Identifying errors . . . . .	201
5.13	Corrections and other changes . . . . .	204
5.14	Portability of software . . . . .	205
5.15	Team-work . . . . .	206
5.16	Lessons learned . . . . .	207
5.17	Final Words . . . . .	208
5.18	Challenging exercises . . . . .	208
<b>6</b>	<b>A representative application</b>	<b>219</b>
6.1	A Lisp interpreter . . . . .	219
6.1.1	Exercises . . . . .	233
<b>7</b>	<b>What you do NOT know yet</b>	<b>235</b>
<b>8</b>	<b>Model Examination Questions</b>	<b>237</b>
8.1	Java vs ML . . . . .	237
8.2	Matrix Class . . . . .	238
8.3	Hash Tables . . . . .	238
8.4	Compass Rose . . . . .	239
8.5	Language Words . . . . .	239
8.6	Exception abuse . . . . .	240
8.7	Queues . . . . .	240
8.8	Loops . . . . .	240
8.9	Snap . . . . .	240
8.10	Partitions . . . . .	241
8.11	Laziness . . . . .	241

8.12	Cryptarithmic . . . . .	242
8.13	Bandits . . . . .	242
8.14	Exception . . . . .	244
8.15	Features . . . . .	245
8.16	More features . . . . .	245
8.17	Debate . . . . .	246
8.18	Design . . . . .	246
8.19	Filter (Coffee?) . . . . .	246
8.20	Parse trees . . . . .	247
8.21	Big Addition . . . . .	248
8.22	Lists in Java . . . . .	248
8.23	Pound, Shillings and Ounces . . . . .	248
8.24	Details . . . . .	249
8.25	Name visibility . . . . .	250
8.26	Several Small Tasks . . . . .	250
8.27	Some Tiny Questions . . . . .	251
<b>9</b>	<b>Java 1.5 or 5.0 versus previous versions</b>	<b>253</b>
9.1	An enhanced <code>for</code> loop . . . . .	253
9.2	Generics . . . . .	254
9.3	<code>assert</code> . . . . .	254
9.4	Static imports . . . . .	254
9.5	Auto-boxing . . . . .	254
9.6	Enumerations . . . . .	254
9.7	<code>printf</code> . . . . .	255
9.8	Scanner . . . . .	255
9.9	Variable numbers of arguments for methods . . . . .	255
9.10	Annotations . . . . .	256
9.11	Enhanced concurrency control . . . . .	256

# Chapter 1

## Preface

### 1.1 What is programming about?

There are two stories you can tell yourself about what this course is going to do for you. The first is the traditional one that it is so you can learn some Java. Acquire knowledge and skills. The second, which may be more interesting, is to see this course as part of your journey as you start to become (or at least appreciate what it is to be) a Computer Scientist. This second perspective suggests that there may be something for you here whether or not you believe you are already skilled in Java, and it challenges you to look beyond the mere details to the tough patterns that link them together.

In the early days of computers programming involved a full understanding of the way that the hardware of your computer worked, your program, when run, took over essentially the whole machine and it had to include everything needed to manage input and output. In extreme cases one started the process of loading code into a computer by using hand-switches to place bit-patterns directly into the machine's memory. After a while operating systems came along and provided serious insulation from that level of extreme awareness of hardware, and high-level languages make it possible to express programs in at least semi-human-understandable form. But still the emphasis was on "writing a program", which tended to be a stand-alone application that solved some problem.

Libraries of pre-written sub-programs grew up, but for a very long time the ones that anybody could rely on having access to were either rather specialist or the functionality that they provided was at a rather low and boring level. There were libraries that could really help you with serious tasks (such as building a windowed user-interface) but none of them gained really global acceptance, and only a few were of any use on more than one brand of computer. The libraries that were standard with typical programming languages provided for fairly limited file

and terminal access input and output, modest string handling and really not a lot else. Operating systems made their capabilities available in the form of libraries that programs could call on, but overall coherent design was rare and use of these “libraries” led to inherently non-portable code.

Building a new library was not part of the common experience of programmers, and indeed large-scale re-use of code was the exception rather than the rule.

There has been an ideal or a dream of re-usable software components for ages, but it is only recently that it has started to become something that can be not just feasible but reasonably convenient. Java is one of the languages that encourages this move, and the whole Object Oriented Programming movement that Java forms part of provides a context.

So in the old world one thought of a program as a large complicated thing that called upon facilities from a few fixed libraries that you happened to have available. Today instead of that you should often start a project with the intention of developing a set of new re-usable and general libraries that themselves build on and extend existing software components. You will design these libraries so that once they exist the program you had to write becomes a fairly simple application of them: it will do some minor customisation and link together different units within the overall structure of your libraries, but with luck it will of itself be fairly small and straightforward. If you do this well you will find that the library you have created will serve you well in future projects, or it may even become something worth circulating (or selling) of itself. With these ideas in mind you will want to make it well-structured, robust and you may even feel motivated to accompany it with some coherent documentation!

So overall the mind-set for the 21st Century is that you design and write re-usable components and libraries, and that writing mere stand-alone programs is a terribly old-fashioned and dull thing to do!

## 1.2 What about *good* programming?

The first and utterly overriding character of a good program is that it must be fit for its purpose. Good programming must not only lead to a good program, but should do so in a way that reaches a successful conclusion reliably and without taking more time and effort than is really required.

These comments may seem bland and self-evident, but they have real consequences! The first is that you can not judge a program until you know what its purpose is. Even though almost all the exercises you will do this year will be both small and will never have any part of their code re-used it will be proper for you to practise writing them as if they are much larger and more important. That will mean that you are expected to accompany the code you write with both notes



about its external behaviour and how to use it and with comments that describe its internal structure and organisation. For certain sorts of library code it will make sense to use the documentation arrangements that the main Java libraries use. This involves things called “documentation comments” and a utility called `javadoc` that will be described later.

Without this documentation *you* may believe that your programs meet their purpose but you do not have any basis for expecting others to agree.

## 1.3 Ways to save time and effort

Working with computers can swallow up an astonishing amount of time. To be able to get everything you need done you will want to find ways of economising. The key to doing this effectively is to concentrate on techniques that save time in the long run. Some ideas that appear to speed things up in the short run can end up costing more later on!

### 1.3.1 Use existing resources

You are encouraged to use code-fragments from these notes in any way you want. You can sometimes get a fresh project off the ground by extracting at least fragments from a previous piece of work you have done. The Java libraries are your friend: they contain facilities to do many of the things you will find yourself needing. In general before you ever write anything from scratch for yourself consider whether there is something that can give you a head-start.

Everybody might reasonably worry that the above paragraph could be seen as an invitation to plagiarise! Do not take it that way: couple it with a very firm remark that when you use other material you should acknowledge your sources, and you should not pillage the material of those who are unwilling to make their work available to you. As far as tickable exercises for this course are concerned you are *encouraged* to discuss what you are doing with friends and supervisors, and collect code-sketches and fragments from them, provided that when you submit your work to the department you really understand everything in your submission and you have learned enough that (if necessary) you could then instantly and comfortably re-create your submission in a sound-proof booth visibly cut-off from further help. So rather than sit and suffer in isolation, seek web-sites, friends, demonstrators, books and code libraries to give you guidance so long as you learn from them and do not just blindly copy!

### **1.3.2 Avoid dead-ends**

Sometimes you can start designing or writing some code and as you go things seem to get harder and harder. Something is not working and you have no idea why. You do not want to get in such a state! Clear advance planning and a well organised set of working habits are the best way to avoid the mess. If you find yourself in what feels like a dead-end then avoid (a) panic (b) a tendency to try almost random changes in the hope that things will improve and (c) temptation to work all afternoon, evening and night until you solve things. Go back and look at your plan. If necessary refine it so you can make progress in tiny steps. Explain your plan and your code to somebody else (either in person or in the form of written documentation). But do not just get bogged down: taking a break and coming back fresh can often save overall time.

### **1.3.3 Create new re-usable resources**

An ideal that this course would like to instil in you is one of creating re-usable bodies of code. This will take more care and time when you first implement them (and of course anything re-usable deserves proper documentation and testing) but that can be well paid back when you get a chance to call on it again. At a minimum this can include keeping all your working code from both the tickable and other exercises in these notes so you can use parts of them as templates in future projects.

### **1.3.4 Documentation and Test trails**

Neatly formatted code with clear comments and a well set out collection of test cases can seem slower to write than a jumble of code that is just thrown together. However long experience suggests that the jumble of code is much less likely to work first time, and that especially as your projects get bigger that early investment in good habits pay dividends.

### **1.3.5 Do not make the same mistake twice**

Especially while learning a new language, such as Java, you will make mistakes. As you design and write gradually larger and larger bodies of code you will make mistakes. Observe and appreciate these, and try to observe yourself as you uncover and correct them. Possibly even keep a small notebook of “bugs I have had in my code”. Then each time you make a mistake seek some scheme that can prevent the same one from causing significant trouble in the future. Your fingers will always leave typos in everything you write and your mind can always wander: the

idea is not to avoid glitches totally, it is to build up a personal toolkit of ways to overcome them without pain or waste.

## 1.4 Where does Java fit in?

There are those who believe that Object Oriented Design and Programming is **The Answer** to reliable large-scale system building, a silver bullet<sup>1</sup> that cures the major woes of the last fifty years of over-costly and haphazard use of computers. Java is one of the major practical and widely-used languages that fall within the Object Oriented family. Key attitudes that come with this are that projects should be structured into potentially re-usable blocks (the Java `class` construct that you will learn about later being a major way of achieving this). These blocks should each take responsibility for just one aspect of the overall behaviour you are trying to code up. The decomposition should be arranged so that interaction between blocks is as tidy and disciplined as possible.



Overall at least a rough caricature is that ML stresses absolute correctness via mathematically styled structure, and encourages very concise programming styles. Java on the other hand follows a view that language constructs that support large-scale structuring of projects are the key. It also expects that having the user write out types and qualifiers explicitly will help others to read your program. ML as taught last term provides a fairly basic library, but mostly you spend the Michaelmas Term writing stand-alone programs and fragments. With Java there is heavy emphasis on a rich (and perhaps hence complicated) library that supports a very full range of computing needs.

Figure 1.1: Silver Bullet Needed.

---

<sup>1</sup>Brad Cox in Byte magazine October 1990, pp 209–218 puts things in much these extreme words.

# Chapter 2

## General advice for novices

Following tradition, I provide ten items of guidance for the benefit of those who are relatively new to programming. I hope that each of these will be re-inforced during the course as a whole, but here they are collected together at the beginning:

- 1 Understand the task you are about to solve before starting to write a program about it. Work through methods and procedures by hand on paper etc. Plan some test cases. Identify cases that will represent boundaries or oddities. In general prepare a plan before you start going anywhere near a computer;
- 2 Sketch the structure of the whole of your code out informally so you have full overview before fussing about exact syntax etc. Ensure you know what you expect that the computer will do. This initial sketch can be very informal, and may be in terms of diagrams rather than anything that looks much like real programming. The key word here is “structure”. This applies with way greater force when your code starts to grow: you should always design a good way to factor your code into reasonably self-contained and independent components (each will be one “class” in your code) right from the start;

- 3 Write out **key** parts of above in the form of comments before you start the real code. Concentrate in these comments on the “what” and “why” of your code rather the details of “how”. This will really help when you show your work to somebody else because you need help! I will explain this one again: The first thing you will type into a computer when you start writing any program will be a set of overview comments that explain its strategy and structure;
- 4 At least for a first version of anything, favour clarity and obvious correctness over pretty well everything else. Clever tricks, worries about efficiency, generalisations etc can come later;
- 5 Neat consistent layout and thoughtfully named fields, methods, variables etc. are a good investment of your time. Cryptic is bad even if it saves keystrokes in the short term;
- 6 If a task is too big to solve in just one gulp look for ways of breaking it down into sub-tasks. As you do this think about ways you will be able to test code you write for each sub-task and work on the whole thing step by step;
- 7 When you try to compile your code and see a syntax error do not panic. Learn to interpret the compiler’s diagnostics. And only try to remove one error at a time: count it as a success if next time you try to compile the *first* error has give so you can then concentrate on the second;
- 8 When you have compiled your program and run it and it gives wrong answers or behaves badly do not panic. First work to understand what is wrong and only after you have found where the problem is think about ways to fit it. Do not just try random changes! Eg. confirm what your program actually does by adding assert and extra print statements;
- 9 Whenever you find you have to change your program review comments, consider if it will now do exactly what you want, and re-run all your test cases. Experience shows that changes (for whatever cause) can introduce new problems while you are in the process of fixing old ones;
- 10 If you find you are spending a seriously long time trying to make sense of anything then find help from friends or a supervisor or a book. Do not just keep building up your frustration not getting anywhere!

# Chapter 3

## Introduction

### 3.1 Introduction

We have been using Java as a first-year teaching language here in Cambridge since 1997-8. We teach this course following on from “Foundations of Computer Science” which used ML, and there are a number of things it is intended to do:

1. Provide all of our students with exposure to a common programming language that can be used by later courses and practical work in the CST;
2. Introduce the syntax that is (almost) common to several of the most widely used practical programming languages today (the syntax of Java has a great deal in common with that of C and C++, so having learned Java you are quite a long way to understanding those languages too);
3. Discuss the process of designing, writing and debugging programs and raise some awareness of issues of style;
4. Present the Object Oriented aspects of a programming language as means to enforce modularity in large programs;
5. Teach basic use of Java, a language that has significant relevance in the outside world today.

Note that in our Part IA course “Software Engineering II” provides significant extra coverage on issues of structuring programs (especially ones that are large or developed by collaborative work in a group), and in Part IB course there is a lecture course once entitled “Further Java” and now renamed “Concurrent Systems and Applications”: it should not be imagined that I will cover all aspects of the language or its use here!

The nature of teaching a course involving programming in some particular language means that some features need to be mentioned well before the place where they can be fully explained, and so it will not make sense to keep the presentation in lectures totally linear and tied to these notes, but for supervision purposes the structure shown here should suffice. With each section I will have a few examples or exercises. Especially at the start of the course these will often be pretty silly, but the ones right at the end can be viewed as samples of the sort of question that might arise in the examination. Although I want some of my examples to be nice and easy I would like to have others that are interesting challenges for those who already think they know it all (ha ha). It is always very hard to judge the amount of trouble these will give you all, so if they are either too easy or too difficult I apologise. Examination questions will be set on the supposition that you have attempted a reasonable sampling of the exercises.

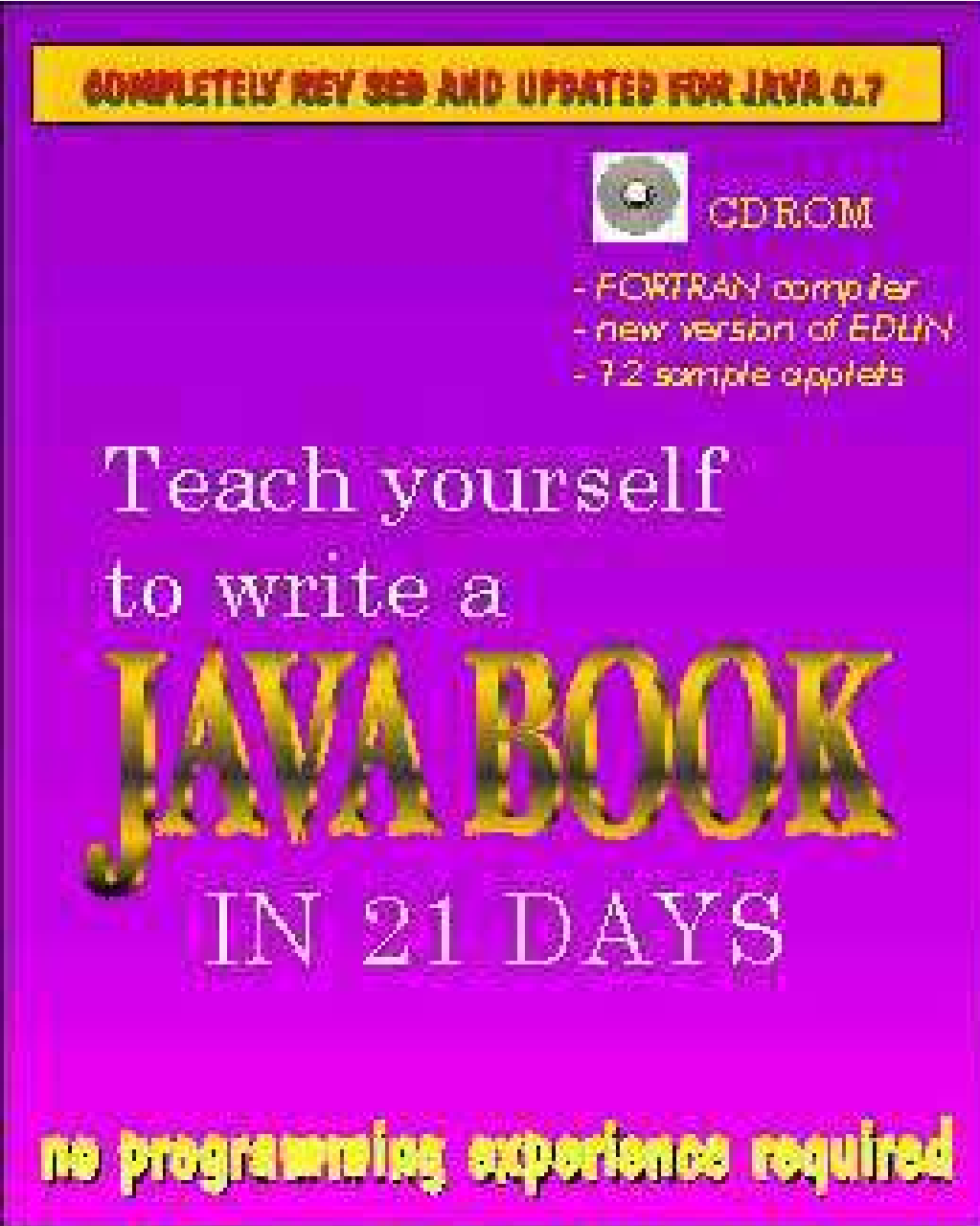
The aim of these notes is that they should serve both as guidance to students and to supervisors, and so there is no separate supervisor's guide. Originally I had intended that they would be structured into sixteen sections corresponding to the sixteen lectures available. As I prepared the notes I concluded that such a rigid arrangement was not tenable. Thus the lectures can be expected to cover roughly the material in these notes in roughly the same order, with an approximation to one-sixteenth of the entire notes corresponding to each lecture!

It might be noted that a Java course for the Diploma students runs during the Michaelmas term. The lecture notes associated with that course may provide a presentation of Java which is different from mine and thus may complement my lectures or shed light on issues that I fail to.


The course is based on use of the version of Java sometimes known as "Java 5.0" and sometimes as "Java 1.5" or 1.6. This should now be counted as the current and widely-used version, but if you use computers other than the main university ones here you may come across earlier releases. Please avoid them for course-related work to avoid confusion.

Some members of the audience for this course will already have significant practical experience with Java. Others will have written lots of programs before but in C, C++ or Pascal, but the only thing I can properly assume here is that everybody has attended the Foundations of Computer Science course given in the Michaelmas term and hence that everybody is used to writing code in the language ML. While those who have seen Java before will undoubtedly find the first few lectures and exercises here very easy, I hope that they will find material that is new and worth-while being introduced in due course. In the first year that this course was given it was observed by one Director of Studies at a large College that some of his students who did already know Java concluded on that basis that they need not attend the lectures, but that their examination results indicated that this had not been a perfect judgement call.





**COMPLETELY REVISED AND UPDATED FOR JAVA 0.7**

 **CDROM**

- FORTRAN compiler
- new version of EDLIN
- 72 sample applets

Teach yourself  
to write a  
**JAVA BOOK**  
IN 21 DAYS

**no programming experience required**

Figure 3.1: Reproduced courtesy Kevin McCurley.

### 3.1.1 Books

All bookshops these days seem to devote many metres of shelf-space to books that purport to teach you Java in a given small number of days, to help you even if you are an “idiot”, or to provide “comprehensive and detailed” coverage of even those parts of Java that the language definers have left deliberately vague. I believe that this is a course where it is important for every student to have their own copy of a supporting textbook/manual. But the issue of which book to buy will end up a somewhat personal choice since differing levels of detail will suit different students! Browse the following in libraries and bookshops, talk to students in higher years and seek advice from your Directors of Studies and Supervisors about what is liable to suit you.

My first recommendation has as most of its pages what is in effect hard copy of the on-line detailed documentation of the Java library. As a result it is not a smooth consistent read, but I find that very many people need that information in printed form while they are getting used to navigating the library and understanding what it can do for them. *Java in a Nutshell* fifth edition (Feb 2005)

*Java Foundation Classes in a Nutshell* (1999)

David Flanagan  
O'Reilly

There are two books[11, 6] that I think you might reasonably consider and that are probably easier for self study in that they do not get so rapidly enmeshed in full detail.

*Thinking in Java*

Bruce Eckel  
Prentice-Hall, 2002 *third edition*

and

*Java Gently*

Judy Bishop  
Addison Wesley, 2003, *third edition*

Eckel's book is distributed (at no cost) via [www.eckelobjects.com](http://www.eckelobjects.com) so if you are actually prefer reading computer screens to real books it counts as a great bargain!

Some Directors of Studies will strongly point you towards the book[2] that was that main text for this course a couple of years ago:

*Objects First with Java: a Practical Introduction using BLUEJ*

David Barnes and Michael Kölling  
Prentice Hall/Pearson, 2005, *second edition*

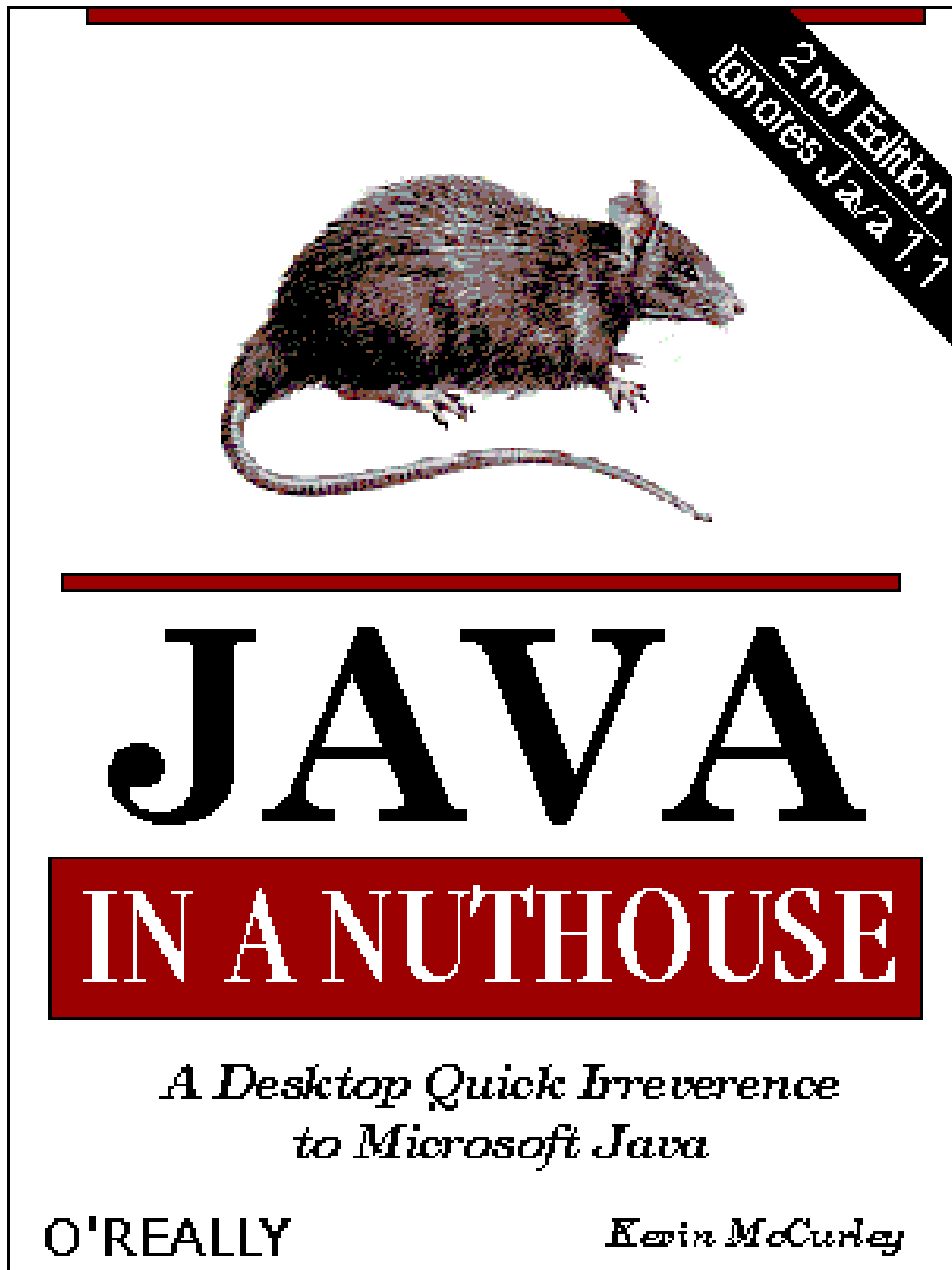


Figure 3.2: Not (quite) the main course book.

This book emphasises issues of overall program structure and design above concern for the exact details of the Java language or its libraries and so is almost exactly the antithesis of the Nutshell books! It was used as the main teaching text here in 2003-4, and you may find the BlueJ software (<http://www.bluej.org>) provides a useful environment within which to develop and test your code. Note that this year's edition of both the book and the software has developed from the versions available last year.

There will be plenty of other useful books, and any individual student may find a different one especially to their own taste. If selecting a book other than the one I suggest that you make sure that what you learn from it can be related to the lectures that I give. Since this year we are using a rather new version of Java beware that old editions of books may not be sufficiently up to date.

Java is a “buzzword-compliant” language, and when people hear that you are learning it they will instantly pick up all sorts of expectations. Even though this course is sixteen lectures long I will not be able to fulfil all of these, and that is in part why the Computer Science Tripos has a course entitled “Concurrent Systems and Applications” in Part IB that follows on from this one. There are three issues that I should mention right here at the start of the notes, if only to protect myself and the department against misunderstandings as to our purpose:

**Java is for use animating Web pages:**

Some of the huge first flush of enthusiasm that greeted the emergence of Java was because it could be used to make rather naff animated figures dance on web pages. This was of course amazing when web pages had previously been so rigidly static, but it is not a good model for the central issues in Computer Science. This will typically not be the sort of use of Java that we try to teach you here;

**Java is the best programming language:**

The Computer Laboratory shows by its actions that it views ML as its preference for a first language to teach its students, with Java as a second one. Later on in the course we will provide coverage ranging from brief mention to detailed explanations of quite a few other languages: certainly C, C++, Lisp and Prolog. The Software engineering courses mention a scheme called just ‘Z’ that is in effect a programming language, and you will see from past examination papers that we have high regard for Modula 3. What is shown by that is that the Computer Laboratory view is that different languages may prove best for different tasks, and that the optimal choice will change as the years go by (it happens that we no longer teach our students either Fortran or COBOL, and our coverage of assembly code is present because it forms an important link between the concerns of hardware designers, operating system experts and compiler writers, and not because we

expect students to do project work in it). At present Java is our choice for the first “traditional”-style programming language we teach: this does not mean it will automatically be the only or best choice for **all** future practical work and projects;

**Students should be taught about “programming in the large”:**

As this is a first year course I will be concentrating on the fundamental building blocks of program construction. This is in line with the Engineering Council “EA1” concern about introducing students to the fundamental tools, materials and techniques in their subject. I view it is self-evident that until a student can write small programs competently and painlessly it would not make sense to expect them to be able to work in groups on large projects. However in all the practical work associated with this course you should expect the assessors to demand that all code you write is well laid out, properly commented, that it displays a sensible programming style and that you are in a position to justify its correctness. In short that a generally professional approach has been taken even though many of the exercises are short and somewhat jokey toy problems.

## 3.2 Practical work

The main environment the laboratory expects you to use for this course is PWF Linux. At the start of Term you should be given an introduction that explains how to re-boot certainly the PWF systems in Cockroft 4 or in the Intel Laboratory in the Gates Building so they run Linux. PWF workstations in other parts of the University may not have been configured with this dual-boot option, but if they have then you can use them. Although Java runs perfectly happily on Windows we want you to do much of your practical work on Linux so that by the time you come to the Operating System course later in the year you have made significant personal use of both Windows and Linux.

At least for the first half of the Term we would also encourage you to use the `emacs` editor and build and run your Java programs using the somewhat primitive command-line driven tools `javac`, `java` and `appletviewer`. Use of these will be explained later. The reasoning behind this is not that it guarantees to make your Java-specific experience as comfortable as possible, but because the technologies involved are ones you need to find out about at some stage! Specifically I note that

- `emacs` is a rich and powerful editor. You can use it in a simple way while you are beginning work, but it has extension mechanisms that allow it to morph to provide specialist support for different sorts of document, and it

can provide a single environment (and set of keystrokes to learn) that covers not just editing your program but also compiling and running it, reading and sending e-mail and many other tasks. It probably counts as the most widely used general-purpose Unix/Linux editor and versions for Windows are also available. Your really simple use of it now will help those of you who choose to use it in more elaborate ways later on.

- The use of the `javac` and `java` commands explicitly (as distinct from you using them implicitly through an all-encompassing specialist Java development environment) means that when you see any curious messages or complaints you know where they come from. It also introduces you to a typical model for how software is built (the *edit, compile, test* cycle). When you are more experienced you will no doubt move on and use integrated environments<sup>1</sup>. In some respects these help by doing things for you – but especially since you have survived the Foundations of Computer Science course last Term it now seems proper that you get to see how to do things for yourself.

For reference material it may prove most convenient to use on-line documentation, and in particular the web-browsable HTML version. This is available to you in `$CLTEACH/acn1/java/docs`, so you can launch a browser and start looking at it by going

```
firefox $CLTEACH/acn1/java/docs/index.html &
```

and around the first thing you may want to do is to set yourself a bookmark on that page. There is a *huge* amount of documentation there. The bits I find most useful are the “Java 2 Platform API Specification” which documents (in painful detail) all of the library facilities that are provided, and the “Java Tutorial” which links to a Sun website with much helpful explanation, and which you may find a very good complement to the textbooks I have suggested. All the time I am writing any Java code at all I will have a web-browser open on the “API” section of the documentation, since it is useful to have a quick way to check details of the library very close at hand.

You can obviously run PWF Linux in one of the big shared workstation areas, and there is a great deal to be said for at least starting off that way: you can compare notes with other students when you have problems. But you can also access PWF by using `ssh` and an “X-windows server” to access one of the lab’s PWF linux systems that are set up for remote use, eg `linux2.pwf.cl.cam.ac.uk` or

---

<sup>1</sup>Microsoft’s Visual Studio is perhaps a definitive example: for Java you can install either Netbeans (from Sun) or Eclipse (from IBM) free of charge. BlueJ has very different objectives but may also prove useful to some.



Figure 3.3: Remember about RSI, posture etc, please.

linux.pwf.cam.ac.uk. If your own computer is set up to run Linux those will already be present for you. If you run Windows you can get good versions free of charge by installing a Unix-compatibility layer from <http://www.cygwin.com>, but getting everything to work nicely there may be messy enough that those of a nervous disposition would do better to work in Cockroft 4 or one of the College computer rooms where PWF Linux is directly available!

It is also perfectly in order for you to install Java on your own computer. Apart from the fact that the Java development kit uses around 450 Mbytes installing it should not prove hard, it does not cost anything and performance should work well under either Windows, Linux or MacOS on any even reasonably recent pc. If you do that you must be willing to take full responsibility for installing and maintaining everything, and should take care to back up all important files. For just running small Java exercises there should not be much difference in the experience you have using your own rather than a public machine<sup>2</sup>, however if you habitually use a PWF system somewhere other than in Cockroft 4 or the Intel laboratory your own system might reduce your need to wait while you re-boot a public machine into Linux, and if you experiment with one of the integrated Java environments you may find performance much better on your own system. If you have a Macintosh note that Java 1.5 has only very recently become available, so please double-check that that is the version you have.

To fetch a Java compiler you will need to connect to

<http://java.sun.com/j2se>

where you can find the Java “SDK Standard Edition, version 5.0”, and its accompanying documentation. You should be aware that the package you have to download is around 50 Mbytes for the main kit, with the documentation being an additional large download and the “Netbeans” development environment yet more that you *may* want to explore but are not obliged to worry about. Sun can supply either Windows (2000/XP) or Linux versions of all of these.

The Eclipse development environment can be found at <http://www.eclipse.org>.

### 3.2.1 Exercises

#### Tickable Exercise 1

The first part of this tickable exercise is issued as part of the introduction to the use of Linux on the PWF. The task set here is thus “Part B” of the complete Tick.

Log on to the PWF. Create a new directory and select it as your current one, eg

---

<sup>2</sup>Great thanks are due to the Computing Service for ensuring that this is the case.



```
mkdir Tick1B
cd Tick1B
```

On the lab's PFW Unix systems issue the following commands that copy two files from the Computer Lab's teaching filespace into your new directory.

```
cp $CLTEACH/acn1/TickBase.class .
cp $CLTEACH/acn1/TickBase1.class .
```

You should be able to check that the files are present. These two files provide a basis upon which the exercise builds. Alternatively you can download the two ".class" files from the "Material provided by the lecturer" web pages of the Computer Lab's web-site, <http://www.cl.cam.ac.uk/teaching/0708/ProgJava/>.

Now inspect Figure 3.4 which is documentation associated with the two files that you have just copied. A more extensive version of the same material is available on-line as

[www.cl.cam.ac.uk/Teaching/current/ProgJava/notes/TickDoc/](http://www.cl.cam.ac.uk/Teaching/current/ProgJava/notes/TickDoc/)

Now prepare a file that called `Tick1.java` containing the text

```
// Tick 1. Your Name Goes Here

public class Tick1 extends TickBase
{

public static void main(String []args)
{
    (new Tick1()).setVisible(true);
}

public String myName()
{
    return "Your Name";
}

}
```

Obviously you will put your own name in the places that are suggested by what I have written here!

Compile your program and then run it:

```
javac Tick1.java
java Tick1
```

## Class TickBase

```
public class TickBase
```

The "TickBase" class provides a foundation for Java Tick 1. It is intended to be used by writing a new class that extends it and provides it with a "main" method, as in

```
public class Tick1 extends TickBase
{
    public static void main(String []args)
    {
        (new Tick1()).setVisible(true);
    }

    public String myName()
    {
        return "Arthur Norman";
    }
}
```

and this will lead to an application that can be launched and will display a certificate confirming success.

## Field Summary

## Constructor Summary

**TickBase()**

The sample version of "main" uses a constructor called TickBase() that makes an instance of the object that displays certificates.

## Method Summary

java.awt.Color	<b>myColour()</b> By overriding the myColour method you can change the colour used in the certificate.
java.lang.String	<b>myName()</b> Override the myName method with a version that returns whatever your own name is to personalise the certificate that you receive.

## Constructor Detail

### TickBase

public **TickBase()**

The sample version of "main" uses a constructor called TickBase() that makes an instance of the object that displays certificates.

## Method Detail

### myName

public java.lang.String **myName()**

Override the myName method with a version that returns whatever your own name is to personalise the certificate that you receive. To receive a tick you are required to do this.

### myColour

public java.awt.Color **myColour()**

By overriding the myColour method you can change the colour used in the certificate. The default is Color.BLUE

Figure 3.4: Documentation of Tick 1 Part B.

If all has gone well a window should appear, and it should have some text and a pattern on it. There is a menu that you can select. If you copy the files to your own machine you can try the `print` menu, but on the PWF there are technical reasons why that is not supported, and these lie outside just Java. So select the menu item labelled `postscript`. You should then see a dialog box asking you to choose a file name. I suggest that you select the name `tick1.ps` and I very strongly suggest that you use the extension `.ps` whatever name you actually choose. When you accept the file-name you have chosen the “select file” dialog box disappears and you can not see that anything much has happened, but the file you indicated should have been created for you. It should contain an image of the screen window in the Postscript document format. Close the little Java window, and you can send this to a printer using the command

```
lpr tick1.ps
```

The resulting sheet of paper is what goes to your ticker.

As an optional extra you can arrange to change the colour of (some of) the text generated by adding lines roughly like the following to your Java source file:

```
public java.awt.Color myColour()
{
    //                      RED GREEN BLUE
    return new java.awt.Color(0.7f, 0.1f, 1.0f);
}
```

where the three floating point numbers given (note that you have to write a letter ‘f’ at their end) should each be in the range 0.0 to 1.0 and they give the proportions of red, green and blue in the colour.

You can also check what happens if you present your name in different ways. For instance I tried “A C Norman” as well as “Arthur Norman”. If you wanted to keep your program in a file called say `MyTick.java` rather than `Tick1.java` you would have to change its name within the file too. Verify that you can do that.

## Discussion

This exercise is intended to send several signals and messages about Java:

- One can build new programs building on existing components that do quite a lot for you. Here you copied in the `TickBase` class files, but your own program then builds on them and can customise the behaviour of the provided code in various ways. Through doing this a very short fragment of code let you create a window and print its contents;

- To use the software component TickBase you do not need to see its internal structure: all you need is documentation about how to use it. As part of stressing this I am not going to provide you with the source code of TickBase, but by the end of this course you will probably be able to re-create it;
- Part of the way of using components like this involves the Java keyword `extends`, and part of the way that the code runs involves the keyword `new`. These are both key parts of the Object Oriented structure of Java, and you should look forward to finding out more about just what they really mean later on.
- The page of documentation included as part of these notes tells you that TickBase is interested in a `myName()`. This documentation is in the style of the bulk of the Java on-line documentation, and was created by using a simple tool called `javadoc` that interpreted some special comments in the TickBase source code. However the full output from `javadoc` is on the web page listed a little earlier and perhaps gives a bit more of an idea of just how much complexity is involved under the surface. The lesson that I learn is that if you use `javadoc` for anything other than a full-scale project you will need to edit its output heavily to remove material that your audience does not really need to see.

*(End of tickable exercise)*

### 3.3 A Cook-book Kick-start

In this section I will try to get you started with Java. This means that all sorts of aspects of it will be described in an order that is not really logical, but is motivated by that fact that some features of the language must be described early if you are to get any programs at all written. I will not provide much justification for the recipes that I give. Later on it will be possible to re-visit these examples and understand what the various odd keywords are all saying and what options might be available, but for now you can just copy them out parrot fashion.

I would like you to type in all the examples for yourselves and try them out, since that will educate your fingers into following the rules that Java imposes, and it will also (each time your fingers stray) give you exposure to Java error messages and the joys of finding and fixing mistakes.

My first example in fact is an echo of the first part of Java Tick 1. A mildly silly tradition in teaching programming languages is that the first program presented should just print out “hello”. The way of doing this in Java looks like this:

```
System.out.printf("Hello");
```

which is a call to a library function called `printf`<sup>3</sup> that will display the given string. The prefix “`System.out`” is not part of the name of the function — it happens to be providing the instruction that that printed output should be sent to the standard output stream, ie typically straight to your screen or terminal. In essential terms the line shown above is the whole important part of your first Java program. However there is actually quite a lot more to be discussed before you can try it!

The first thing is that Java is a *compiled* programming language, so unlike the situation you have seen in ML it is essential to place your program in a file before it can be used. In this case you should use a file called `Hello.java` and it is essential that the file name should start with the word `Hello` since that is the name that we will soon repeat within the file. The spelling should be with a capital letter as shown<sup>4</sup>, and the file-name should be completed with the suffix `.java`.

If you start `emacs` and use the menu selection “Files/Open File” you get a chance to create a new file for this project, and if you may<sup>5</sup> notice that when you type in the string in the example it is displayed in an alternate colour (to help remind you to match your quote marks), and when you type the close parenthesis after the string the matching bracket gets flashed to help you keep that side of things under control.

It is possible to make very extensive customisations of `emacs`. If you put a file called `.emacs` in your home directory it can contain directives that apply whenever `emacs` starts. In particular if you put a line

```
(global-font-lock-mode t)
```

then you will get syntax colouring enabled every time: I find this convenient. For now I suggest that you avoid putting large amounts of other clever stuff there!

You will also see that the menu-bar at the top of the `emacs` window has entries that let you do all the things that editors ought to — and more besides. See figures 3.5 and 3.7: note that the printed form of my notes will be in black and white but the downloadable version on the lab’s web page

<http://www.cl.cam.ac.uk/Teaching/2005/ProgJava>

will show relevant information in colour. Also note that the sample programs being edited and tested in the pictures of `emacs` in use may be ones taken from previous years’ versions of this course.

---

<sup>3</sup>Many Java texts use a function `println` here rather than `printf`.

<sup>4</sup>Well actually if you are working on a Windows system the capitalisation is not so important, but even there you are strongly advised to keep to it so that when you transfer your programs back to Unix before showing them to the assessors they still work!

<sup>5</sup>Provided the “global font lock” options is selected.

A “Java mode” is automatically selected when you edit a file whose name ends in `.java` and this is the first pay-off you see from this convention. If you select a “global font lock” this can colour your code so that language keywords, strings, comments and so on are all displayed in different colours<sup>6</sup>. It also assists with indentation and provides editing commands that move around in the file in a way that understands Java syntax. A major feature of `emacs` is that it is amazingly customisable, and configuration files can provide it with special support for many languages and layout conventions. If you browse enough sites on the web you may find many extra options that you can install: hold back and avoid these until you have got really used to the default setup! Please!

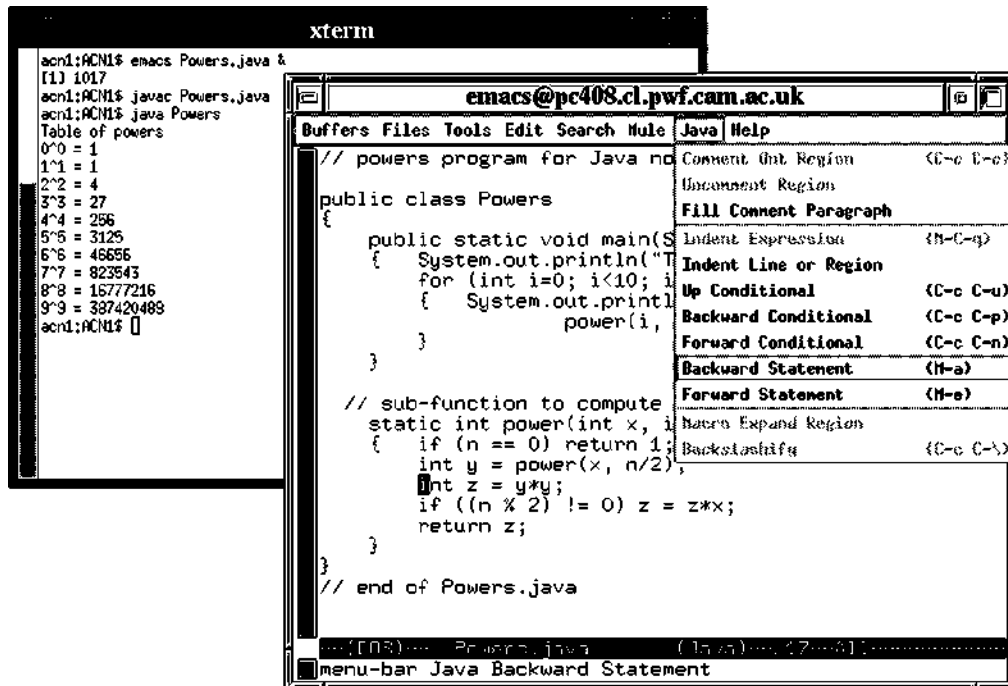


Figure 3.5: Two windows, with emacs editing a program.

Your complete first Java program needs a great pile of guff that surrounds the one interesting line we have and turns it into something that can be executed. In essence two things need to be documented. The first is something that indicates the external name that the program will be known by. This will always be exactly the same as the start of the name of the file it is stored in. You may consider it silly to have to re-state information that looks as if it should already be available, but for now please suspend disbelief and accept that a program that lives in a file called `Hello.java` will have to contain the text

<sup>6</sup>At a minimum this can be very helpful if you accidentally fail to close a string or comment!

```
public class Hello
{
    ...
}
```

where the ... will be filled in soon with material that includes our call to `System.out.println`.

The second piece of information needed is an indication of where Java should start its processing, and the convention that the language imposes here is that it expects to find a procedure<sup>7</sup> with the name `main`. The definition of a suitable procedure then involves the incantation

```
public static void main(String[] args)
{
    ...
}
```

of which the only word that is currently worth describing is “main”, which is a reminder of the historical tendency to refer to the place where a program started as being the “main program” while what are now known as functions or procedures might have been called “sub-programs”.

Comments can be introduced by “//” and every good program starts with a note that explains a few key facts about it. Obviously the longer the program the more that it will be proper to put in comments at both the start and throughout your code, but note that assessors will certainly expect your name and the exercise identification to be at the head of every submission you make.

Putting this all together we get the full contents of the file `Hello.java` as

```
// This is the file "Hello.java" prepared by A C Norman
// and the program just prints a fixed message. 1998-2006.

public class Hello
{
    public static void main(String[] args)
    {
        System.out.printf("Hello%n");
    }
}
```

---

<sup>7</sup>In these notes I will use the terms “function”, “procedure” and “method” pretty-well interchangeably. Some other languages use these words to indicate refined differences — typically the term “procedure” would be something that did not return a value, while a “function” would. The word “method” comes out of ideas of so-called Object Oriented Programming and indicated a function that is defined within a “class”. Although I have not yet explained what a class is we have seen the keyword `class` towards the head of our Java programs.

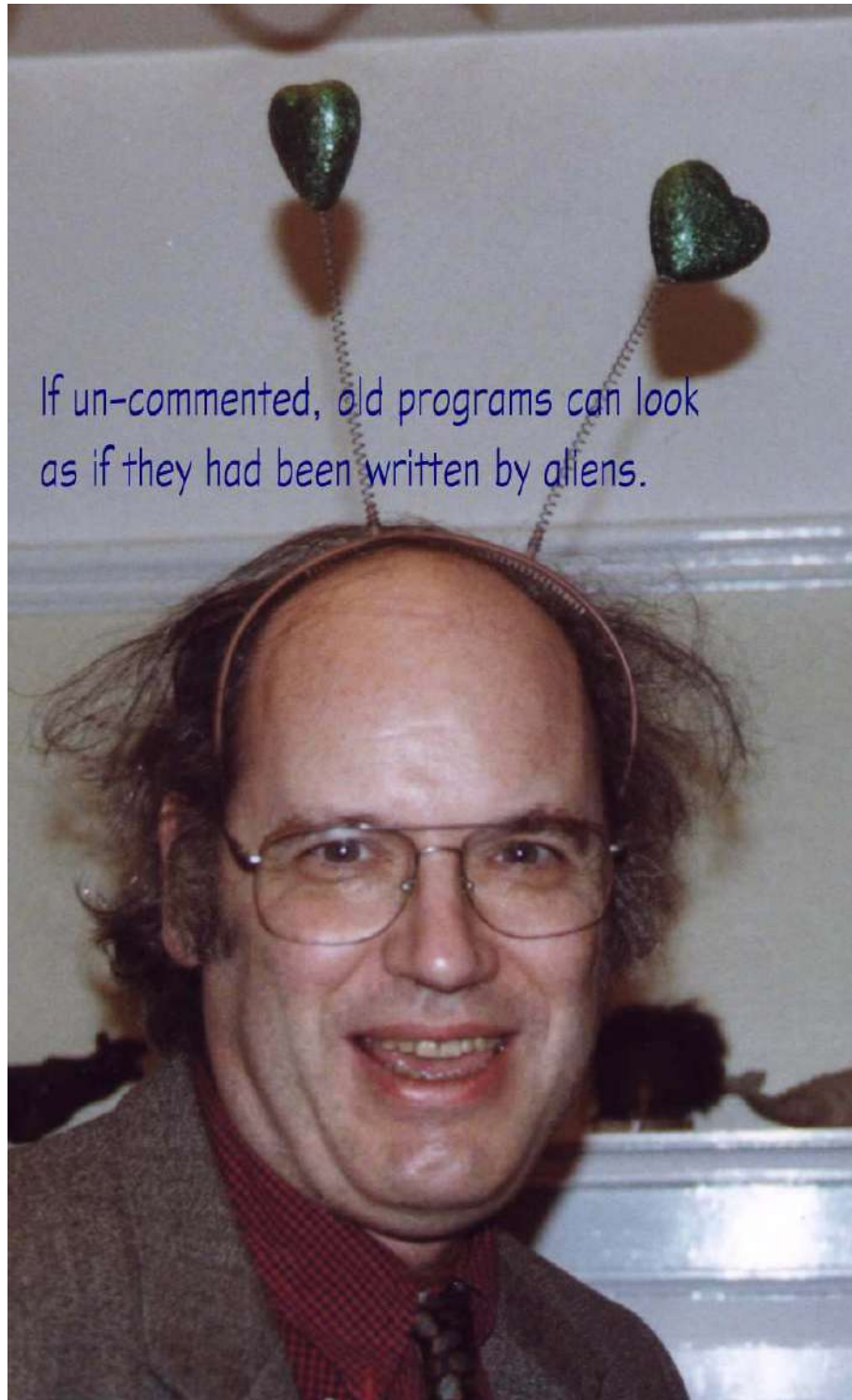


Figure 3.6: Style does matter.



There is a yet further odd addition in what I have just shown. The `%n` arranges to put a newline at the end of your message.

For a very short program that hardly does anything interesting that seems to have a lot of “magic” keywords. But in only a few weeks time you will know what they all mean, and why they make sense. For now just keep a file that contains the above basic sample code and copy it every time you want to start a new program so you do not have to waste time keying in all the junk repeatedly!

### 3.3.1 Code Layout

Many people have quite strong views about code layout and indentation. That includes me! The style you will see in these notes almost always places any “}” vertically below the “{” that it matches. I try to indent anything that is within such braces by four space positions. Beyond that my guiding principle is to try to keep my code so that it looks pretty on the screen or page, is efficient in its use of the page and is as easy to navigate over as I can manage. Saving keyboard effort is not a high priority, since actually typing in programs is such a very small part of the total pain that goes into getting a complete and robust working program. The default `emacs` idea about indentation and brace layout is differs from mine: whichever you choose to follow please be consistent and try to make your code easy for yourself and others to read.

The comment above about efficiency in the use of the page is because when reading your code it is especially convenient if all the bits you want to see fit within one screen-full of the editor’s window. Thus I count excessive splitting of constructs over multiple lines as unhelpful, just as are large swathes of blank lines. I prefer comments in blocks (which may often make up significant paragraphs) that describe the code that follows them. And the comments should be readable English in proper sentences intended to help some poor person faced with revising or updating the code to correct some imaginary bug or add a new feature.

Java provides some encouragement for special comments that are introduced with the sequence “`/*`”<sup>8</sup> and going on over possibly many lines until the next “`*/`”. These are there to support extra software tools that extract those comments and format them as separate documentation for the program. In this course I will illustrate that scheme later on.

Well all the above discussion has just left us with a file `Hello.java`. Unlike (typical teaching use of) ML, Java expects programs to be processed by a separate compiler before they are executed. This compiler is a program that checks the syntax and types of the code you show to it, and translates from the human-

---

<sup>8</sup>Ordinarily as well as “`/**`” comments that just run to the end of the line you can write long comments starting with “`/*`”.

readable<sup>9</sup> source file such as `Hello.java` into a more compact<sup>10</sup> digested binary file (called `Hello.class` in this case) that can subsequently be executed repeatedly without any need to re-do all the potentially time-consuming checking. To carry out this conversion you need to say

```
javac Hello.java
```

The `javac` command tends to be jolly taciturn unless it finds something in your program that offends it. It does not say anything and so after it has run you may like to use `ls` to verify that the file `Hello.class` has been created. Finally we can run it:

```
java Hello
```

Note that when `javac` was used to compile the program it was essential to quote the `.java` extension, while when the program was to be run you must not use the `.class` extension that the pre-digested version of the program was given. This level of apparent inconsistency is not at all restricted to Java, and the exact rules on matters such as this are liable to differ between different vendor's sets of Java tools. What I describe here relates just to Sun's SDK!

### 3.3.2 Emacs

The editor `emacs` is the preferred text editor to use while taking this course. I think it may be best for most people to start by keeping two windows available on their screens, one the `emacs` edit window and the second a command-prompt from which they can issue the build and run commands directly. When working with an edit and a command window note that you have to go "File/Save Buffer"<sup>11</sup> to get `emacs` to ensure that the file on disc is brought up to date with respect to the version you have been editing in its buffer. Provided you do this before issuing the `javac` command from your other window it is reasonable and most convenient to keep `emacs` loaded throughout your session. It is also possible to compile and run Java (or other) programs while remaining entirely within `emacs`, and to get any reports of syntax errors generated by a compiler to re-position the editor's caret close to where the error was detected. But for the rather small programs you will be working with during this Part IA course all is excessive and using one window to edit and one to compile as in Figure 3.5 remains simplest.

The next program to be shown is a rather simple extension of the one we have already discussed, but instead of just printing a fixed message it prints a table of squares. In a file called `Squares.java` you should place:

---

<sup>9</sup>Well, at least it is readable if you include enough comments!

<sup>10</sup>Actually for really tiny programs like this one the binary file may be bigger than the source it relates to, but for and program big enough to be interesting what I say will hold true.

<sup>11</sup>Or the equivalent keyboard sequence, `Ctrl-x Ctrl-s`.

```
public class Squares
{
    public static void main(String[] args)
    {
        for (int i=0; i<10; i++)
        {
            System.out.printf("The square of %d is %d\n",
                i, i*i);
        }
    }
}
```

There are two new things here. The first is the iteration statement

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

which arranges to declare a variable called `i` and set it first to 0, then to 1, then 2 and so on for so long as `i<10` remains true. The curious syntax `i++` is inherited from the C programming language and means “increment `i`”: a less cryptic way of achieving the same effect would be to write “`i=i+1`” instead. The single `=` sign in Java is an assignment operator and changes the value of the variable named on its left. The word `int` is short for “integer” and specifies the type that `i` should have. Type Java type `int` denotes integers which are explicitly limited to a range that is consistent with representation as 32-bit values. Unlike ML Java expects you to specify the type of pretty well everything you mention, and when you introduce a new variable you can change its value later using a `=` operation without having to worry about any special extra works like `ref`.

The second new feature is the string argument to `printf` where emdedded percent signs stand for where the numeric values you want displayed need to be substituted in. The `%d` indicates that what you want displayed is expected to be an integer: other letters could be used when you were needing to print other sorts of item. Once again I need to make a remark this year that is to do with the transition to Java 1.5: in previous years and in many books you will see this code written as

```
System.out.println("The square of " + i + " is " + (i*i));
```

where the plus signs in fact indicate string concatenation and Java is converting integers to printable form fairly automatically. I prefer the use of `printf` because the `%d` indicates very explicitly that I am about to print an integer (not some other sort of thing). It can also be extended to give me quite refined control over the layout of the table I generate.

Note that when I came to want to type in the Squares program to check it I did not type it in from scratch. Instead a copied the earlier Hello program and adjusted the few lines in its middle to perform the new operations. Typically it will also be necessary to change a few comments to make them relate to the new reality, but creating new code by making incremental extensions to old is a very useful

technique and can save a lot of time and effort. It also means that remembering all those boring bits is at least slightly less necessary.

One further development of the Squares example will illustrate a few more Java idioms. This code (which I will put in a file `Powers.java`) computes powers and does so by a repeated-squaring technique that may be familiar from the ML-based course last term:

```
public class Powers
{
    public static void main(String[] args)
    {
        // I will use println for simple fixed text
        System.out.println("Table of powers");
        for (int i=0; i<10; i++)
        // .. and printf to incorporate values within a template
        {   System.out.printf("%d^%d = %d%n", i, i,
                               power(i, i));
        }
    }
    static int power(int x, int n)
    {   if (n == 0) return 1;
        int y = power(x, n/2);
        if ((n % 2) != 0) return x*y*y;
        else return y*y;
    }
}
```

which produces the results

```
Table of powers
0^0 = 1
1^1 = 1
2^2 = 4
3^3 = 27
4^4 = 256
5^5 = 3125
6^6 = 46656
7^7 = 823543
8^8 = 16777216
9^9 = 387420489
```

The new features shown here are the definition of a function and calls to it. Observe that the types of the arguments for the function and the type of its result are all explicitly given (as `int` here). The code does distinctly more arithmetic,

where `+`, `-`, `*` and `/` stand for addition, subtraction, multiplication and division. The percent sign `%` gives a remainder. Numeric comparisons are written with `>`, `<`, `>=` and `<=` for the obvious comparisons, and the rather less obvious `==` for an equality test and `!=` for inequality.

Conditional statements appear as

```
if (condition) statement
```

or

```
if (condition) statement
else statement
```

Note that the parentheses around the condition are part of the Java syntax (inherited from C) and they may not be omitted.

You need to use the word `return` explicitly to indicate what value your procedure should hand back.

It is a very common beginner's error to get mixed up about where braces and semicolons are needed — and mix-ups on this front can cause special trouble with the `else` after an `if` statement. In doubt just remember that you can group several statements (or indeed just one) together to make a single big statement just by enclosing them (or it) in braces “`{ ... }`”. The braces I have around the call to `printf` just after the `for` were put in not because they are essential (the call to `printf` counts as a single statement and could be the thing that the `for` loop performed in a repetitive way) but because I think the braces there make it easier to see just what the range of the `for` is. Similarly it is often good style to use braces that are in some sense redundant after the keyword `if` just to ensure that the structure of your code is utterly evident to any reader.

The series of small examples above show enough of Java that they can form the basis for exercises that use integer arithmetic and a few recursive sub-functions. With luck they contain enough examples of usage that you can now go away and write all sorts of little programs that perform calculations with at most minor recourse to the textbook to check exact details.

### 3.3.3 Drawing to a window: JApplets

I will therefore move onto another cook-book example which shows a different sort of Java program. The ones seen so far are referred to as stand-alone applications. The next one will be described as an “applet”. It has an even higher load of mumbo-jumbo to surround the small bits that are its essential core, but illustrates how you can start to use Java for graphics programming and to interact with windows, mice and the like. As with my Hello program I will start by quoting

The screenshot shows the Emacs editor window titled 'emacs@PANAMINT'. The main window displays a Java program named 'Powers.java' with syntax highlighting. The code is as follows:

```
// powers program for
public class Powers
{
    public static void
    {
        System.out.pri
        for (int i=0;
        { System.out
        pc
    }
}

// sub-function to compute x to the power n
static int power(int x, int n)
{
    if (n == 0) return 1;
    int y = power(x, n/2);
    int z = y*y;
    if ((n % 2)OOPS != 0) z = z*x;
    return z;
}
}

// end of Powers.java
```

The code is color-coded: comments are grey, keywords are red, class names are green, and identifiers are blue. A 'Customize' menu is open, showing the 'Options' section with the 'Global Font Lock (highlights syntax)' option checked. Below the code, the terminal output shows the compilation process and an error:

```
--\-- Powers.java (Java)--L18--All-----
cd d:/univ/notes/java/
javac Powers.java
Powers.java:18: ')' expected
    if ((n % 2)OOPS != 0) z = z*x;
                ^
Powers.java:18: incompatible types
found   : int
required: boolean
    if ((n % 2)OOPS != 0) z = z*x;
                ^
2 errors

--\** *compilation* (Compilation:exit [1])--L1--Top-
```

Figure 3.7: emacs on Windows, with the “Global Font Lock” option for syntax colouring.

the important bit of the code that lives in the middle. In this case it will arrange to keep track of where your mouse last was when you pressed its button, and will respond to new mouse clicks by drawing a straight line on the screen to join the old to new position.

Since at this stage I want to make this key part of the code look as short and easy as possible I have omitted any comments — after all I am about to give an explanation here in the accompanying text!

In the Hello program we used a function called `printf` by referencing it relative to some library object `System.out`. Here we need to suspend disbelief for a short while and imagine two things, one called `e` that allows us to call library functions that reveal the position of the mouse (`getX` and `getY`) and another called `g` that is analogous to `System.out` but which supports a function `drawLine` for putting a straight line up on the screen. Suppose furthermore that there are integer variables `lastX` and `lastY` that will be used to store the previous position where the mouse was clicked. It now makes sense to show the kernel of the drawing program:

```
int x = e.getX(), y = e.getY();
g.drawLine(lastX, lastY, x, y);
lastX = x;
lastY = y;
```

Look under the link [Java Platform Core API](#) on the web-browsable documentation. Clicking at the top of the screen through `Index` makes it almost as quick to look up `getX`, `getY` and `drawLine` as it would be to check for them in the index of a book. In either case you are liable to find near their documentation the explanation of other related functions, such as `drawRect`, `drawOval`, `fillArc`, `drawString` and many many more.

Once one has sorted out how to use one of these in general the rest follow on naturally, so it can be useful to browse the documentation occasionally to make yourself familiar with the collection of operations that are supported.

The next natural question is one of where the mysterious `e` and `g` came from, and how it could be arranged that the above code is activated every time the mouse button is pressed. Well just as a simple stand-alone application has a special function called `main`, one that deals with the mouse will have one called `mousePressed`. This gets the object `e` passed down to it from the system. all one needs to know is that the type used to declare this variable is `MouseEvent`. Access to the screen is obtained by declaring `g` to be of type `Graphics` and initialising it with the value returned by a call to `getGraphics`<sup>12</sup>. These types and conventions are to some extent part of a large design that underlies the Java libraries, but at this stage the only proper way to cope with them is to copy them carefully from existing working programs and check details in the documentation. When you look at the documentation I expect your main initial response to be one close to “Wow” as you see just how many types and functions Java provides you with. Overall there is more complexity and power in these libraries than there is in the language itself. Anyway here is the full version of the mouse click handler function — not too messy provided one is happy to take the library calls on trust!

---

<sup>12</sup>In this initial example I use `getGraphics` but often the object you want will come to you in other ways.

```

public void mousePressed(MouseEvent e)
{
    // I have to obtain access to a drawing context
    Graphics g = getGraphics();
    // I also need to extract (x,y) co-ordinates from
    // the mouse event.
    int x = e.getX(), y = e.getY();
    g.drawLine(lastX, lastY, x, y);
    lastX = x;
    lastY = y;
}

```

I can now give the whole file `Draw.java` which includes the above important function definition, but which also has the relevant junk that is needed to connect it in to the Java run-time environment. You will see that I have this time used comments from `/*` to `*/` for some of the big block comments. The arrangement with columns of vertical stars is purely a convention that I like and which makes the range of the comment clearly visible. The lines starting `import` arrange for convenient access to several extra Java libraries. You will find `import` statements at the top of most of my sample programs from now on and the exact list of things you need to “import” will seem jolly mysterious. All I can say at this stage is that you can start by copying the lines I give and that in a week or so you will understand how to check the Java on-line documentation to sort out exactly what you need exactly when.

The qualifications (`extends` and `implements`) on the declaration of the `Draw` class ensure that this program can draw to the screen and respond to the mouse. When a file contains a class that extends `JApplet` the rules for it starting up are not like ordinary programs. Instead of defining `main` it defines the functions shown here.

```

/*
 * Draw.java                                A C Norman
 *
 *      Simple applet to draw lines on a screen
 *      in response to mouse clicks. See also "Draw.html".
 */

/*
 * At the start of almost any Java program it will
 * be necessary to incant a few "import" statements to
 * provide Java with more convenient access to various
 * standard libraries.
 */

```



```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Draw extends JApplet
    implements MouseListener
{
    private int lastX = 0, lastY = 0;

    public void init()
    {
        // I need to activate the mouse event handlers.
        this.addMouseListener(this);
    }

    /*
    * Each time the mouse button is pressed I will draw a
    * line on the screen from the previous mouse position
    * (or (0,0) at the start) to where the mouse now is.
    */
    public void mousePressed(MouseEvent e)
    {
        // I have to obtain access to a drawing context
        Graphics g = getGraphics();
        // I also need to extract (x,y) co-ordinates
        // from the mouse event.
        int x = e.getX(), y = e.getY();
        g.drawLine(lastX, lastY, x, y);
        lastX = x;
        lastY = y;
    }

    /*
    * The full mouse event model uses the four extra
    * procedures shown below. To keep this code as short
    * and simple as I can I will not cause them to do
    * anything, but the Java event handler scheme demands
    * that they exist. Hence these definitions of functions
    * that do nothing at all!
    */
    public void mouseReleased(MouseEvent e) {}
    public void mouseClicked(MouseEvent e) {}
}
```

```

    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}

/* end of Draw.java */

```

Without a prototype such as the above to start from it could take a huge amount of reading of the manuals to find all of the Java types and functions to put together. However once you have the prototype to work from there is at least some chance that variations on the theme can be constructed by making incremental changes, and the details of these changes can be sorted out by looking in the manuals close to the place where the features that are currently used get documented. The version I have given here uses the class `JApplet` while you may find some books use `Applet` (without the initial letter J) that is an older version of something similar.

### 3.3.4 HTML and appletviewer

The earlier examples were run using commands such as `java Hello`. This one is not a stand-alone application but a `JApplet`, and so has to be run using a thing called `appletviewer`. What is more it needs yet another file to be prepared: one that will let it know how large an area of the screen should be set aside for the drawing to appear in. This new file must be called `Draw.html`, and its contents are as follows:

```

<HTML>
  <BODY>
    <APPLET code="Draw.class" width=400 height=400>
      Java is not available.
    </APPLET>
  </BODY>
</HTML>

```

This file consists of a set of nested sections, where the start of a section is a word contained in angle brackets and the corresponding end-marker is the same word but with “/” in front of it. Once again the most interesting part is in the middle where the `APPLET` tag is used to provide a reference to the compiled version of our program (ie `Draw.class`) and to specify the width and height of the window in which it is to work. The text “Java is not available” should never appear when you use this file! It is there so that it can be displayed as an error message if this `HTML`<sup>13</sup> file is inspected using software that does not understand Java. For the purposes of this course the only use you will make of the file is to say

---

<sup>13</sup>Hypertext Mark-Up Language.

```
appletviewer Draw.html
```

which should cause a 400 by 400 (pixel) window to appear within which you can click the mouse to good effect. The window that `appletviewer` should provide a pull-down menu that contains an entry `quit` or `close` than can be used to terminate the program. Observe (with quiet gloom) that `appletviewer` demands that you quote the `.html` suffix, and that inside the HTML file you have to specify the full name of your class file (ie including the `.class` suffix), while to run simple stand-alone Java applications you just gave the base part of the file-name. Ah well!

The Draw program shown here is a useful prototype, but the most glaring problem it exhibits is that if you rearrange your windows while using it so as to obscure part of what you have drawn then that bit does not get re-painted when you reveal the window again.

When you move towards larger programs (ones spread over very many files) you will probably need to read up about a tool called `jar` and find out (it is easy!) how you can package many Java class files into a single archive, and how HTML files refer to such archives. I will not explain that in this Part IA course.

### 3.3.5 Exercises

#### Tickable Exercise 2

Do both parts A and B please.

##### Part A

The following Java function is a rather crude one for finding a factor of a number  $n$  and returning it, or returning 1 if the number is prime.

```
static int factorof(int n)
{   int factor = 1;
    for (int i=2; i*i<=n; i++)
    {   if (n % i == 0) factor = i;
        }
    return factor;
}
```

The code works by checking each possible factor from 2 upwards, stopping when the trial factor exceeds the square root of the number being tested. This stopping condition is reasonable because if a number  $n$  is not prime than it must have at least one factor in the range  $2 \dots \sqrt{n}$ .

Write a stand-alone Java program that incorporates the above code and that prints out a list of all the prime numbers from 2 to 100.

*Optional:* Change `factorof` to make it more efficient by first letting it check whether 2, 3 or 5 is a factor and then instead of trying *all* possible factor up to the square root of `n` let it just try those that are 1, 7, 11, 13, 17, 19, 23 or 29 mod 30. It do not bother with numbers that are themselves divisible by 2, 3 or 5. This should lead to making just  $8/30 = 26.66\%$  of the number of test divisions that the original version did. Does the new version run faster, and if so by about what factor?

#### Part B: Binomial Coefficients

This exercise is a deliberate incitement to write a very inefficient program. Later on there will be an example that prompts you to write a much faster program that can compute the same answers! The binomial coefficients<sup>14</sup> may be defined by the rules

$$\begin{aligned} {}^nC_r &= {}^{n-1}C_r + {}^{n-1}C_{r-1} \\ {}^nC_0 &= 1 \\ {}^nC_n &= 1 \end{aligned}$$

This definition could naturally turn into an ML function definition

```
fun binom(n, r) =
  if r = 0 orelse r = n then 1
  else binom(n-1, r-1) + binom(n-1, r);
```

Write the corresponding Java code and use it to tabulate  ${}^{2^n}C_n$  for  $n$  from 0 to 12.

If you are using Java on Unix you should go “time java Binom” to run the example and when your program has run you will get a report of how long the computation took. Keep all the output so you can compare both results and timing data with the method described later on.

*(End of tickable exercise)*

---

<sup>14</sup>There is some question as to whether I should use the notation  ${}^nC_r$  here or  $\binom{n}{r}$ . I hope this will not confuse you too much!

### A better drawing program

The drawing program as presented is very clumsy. There are a number of ways it could be improved. The suggestions made here are not the correct route towards a properly finished professional quality drawing program but may still count as useful practise with Java.

1. In the 400 by 100 window, interpret mouse clicks in the top 50 pixels as button activity that can select options. The x co-ordinate may be split into (say) 4 ranges to give four buttons. In `mousePressed` add:

```
if (y < 50)
{
  if (x < 100) .. action1
  else if (x < 200) .. action2
  else if (x < 300) .. action3
  else .. action4
}
else
{
  normal mouse processing
}
```

2. The crude buttons as above could select whether further regular mouse clicks drew lines (as before) or used `drawOval` to draw circles. Another button might select a drawing colour using

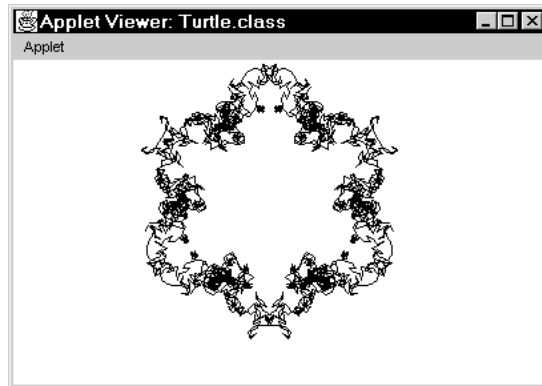
```
g.setColor(Color.blue); // or red, black etc
```

3. My code, which was trying to be as short as possible, did not treat the first mouse click specially, and so all trails started at (0,0). That should be changed.
4. `drawString(string, x, y)` places text in a window at the given position. It could be used to label the “buttons”.

I think that the code that you could potentially achieve here would be pretty good for this stage in the course!

## Turtle Graphics

The following code shows some more new features of Java. It defines a `paint` method (ie function) in an applet. The appletviewer arranges that this function is invoked every time the applet's window is uncovered or otherwise needs re-drawing, and so it leads to pictures that are a lot more robust than the mouse-driven Draw program shown earlier. The code also uses a new type, `double` which is for floating point numbers, and some calls to the



Maths library to compute sines and cosines. The odd notation `(int)x` indicates that the code wants to convert the floating point value `x` into an integer `(int)` so it is of the correct type to be passed on to `drawLine`.

Put the code in a file `Turtle.java` and prepare a suitable associated file `Turtle.html`. Experiment with the code and see how the image changes depending on the values of the three variables marked. For most values of `inc` I seem to find that a closed figure is drawn provided `N` is large enough, but I have some trouble producing an explanation of why or a characterisation of exactly what values of `inc` will lead to this behaviour. I also find the degree of symmetry hard to explain. Generally this is an illustration of the fact that quite short programs can have behaviour that is complicated to explain!

```

/*
 * Turtle.java                                A C Norman
 * illustration of Turtle Graphics and the "paint" method.
 */

import javax.swing.*;
import java.awt.*;
import static java.lang.Math.*;

public class Turtle extends JApplet
{
    public void paint(Graphics g)
    { // Try changing the following 3 numbers...
      double size = 5.0, inc = 11.0;
      int N = 5000;
      double x = 200.0, y=200.0,

```

```

        th1 = 0.0, th2 = 0.0, th3 = 0.0;
    for (int i=0; i<N; i++)
    {
        th3 = th3 + inc;
        th2 = th2 + th3;
        th1 = th1 + th2;
        double x1 = x+size*cos(PI*th1/180.0);
        double y1 = y+size*sin(PI*th1/180.0);
        g.drawLine((int)x, (int)y, (int)x1, (int)y1);
        x = x1;
        y = y1;
    }
}
/* end of Turtle.java */

```

The code is really using angles in degrees (not in radians), and the variables `th1`, `th2` and `th3` hold values that are angles. As coded above some of these angles can grow to ridiculously large values, it might make sense to insert lines based on the prototype

```

    if (th2 >= 180.0) th2 = th2 - 360.0;

```

in suitable places with a view to keeping all the angles that are used in the range  $-180.0$  to  $+180.0$ .

The `import static java.lang.Math.*` line makes it possible to use `sin`, `cos` and `PI` in the simple way shown.

Note that in Java (and indeed with many window systems) the y co-ordinate starts at 0 at the top of the screen and increases as you go down. This makes sense (sort of) when the screen is containing text, in that counting lines you normally start at the top. For pictures it can be a little muddling until you are used to it, and can mean that things sometimes come out upside down the first time you try them.





# Chapter 4

## Basic use of Java

### 4.1 Data types, constants and operations

The first section of these notes introduced a few small but complete Java programs, but when you type them into the computer you still have to take a great deal on trust. But with those examples to use as a framework I can now start a more systematic introduction of the Java language and its associated libraries. Actually in the lectures I expect to skim over this material very rapidly: you will in reality learn about the Java data types and syntax as you write programs. However I view it as important that you have reference material in your handout that shows what everything is so that if you have trouble in your coding you have somewhere reasonably close at hand where you can check some details. However if you need a gentle path to settle into Java programming I do suggest that you try various of the example programs and exercises here so that you get used to and comfortable with a good range of Java syntax.

#### 4.1.1 Reserved Words

The first thing to do is to catalogue the words that are reserved: if you accidentally try to use one of these names as the name of one of your variables or functions you can expect most curious error messages from Java! So even though I do not want to explain what all of these mean yet it may help you if I provide a list of words to be avoided. In some cases of course the presence of a word here will alert you to the availability of some operation, and you can then look up the details in the manual. A clever editor might display words from this list in some alternative colour to help you notice any unintentional uses. An even more clever one might use different colours for the one (such as `int`) that name basic types, the ones such as `for` that introduce syntax and ones like `true` that are just the names of



Figure 4.1: Start with some small examples...

important built-in constants.

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	false	final	finally
float	for	goto	if	implements
import	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	short	static
strictfp	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while		

A joke about the above table of reserved words is that at present Java does not actually use them all — specifically `const` and `goto` do not represent any operation supported by Java. By prohibiting people from using these words as names for variables the Java designers have left themselves a little flexibility to extend the language in the future if they want to or are forced to, and they can perhaps give better error messages when people who are used to some other language that does include these keywords first tries out Java.

There are some other words which are used for names that perform important system functions. If you unintentionally define a function with one of these names you might find that you have introduced side-effects of amazing magnitude when the system calls your function instead of the one it was expecting! Beware here, because although incorrect use of a genuine reserved word will result in a syntax error it could be that defining a function with one of the following names would have subtle or delayed bad consequences rather than a nice clean instant crash.

<code>clone</code>	<code>equals</code>	<code>finalize</code>	<code>getClass</code>	<code>hashCode</code>
<code>notify</code>	<code>notifyAll</code>	<code>toString</code>	<code>wait</code>	

OK so the above information is more negative than positive, but I hope it will rescue a few of you from otherwise most mysterious behaviour when you might otherwise have tried to use one of the reserved words for your own purposes.

## 4.1.2 Basic Types

Earlier examples used the word `int` to declare integer variables, and the range of values that can be represented goes from around  $-2$  billion to around  $+2$  billion. To be more precise the smallest valid `int` is  $-2^{31} = -2147483648$  and the largest one is  $2^{31} - 1 = 2147483647$ . You are not expected to remember the decimal form of the numbers, but you should be aware of roughly how big they are. Integer

overflow is ignored: the result of an addition, subtraction or multiplication will always be just what you would get by representing the true result as a binary number and just keeping the lowest 32 bits. A way to see the consequences of this is to change the `Powers` program so it goes up to higher powers, say 20. The final section of the output I get is

```

...
8^8 = 16777216
9^9 = 387420489
10^10 = 1410065408
11^11 = 1843829075
12^12 = -251658240
13^13 = -1692154371
14^14 = -1282129920
15^15 = 1500973039
16^16 = 0
17^17 = 1681328401
18^18 = 457441280
19^19 = -306639989

```

where the value shown for  $10^{10}$  is clearly wrong and where we subsequently get values that probably count as rubbish. Note both the fact that overflow can turn positive values into negative ones (and vice versa) and the special case (obvious in retrospect) where  $16^{16}$  shows up as zero. Since 16 is  $2^4$  the binary representation of  $16^{16}$  is clearly a 1 followed by a string of 64 zeros, and in particular the least significant 32 bits are all zero. This lack of detection of integer overflow is sometimes convenient but it is also a potential major source for getting wrong answers without even knowing it.

Java provides several alternative integer-style primitive data-types which represent different trade-offs between expected speed, space and accuracy. They are:

**byte:** 8-bit integers in the range  $-128$  to  $+127$ ;

**short:** 16-bit integer, range  $-2^{15} = -32768$  to  $2^{15} - 1 = 32767$ ;

**int:** 32-bit integers as discussed already;

**long:** 64-bit integers, is range is from  $-2^{63}$  to  $2^{63} - 1$  which means that almost all numbers with up to 19 decimal digits can be represented.

It may be helpful to those who are not already used to the binary representation of signed values if I tabulate the representation used for the `byte` datatype. The wider integral types use just the natural generalisation:

Number	Representation in binary							
	$-2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
127	0	1	1	1	1	1	1	1
126	0	1	1	1	1	1	1	0
...								
3	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0
-1	1	1	1	1	1	1	1	1
-2	1	1	1	1	1	1	1	0
-3	1	1	1	1	1	1	0	1
-4	1	1	1	1	1	1	0	0
...								
-126	1	0	0	0	0	0	1	0
-127	1	0	0	0	0	0	0	1
-128	1	0	0	0	0	0	0	0

One way to understand how the negative numbers arose is to see that  $-1$  is the bit-pattern that has the property that if you add 1 to it and ignore the final carry you get the representation that means 0. It can also help to suppose that negative number “really” have an infinite string of 1 bits glued onto their left hand end. The representation used is known as “two’s complement”.

When you write an integer literal in your Java code you can write it in decimal, octal or hexadecimal (base-16). You can also make your written integer either of type `int` or type `long`; there is no direct way to write either a `byte` or `short`. Decimal numbers are written in the utterly ordinary way you would expect. You add a suffix “L” if you want to make the value a `long` and you should always do this if the value is outside the range of ordinary available to an `int`, but you might sometimes like to do it for even small values when using them in a context where arithmetic on them should be done in `long` precision. Examples are:

```

12345           an ordinary int
1234567890123L  a long value
10L            a long but with a smallish value
1000000L*1000000L  an expression where the L suffix
                  matters
1000000*1000000  without the L this would overflow.
```

My belief is that hardly anybody ever wants to write a number in octal these days<sup>1</sup>, but Java allows it, taking any number that starts with 0 as being in octal.

<sup>1</sup>But this may be just a matter of fashion, and perhaps elsewhere in the world octal is still appreciated.

Thus `037` is the octal way of writing the number 31. The `L` suffix can be used to specify long octal values. Observe a slight jollity. If you write the number `0` it is interpreted as being in octal. Fortunately zero is zero whatever radix is used to write it!

Hexadecimal is much more useful. Each hexadecimal digit stands for four bits in the number. Letters from `A` to `F` are used to stand for the digits with weight `10...15`. Hexadecimal numbers are written with the prefix `0X`. Note that the suffix `L` for `long`, the `X` in hexadecimal numbers and the extended digits from `A` to `F` can all be written in either upper or lower case. I strongly recommend use of upper case for `L` since otherwise it is painfully easy for a casual reader to muddle `101` (long ten) and `101` (one hundred and one).

Here are some numbers written in hexadecimal

```

0X0           this is zero, not gravy powder
0xe          otherwise 14
0xffffffff   -1 as an int
0xBadFace    what other words can you spell?
0x7fffffff   largest int
0x80000000   most negative int
0x00010000   2 to the power 16
0x7fffffffffffffffffL largest long

```

I rather suspect that the main importance of `byte` and `short` is for when you have huge blocks of them<sup>2</sup> where the fact that they take up less space can be of practical value.

Floating point values also come in two flavours, one with a larger range and precision than the other. The more restricted one is called `float`. A `float` uses 32-bits of memory and can represent values up to about  $3.4e38^3$  with a precision of six to seven significant figures. Until you have sat through the course on numerical analysis please avoid use of it<sup>4</sup>. The more sensible floating point type is called `double` and uses 64 bits to store numbers with magnitude up to about  $1.7e308$ , with an accuracy of sixteen or seventeen significant figures. The internal representation of floating point values and the exact behaviour in all circumstances was originally taken from an International Standard referred to as IEEE 754. Some bit-patterns are reserved to represent “ $+\infty$ ” and “ $-\infty$ ” while others are values that are explicitly not representations of valid floating point values — these are known as NaNs (Not A Number). A few possibly unexpected effects arise from this. For

<sup>2</sup>See the description later on of arrays.

<sup>3</sup>ie  $3.4 \times 10^{38}$ .

<sup>4</sup>In fact a number of the Java library functions require arguments of type `float`, so it is not possible to avoid this type. Its use is satisfactory in circumstances where the precision it supports is all that is justifiable, for instance when specifying the brightness of a colour.

instance floating point division never fails: `0.0/0.0` yields a NaN, while any other value divided by `0.0` results in an infinity. Also if `u` is a floating point value then it is possible for the expression `u == u` to evaluate to `false` (!) because the rules for all numeric comparison operators is that they return `false` when at least one of their arguments is a NaN. Another oddity is that one can have two floating point values `u` and `v` such that both `u == v` and `(1.0/u) != (1.0/v)`! This oddity is achieved by having `u = -0.0` and `v = +0.0`. These delicacies are almost certainly exhibited by most other languages you will come across, but Java documents them carefully since it is very keen indeed to make sure that Java program will give exactly the same results whatever computer it is run on. Even if it does very delicate things with the marginal and curious cases of floating point arithmetic. Recent versions of the Java language use a keyword `strictfp` to indicate places where *all* the consequences of IEEE floating point must be honoured: specifically its use means that the results computed should be identical whatever machine run on, and will have rounding errors exactly as expected. Without `strictfp` and on some computers Java will deliver results that are both more accurate and are computed faster!

Here are some floating point constants:

<code>0.0</code>	this is double (by default)
<code>0.0F</code>	if you REALLY want a float
<code>1.3e-11</code>	specify an exponent (double)
<code>22.9e11F</code>	float not double
<code>22.9e11D</code>	be explicit that a double is required
<code>1e1</code>	no "." needed if there is an "e"
<code>2.</code>	"." can come at end
<code>.2</code>	"." can come at start
<code>2D</code>	must be a double because of the D

I would suggest that you always make your floating point constants start with a digit and contain a decimal point with at least one digit after it since I think that makes things more readable.

In Java the result of a comparison is of type `boolean`, and the boolean constants are `true` and `false`. As in ML (and unlike the situation in C, in case you know about that), `boolean` is a quite separate type from `int`.

Despite the fact that this section is about the Java primitive types and not about the operations that can be performed on data, it will make some of my examples easier if I next mention the ways in which Java can convert from one type to another. In some cases where no information will be lost (eg converting from a `byte` or `short` to an `int`) the conversion will often happen without you having to worry much about it and without anything special having to be written. However the general construction for making a type conversion is called a *cast*, and it is written by using a type name in parentheses as an operator. We have already

already seen a couple of examples in the `Draw` program where `(int)x` was used to convert the floating point value `x` into an integer. The opposite conversion can then of course be written as in

```
for (int i=1; i<10; i++)
    System.out.printf("%22.8g%n", 1.0/(double)i);
```

where the `(double)` is a cast to convert `i` to floating point<sup>5</sup>. The format specifier<sup>6</sup> `%22.8g` is for printing a floating point value in a General format using a precision of 8 significant figures and padding with blanks to make 22 characters printed in all. Until you understand exactly when automatic conversions apply it may be safest to be explicit. Java allows you to write casts for those conversions that it thinks are sufficiently reasonable. You can cast between any of the flavours of integer. When you promote from a narrower integer to a wider one the value is always preserved. When you cast from a wider integer to a narrower one the result is what you get from considering the binary representation of the values concerned, and the cast just throws away unwanted high-order bits. Casts from integers to `float` and `double` preserve value as best they can<sup>7</sup>. Casts from floating point values to integers turn NaNs into 0, and infinities into either the most positive or most negative integer. Floating point values that are too large to be an `int` or `long` also turn into the largest available integer. The exact rules for casts from floating point values to `byte` and `short` are something to look up in the reference manual in the improbable case it matters to you. There are no casts between `boolean` and other types. You need to use explicit expressions such as `(i != 0)` to map from an integer to a truth-value<sup>8</sup>.

The type `char` can be used to declare a variable that holds a single character. To write a constant suitable for putting into such a variable you just write the relevant character within single quote marks, as in

```
char mychar = 'A';
if (myChar == 'q') ...
```

It is frequently necessary to use characters that do not fit so neatly or clearly between quotes. For instance the single quote character itself, or a “character” to represent the end of a line. A set of escape codes are used for these, where instead of a single character one writes a short sequence starting with the escape character “\”. The supported escape sequences are:

---

<sup>5</sup>In this case the cast is not needed: Java will do the required conversion so that it can perform the division.

<sup>6</sup>You will see a bunch of common format specifiers just in examples here. You can look up full details in the on-line documentation, or find a medium-sized synopsis later in these notes.

<sup>7</sup>Casts from `int` or `long` to `float` or from `long` to `double` can not always preserve an exact result because the floating point format may not have enough precision available. The closest floating point value to the true result will be delivered.

<sup>8</sup>Unlike the position in C where there is not much distinction between integers and booleans.



<code>\n</code>	newline, linefeed (very commonly used)
<code>\"</code>	double quote mark
<code>\'</code>	single quote mark
<code>\\</code>	use when a single <code>\</code> is wanted
<code>\b</code>	backspace
<code>\t</code>	tab
<code>\f</code>	newpage, formfeed (unusual)
<code>\r</code>	carriage-return

Carriage returns are used in Windows text files and as line separation in some Internet protocols, but when creating simple text files you do not generally see or need to mention it: Java does any necessary conversions to you so that on Windows, Macintosh and Unix an end of line in a file is talked about in your programs as just `'\n'`.

In addition it is possible to write `\nnn` where *nnn* stands for 1, 2 or 3 octal digits: this indicates the character with the specified character-code in a standard encoding. Use of octal escapes is not at all common. Furthermore Java allows inclusion of characters from an astonishingly large character set by use of a notation `\u` followed by four hexadecimal digits. The 16-bit number represented by the hexadecimal digits is taken as being in a set of character encodings known as Unicode. Casts between `int` and `char` give direct access to this encoding. For example `\u2297` and `(char)0x2297` both give the character “⊗”. In fact the Unicode escapes do not just apply within Java character literals but can be used **anywhere** in a Java program where you want an unusual symbol — and this means that in some sense you can have variables names with Greek, Russian and Eastern glyphs in them. Unicode gradually becoming more widely used, but most computers still do not have full Unicode fonts installed, and so exotic characters will not always be displayed properly even though within Java they are handled carefully. The following applet displays the characters that are available using the viewer it is run under. It uses a cast `(char)` to convert an integer to a character and some fresh library calls (eg `setFont(new Font(...))` and `drawString`). It also illustrate something that you will probably want to retrofit to most of the little examples in these notes. It allocates a `BufferedImage` that it draws into, and then the `paint` method just displays whatever is in the bitmap. This does wonders for arranging that when you obscure bits of your window the content gets re-painted nicely!

It also makes a crude modification of the earlier `Draw` program so that mouse clicks at various places in the window adjust the range of characters displayed.

```
/*
 * Unicode.java                                     A C Norman
 *
```

```
* Display the Unicode characters as supported
* by the current browser.
*/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.image.*;

public class Unicode extends JApplet
    implements MouseListener
{
    private boolean isFilled = false;
    private int fontSize = 20; // or whatever!
    private int page = 0;
    private BufferedImage p =
        new BufferedImage(
            32*fontSize,
            35*fontSize,
            BufferedImage.TYPE_BYTE_BINARY);

    public void init()
    {
        addMouseListener(this);
    }

    public void mousePressed(MouseEvent e)
    {
        if (e.getX() < 200) page++;
        else page--;
        if (page > 63) page = 0;
        if (page < 0) page = 63;
        isFilled = false;
        repaint(); // force screen to re-draw
    }

    public void paint(Graphics g)
    {
        if (!isFilled) fillImage();
// Note drawImage may need repeating!!!
        while (!g.drawImage(p, 0, 0, this));
    }
}
```

```

void fillImage()
{
    Graphics g = p.getGraphics();
    g.setColor(Color.WHITE); // background
    g.fillRect(0, 0, 32*fontSize, 35*fontSize);
    g.setFont(new Font("Serif",
                      Font.PLAIN, fontSize));
    g.setColor(Color.BLACK); // text
    g.drawString("page = " +
                Integer.toHexString(32*32*page),
                0, fontSize);
    for (int y=0; y<32; y++)
    {   for (int x=0; x<32; x++)
        {   char c = (char)((32*page+y)*32+x);
            g.drawString(String.valueOf(c),
                        fontSize*x,
                        fontSize*(y+2));
        }
    }
    isFilled = true;
}

public void mouseReleased(MouseEvent e) {}
public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
}

/* end of Unicode.java */

```

The output from this program will depend on the range of fonts installed on the computer you run it on. PWF Linux has a range of European characters, mathematical symbols and oddments available. While preparing these notes I ran the code on my home Windows XP system where all sorts of fonts have accumulated over the years, and the image included here (figure 4.2.) is from there. I also use a program called VMware which lets me install many “virtual” computers on my single home one: using that I installed essentially the version of Linux used on the PWF but told the Linux installer to include support for all available languages: by moving some files into a directory “jre/lib/fonts/fallback” I could get results very similar to those that I get from Windows. A message I hope you will absorb here is that Java itself provides portable support for international and special-purpose character sets you may need to configure its runtime before you can take *full* advantage of it. Also before you distribute applications relying



Figure 4.2: Unicode characters in the range 0x3000 to 0x33ff

on that you have to concern yourself with how well your customers’ operating systems will deal with the fonts!

We have already seen string literals in our code, just written within double quote-marks. The associated type is String. Although the use of capitals and lower case is just a convention in Java the fact that the type is String rather than string is a hint that this does not have exactly the same status as the types int, char and so on. In fact String is the name of a quite complicated data-type (in Java we will find that this is known as a class) and this class pro-

vides access to a number of string conversion and manipulation functions. We have already seen “+” can be used to give string concatenation. Java also arranges that if one argument for + is a String it will take steps to convert the other to String format so that this concatenation can take place. You can look up “Class java.lang.String” in the on-line documentation or a reference manual to see that there are standard library functions for case-conversion and all sorts of other string operations.

For now the points to observe are that

1. Strings are represented by a data-type that exports functions to find the length of a string, concatenate strings and perform various conversions;
2. Strings are read-only, so if you want to change one you in fact make a new string containing the adjusted text;
3. Strings are not the same as arrays<sup>9</sup> of characters;
4. It is not necessary to memorise every single string operation that Java provides.

Java supports arrays. An array is just a block of values where one has the ability to use an integer *index* to select which one is to be referenced. The types for arrays are written with the array size in square brackets. An empty pair of square brackets means that the size is not being specified at that point, but it will be made definite somewhere else in the program. We saw an array declaration as early as the Hello program where the function main was passed an array of Strings. In this case the array will be used to pass down to the Java application any words given on the command line after the parts that actually launch the application:

```
// File "Args.java"  
//   Display arguments from command line  
  
public class Args  
{  
    public static void main(String[] args)  
    {  
        for (String s : args)  
            System.out.println(s)  
    }  
}
```

This introduces a new version of the `for` statement. It can be compiled and then run by saying

---

<sup>9</sup>Which will be covered in the next section of these notes!

```
javac Args.java
java Args one two three
```

it prints out

```
one
two
three
```

The points to notice here are that the type of argument that `main` was an array of strings, and the `for` loop will obey its body once for each string in that array. An alternative and older-fashioned way of achieving the same effect would be to find the length of the array and count, indexing into the array to extract values explicitly:

```
for (int i=0; i<args.length; i++)
    System.out.println(args[i])
```

In this case the array held Strings, but Java arrays can be declared in forms to hold any sort of Java data. This includes having arrays of arrays, which is the Java way of modelling multi-dimensional structures.

In Java a distinction is made between declaring a variable that can hold an array and actually creating the array that will live there. Declaring the variable happens just as for declaring integer or floating point variables, and you do not at that stage specify how big the array will be:

```
{    int[] a;
    ...
```

has declared<sup>10</sup> `a` to be ready to store an integer array (of unspecified size and currently unknown contents), much as

```
{    double d;
    ...
```

says that `d` will subsequently be able to store double-precision floating point values. There are two ways that the actual concrete array can come into existence. The first is to combine the declaration with an initializer that makes the array and fills in its elements:

---

<sup>10</sup>The syntax that I will try to use throughout these notes has all declarations written as a type followed by the name of the variable that is to be declared. Thus `int[]` is the type of an array able to hold integers. When you declare a variable of an array type Java allows you to put the brackets either next to the base type (as I will generally do) or after the name of the variable that is being declared, as in `int myArray[];`. This latter case is perhaps useful when you want to declare a bunch of variables at once, some scalars and some arrays, as in `int simple,row[],grid[][];`

```
{   int[] p = {2,3,5,7,11,13,17};  
    ...
```

where the values within the braces can in fact be arbitrary (integer-valued) expressions. The second way of creating an array uses a keyword `new` which is Java's general mechanism for allocating space for things. The word `new` is followed by a type that describes the structure of the item to be allocated.

```
{   int[] fairly_big = new int[1000];  
    ...
```

In this case the array contents will be left in a default state. In fact Java is very keen to avoid leaving anything undefined or uncertain, since it wants all programs to execute in exactly the same way every time and on every machine, so it demands that a fresh integer array starts off holding 0 in every element. It has analogous rules to specify the initial value of any other field or array element that has not had a value given more explicitly.

Note that all Java arrays use subscripts that start at 0, so an array of length 1000 will accept subscripts 0, ..., 999. Attempts to go outside the bounds will be trapped by the Java run-time system. Subscripts must be of type `int`. You may not use `long` values as subscripts. If you write a `char`, `byte` or `short` expression as a subscript Java will convert it to an `int` for you as if there had been a suitable cast visible.

When an array is passed as an argument to a function the called function can update the members of the array, but if it creates a whole new array by something such as

```
args = new String [5];
```

this will replace the array wholesale within the current function but have no effect on the calling routine. The terminology that Java uses for all this is that it will pass a "reference" to the array as the actual argument.

The following example shows the creation of an array, code that fills in entries in it, a slightly dodgy illustration of the fact that two-dimensional arrays can be viewed as arrays of one-dimensional arrays and a crude demonstration of how you might print multiple values of a single line by building up a string that maps the entire contents of the line.

```

// Array1.java
// Create a 3 by 3 array, swap rows in it (!)
// and print tolerably neatly.

public class Array1
{
    public static void main(String[] args)
    {
        int [][] a = new int[3][3]; // 3 by 3 array
        int [] b; // array of length 3
        for (int i=0; i<3; i++) // fill in all of a
            for (int j=0; j<3; j++) a[i][j] = i+10*j;
// The next line recognises that a[i] are 1-dimensional
// arrays of length 3. It swaps two of them around!
        b = a[0]; a[0] = a[2]; a[2] = b;
        for (int i=0; i<3; i++) // Print each row
        {
            String s = ""; // Build row up here
            for (int j=0; j<3; j++)
                s = s + " " + a[i][j];
            System.out.println(s); // Then print it
        }
    }
}

```

which prints

```

2 12 22
1 11 21
0 10 20

```

The things to notice in the above example are firstly that variables `a` and `b` are declared with array types, but these types neither specify sizes nor imply that a genuine array actually exists, and secondly the way in which the two-dimensional array is dismembered. Observe also the syntax associated with `new` for allocating space for the array, and the fact that nothing special had to be done at the end to discard the space so allocated. Java will recycle memory previously used by arrays (and indeed any other structures) once it knows that they are no longer needed. This is of course just like the situation in ML.

We have seen a number of other types in the sample programs. As well as `String` there was `Graphics`, `Font` and `MouseEvent`. Java 1.2 defines over 500 such non-simple types! Thus one thing you can be certain of is that I will not discuss all of them, and neither will the follow-on Java course next year. Each of these has (in some sense) the same status and possibilities as the programs you have written where you start off by declaring a new `class`. Each of `String`,



Graphics and so on represents a class and its implementation might well be in Java stored in a file that starts off

```
public class Whatever ...
```

You have seen with the classes that you define for yourself that a class is a context within which you can define a collection of functions, and so it should be no surprise that each of the 500+ Java library classes provides a whole bunch of associated functions (eg for `String` we have mentioned the `valueOf` operation, and for `Graphics` we have used `drawLine` and `drawString`). There are thus literally thousands of library functions available. Their organisation into classes provides some structure to the collection, but in the end you probably have to find out about the ones you need to use by searching through the documentation. These notes will introduce a sampling of library classes and the functions they support with a view to directing you towards interesting areas of Java functionality. Very often I find that the best way to start to understand the use of a new part of the class library is to study and then try to modify some existing code that uses it.

The Java designers suggest use of a convention where the names of ordinary variables and functions start with a lower case letter while class-names start with a capital. They further recommend use of words spelt in all capitals for things that should be thought of as constants (such as `PI` that we used earlier). The syntax associated with declaring something immutable will be covered later on once we have got through the use of other important words such as `public` and `static` which are of course still unexplained.

### 4.1.3 Exercises

#### Tickable Exercise 3

The function `System.currentTimeMillis()` returns a long value that is the count of the number of milliseconds from midnight on 1st January 1970 to the moment at which it is executed. Thus something like

```
long start = System.currentTimeMillis();
for (int i=0; i<13; i++)
{   System.out.println(binom(2*i, i));
}
long timeSpent = System.currentTimeMillis()-start;
System.out.println("Done in " + timeSpent +
                  " milliseconds");
```

in the middle of a program can be used to record how long it takes to run. Note that this is the time as measured by a stop-watch (or hour glass), and will depend

quite strongly on how many other people are using the computer at the same time. On a single-user computer it can give a tolerably reliable indication of the cost of a computation and even on a shared machine it is better than no information at all.

1. Adapt the Binomial Coefficients program suggested in the previous set of examples so that it reports the time it takes to get as far as displaying  ${}^{24}C_{12}$ , which (I think) has the value 2704156. Your submission to the assessors should include a table of the values of  ${}^{2n}C_n$  for  $n$  from 1 to 12, and the number of milliseconds that your program took to run.
2. Remove the definition of the sub-function that you used to compute the binomial coefficients, and add to your program a line that declares and create an array called `c` of size 25 by 25. Set the `c[0][0]` to 1. Now the first row of the matrix holds values of  ${}^0C_r$ .

Now fill in subsequent rows one at a time using the rules

$$\begin{aligned}c[n][0] &= c[n][n] = 1 \\c[n][r] &= c[n-1][r-1] + c[n-1][r]\end{aligned}$$

so that the matrix gradually gets filled up with binomial coefficients. Keep going until the 24th row has been filled in. Then print out the values of `c[2*i][i]` for  $i$  from 0 to 12, and again measure how long this takes.

3. *[From here on is optional]* The above calculation can be done using a one-dimensional array, so that at each stage in the calculation it holds just one row of binomial coefficients, ie values of  ${}^nC_r$  for a single value of  $n$ . At each stage by filling its values in in reverse order something like

```
c[i] = 1;
for (int j=i-1; j>0; j=j-1) c[j] = ...
```

the new values can replace the old ones in such a way that nothing is overwritten too early. The `for` loop shown here sets `j` first to the value `i-1`, then to `i-2`, and so on all the way down to 3, 2 and finally 1. I could of course have written `j--` or `--j` where here I put `j=j-1`!

Write this version of the program using an array of length 80, and make the array contain `long` values rather than just `int`. First arrange that on every even row it prints the middle element from the part of the array that is in use, so it duplicates the output printed by the previous two examples. Then make the loop continue further and thus find (by inspection) the largest value of  $i$  such that  ${}^{2i}C_i$  can be represented exactly as a Java `long`. The value is less than 40.

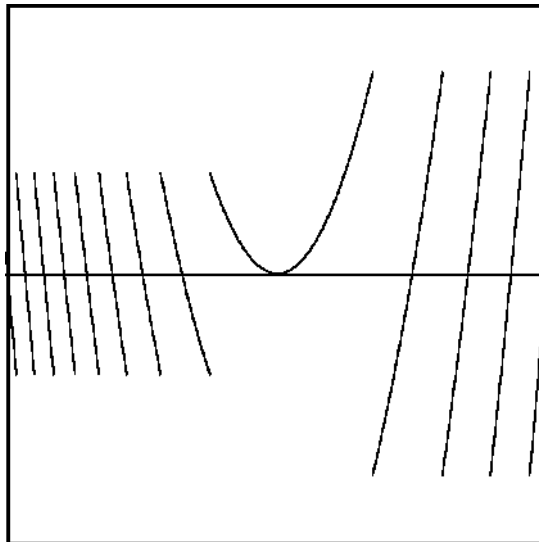
*(End of tickable exercise)*

### A Numerical Monster

A very fine paper called “Numerical Monsters” by Essex et al[12] explains how many calculations that you might think were straightforward have extra depth when done using finite precision computer arithmetic. One example is the function

$$y = (x^2 - 2x + 1) - (x - 1)^2$$

In ideal arithmetic  $y$  would always be zero. If however the function is computed using floating point arithmetic as shown (and provided an over-clever compiler does not do algebraic re-arrangement of the formula: the Java compiler is well-behaved in



this respect) an interesting graph can emerge. For instance the graph shown here was produced using a very simple Java program and plotting for  $x$  in the range  $1 - 3.0 \times 10^{-8} < x < 1 + 3.0 \times 10^{-8}$  and  $y$  from  $-1.5 \times 10^{-16}$  to  $+1.5 \times 10^{-16}$ . Write your own version of a program to re-create this graph, and investigate the what it shows near other values of  $x$ . Two cases I will suggest investigating are  $1.0 \times 10^8 < x < 2.0 \times 10^8$ ,  $|y| < 10$  and  $14999.99999999253 < X < 14999.99999999257$ ,  $|y| < 3.2 \times 10^{-8}$ . Your challenge is to understand exactly how the finite nature of computer arithmetic leads to the precise patterns generated, and how these patterns would vary if details of the arithmetic model were altered.

### The Dutch National Flag

Provided that at the start of your program you have written

```
import java.util.*;
```

The code

```
int [] a = new int [1000];
Random r = new Random();
for (int i=0; i<1000; i++) a[i] = (byte) r.nextInt();
```

first makes an array of length 1000. It then creates a new random-number generator (called  $r$ ), and finally calls the random number generator 1000 times to fill in entries in the array. The cast to type `byte` ensures that each entry in the array will

end up in the range from  $-128$  to  $+127$ . There will of course be duplicate values in the array.

The task you have to achieve is to rearrange the numbers in the array so that they fall into three bands. The first band, say all the elements from  $0$  to  $m$ , should contain all the numbers  $x$  with  $x < -40$ . The second band ( $m + 1$  to  $n$ ) will be for  $-40 \leq x < 40$ , while the final band ( $n + 1$  to  $999$ ) is for  $x \geq 40$ . This is known as the Dutch National Flag problem because its originator (E J Dijkstra) presented it in terms of values that had one of the three colours that his country's flag<sup>11</sup> used, rather than the numeric ranges I have suggested here.

The problem would probably be easy if you could allocate three fresh arrays and copy each item from the original into one or the other of these, based on its "colour". At the end you could then copy the three chunks back into the original array at the positions they needed to go. But this challenge involves the idea of efficiency too, and your final solution must not use any extra arrays, and it should ideally inspect or move each value as few times as it can. Note that just sorting the values in the array into ascending order would satisfy the objectives that concern where values must end up, but since the problem does not state anything at all about any desired order of the items that fall within any of the three bands a solution based on sorting is over-elaborate and too expensive.

It may well be useful to try your hand at the Polish Flag problem — my encyclopaedia shows the Polish flag as having just two stripes<sup>12</sup>. Thus the Polish Flag problem is to rearrange the values in the original chaotic array so that all negative ones (say) come before all positive ones, but without any further constraint on the re-ordering apart that it should be achieved reasonably efficiently.

The Mauritian Flag seems to go Red, Blue, Yellow and then Green. . .

## Matrix Operations

Set up two 5 by 5 arrays of type `double[][]`. Fill in the first so that<sup>13</sup> the element at position  $(i, j)$  has value  $1/(i + j + 1.0)$ . Fill in the other so that the the elements on the diagonal have the value 1.0 while all other elements hold 0.0.

The program wanted now will be one that gradually turns the first matrix into one that has just 1.0 elements down its diagonal and zeros elsewhere. The permitted operations are

1. Multiply all the elements in a row by the same non-zero value;
2. Subtract a multiple of one row from another.

---

<sup>11</sup>Red, White and Blue in that order

<sup>12</sup>Red and White

<sup>13</sup>This form of matrix is known as a Hilbert Matrix.

and whenever one of these operations is performed on the first matrix it must also be performed on the second.

The first matrix can be made diagonal by tidying up first the first column, then the second, then the third and so on. To tidy up column  $i$  you first multiply row  $i$  by  $1/a_{i,i}$ , since this will leave<sup>14</sup> the element on the diagonal as 1.0. Then for every row apart from row  $i$  you subtract a suitable multiple of row  $i$  so as to make the element in column  $i$  vanish.

Do this and display the elements of the second matrix, which should in fact have ended up as the inverse of the original Hilbert matrix!

## Encryption

The following code fragment starts with a string called `key` and fills out an array `k` with repeated copies of the character codes from the key until `k` has 256 entries in it:

```
String key = "My Secret Key";
int keyLength = key.length();
int [] k = new int [256];
for (int i=0; i<256; i++)
    k[i] = (int)key.charAt(i % keyLength) % 256;
```

All the values stored in `k` have been reduced so as to be in the range 0 to 255. Observe the use of functions `length()` and `charAt()` from the `String` class. I have used a fixed string as the keyword here.

The repeated use of the fixed numeric constant “256” in this code is a stylistic oddity. In some ways once the array `k` has been declared it might be nicer to use `k.length` to talk about the number of elements it has. I took the view when writing this that the exact size of the array is part of the core specification of the algorithm I am implementing... When you write this program whatever else you do *please* do not use your password as text in the program you write!

The program you have to write here may be related to an encryption method known as RC4 that was once a trade secret of RSA<sup>15</sup> but which was published anonymously and presumably improperly a couple of years ago. RC4 is used as the encryption technology in a large number of generally used packages and although its security may not have been proved it is widely believed to be respectable. It is also fast.

---

<sup>14</sup>For now assume please that the diagonal element was non-zero so that the division behaved properly and did not end up yielding and IEEE infinity.

<sup>15</sup>The major American encryption and security company. You may like the view consideration of the proper uses of such code as an exercise relating to the Professional Practise and Ethics course.

The first part of the procedure is to create an array  $s$  of 256 integers, and initialise it. It is first set so that the value at position  $i$  is just  $i$  ( $i$  runs from 0 to 255). Now your code should scramble this up using the key  $k$  using the following process:

*For  $i$  from 0 to 255 do*  
     *Let  $j$  be  $(s[i] + k[i])$  reduced to 8 bits*  
     *Swap the items at positions  $i$  and  $j$  in the array  $s$*

The term “reduced to 8 bits” can be implemented by just taking the remainder<sup>16</sup> when the value concerned is divided by 256.

At the end of this the array  $s$  holds a working collection of scrambled data. This is used to generate a sequence of 8-bit numbers which can be combined with a message to encrypt it. Starting with variables  $i$  and  $j$  both zero the next encryption-number is obtained as follows:

*Increment  $i$  modulo<sup>17</sup> 256*  
     *Set  $j$  to  $j + s[i]$ , again modulo 256*  
     *Swap  $s[i]$  with  $s[j]$*   
     *Let  $t$  be  $s[i] + s[j]$  modulo 256*  
     *The result is  $s[t]$*

The algorithm is clearly short enough to be utterly memorable! The sequence of numbers it generates can be added to the (8-bit) character codes of a message to give the encoded version, and if the recipient knows the key that was used then decryption is just generating and subtracting the same sequence of values. It is vital that a key used with this method should *never* be re-used, and competent security tends to involve really careful attention to many details that do not belong in this course<sup>18</sup>.

Code the scheme described above. Print the first dozen output values from it for your chosen key. You may like to check with a friend to see if their implementation generates the same sequence as yours when given the same key — by the nature of this code there is not obviously going to be any other way of characterising the correct output!

---

<sup>16</sup>In the next section we will also see that it can be achieved by writing something like  $(s[i] + k[i]) \& 0xff$ .

<sup>17</sup>“modulo” means just the remaindering operation.

<sup>18</sup>But which will be covered later in the CST.

## 4.2 Operators and expressions

The examples shown already have included uses of the usual arithmetic operators, both as used on integers and on floating point values. Now is the time to present a systematic list of the operators that Java provides and the way they are used in expressions. One of the critical things in any programming language is the syntactic priority of operators. For instance in normal usage the expression  $a + b \times c$  must be read as if it had been  $a + (b \times c)$  rather than as  $(a + b) \times c$ . To stress the importance of knowing which operators group most tightly I will list things ordered by their syntactic precedence rather than by what they do. The simple arithmetic cases will be listed but not discussed at any great length.

- ++, --: We have already see use of ++ as a postfix operator that increments a variable. The full story is that the expression ++a has the side-effect of increasing the value of the variable a by 1 and its overall value is that incremented value while a++ increments a but the value of the expression is the *original* value of a. The use of -- is similar except that it subtracts one rather than adds one to the variable mentioned. These operations can apply to either integer or floating point variables;
- +, - (*unary*): Unary + does not do anything but is there for completeness. Unary - negates its (numeric) argument;
- ~: The ~ operator treats its integer operand as a binary number and negates each bit in the representation. If you look back at the earlier table that illustrated binary numbers you can check that ~0 will have the same value as -1;
- !: The ! operator may only be applied to a boolean operand, and it complements the logical value concerned, so that !true is false;
- (*type*): Casts are included here to show their precedence and to point out that as far as syntax is concerned a cast acts just as a unary operator;
- \*, /, %: Multiplication, division and remaindering on any arithmetic values. The odd case is the % operation when applied to floating point arguments. If  $x \% y$  is computed then Java finds an *integer* value  $q$  that is the quotient  $x/y$  truncated towards zero, and then defines the remainder  $r$  by  $x = qy + r$ . If integer and floating point values both appear in the same expression the integers are promoted to the floating type before the arithmetic is performed. Similarly if integers of different lengths are mixed or if floats and doubles come together the arithmetic is performed in the wider of the two types;
- +, -: Both integers and floating point can be added and subtracted much as one might expect;

+ (*string*): If the + operator is used in a context where at least one argument is a string then the other argument will be converted to a string (if necessary) and the operation then denotes string concatenation. We have seen this used as a way of forcing a conversion from a numeric type to a string ready for printing. Note that the concatenation step will generally involve allocating extra memory and copying data from each of the two original strings, so it will tend to be much more expensive than the arithmetic uses of the + operator.

<<, >>, >>>: Consider an integer as represented in binary. Then the << operator shifts every bit left by a given number of places, filling in at the right hand end with zeros. Thus the program fragment:

```
for (int i=0; i<32; i++)
    System.out.printf("%d : %d%n", i, (1 << i));
```

will print out numbers each of which have representations that have a single 1 bit, with this bit in successive places. The result is a table of powers of 2, except that the penultimate line of output will display as a negative integer and the final one will be zero! There are two right-shift operators. The usual one, >>, treats numbers as signed values. A signed value is treated as if they were first converted to binary numbers with an unlimited number of bits. For positive values this amounts to sticking an infinite run of 0 digits to the left while for negative ones it involved preceding the number with lots more 1 digits. Next the value is shifted right, and finally the value is truncated to its proper width. The effect is that positive integers get 0 bits moved into the vacated high order positions while negative ones get 1s. When shifting an `int` the shift amount will be forced to lie in the range 0 to 31, while for type `long` it can only be from 0 to 63. The special right shift written as >>> shifts right but always fills vacated bits with 0. It is very useful when an integer is being thought of not as a numeric value but as a collection of individual bits;

<, <=, >, >=: The usual arithmetic comparisons are available, and I have already remarked that there are a few delicacies with regard to floating point comparisons and NaNs, infinities and signed zeros;

`instanceof`: This will be discussed later;

`==`, `!=`: Equality and inequality tests. For the primitive types they test to see if things have the same value. For other types (arrays and the object-types that are introduced later on) the test determines if the things compared are “the same object”;



`&`: On integer operands the `&` operator forms a number whose binary value has a 1-bit only where both of the inputs have a 1. For positive values, for instance `a & 0xf` and `a % 16` will always yield the same result. Long tradition of languages where the “and” operator is significantly faster than division and remainder means that many old-fashioned programmers will make what is now maybe excessive use of this idiom. The `&` operator can also be applied to boolean operand, in which case it means just “and”;

`^`: Exclusive or. See below to compare inclusive and exclusive or;

`|`: Inclusive or. Note that for integer values `a & ~b | b & ~a == a ^ b` and the same identity holds for boolean values except that `!` has to be used for the complement/negation operation rather than `~`. Here are the truth tables for inclusive and exclusive or:

“ ”	0	1	“^”	0	1
0	0	1	0	0	1
1	1	1	1	1	0

`&&`: In a boolean expression such as `A & B` if the value of `A` was false there is perhaps no need to evaluate `B`. The simple “and” notation does not take advantage of this, but the alternate form `A && B` does. Apart from efficiency this can only make a difference if evaluating the sub-expression `B` would have side-effects. In general I think it is probably good style to use `&&` rather than just `&` whenever a boolean expression is being used to control an `if` or similar statement, while `&` is probably nicer to use when calculations are being performed on boolean variables;

`||`: This is the version of the “or” operator that avoids processing its right hand operand in cases where the left hand one shows that the final value should be `true`. Its use is entirely analogous to that of `&&`;

`?`: It is sometimes nice to embed a conditional value within an expression, and Java lets you do that using the slightly odd-looking syntax `a ? b : c`. This expects `a` to be of type `boolean`, while the other two operands can have any type provided that they are compatible. If `a` is true the result is the first of these expressions, otherwise it is the second. For instance the messy-looking expression

```
(a==0 ? "zero" : "non-zero")
```

has the value `"zero"` if `a` is zero and `"non-zero"` otherwise. The phrase `(a || b)` could be replaced by the equivalent form `(a ? true : b)`, while `(a && b)` has the same meaning as `(a ? b : false)`;

`=`: Assignment in Java is just an operator. Thus you can assign to a variable anywhere within an expression. The value of a sub-expression that is an assignment is just the value assigned. Thus silly things like `((a=a+a) + (b=b-1))` are good syntax if not good sense. A more benign use of the fact that assignments are just expressions arises in the idiom `(a = b = c = d = 0)`. The assignment operator associates to the right so the example means `(a = (b = (c = (d = 0))))` and thus assigns zero to all of the four variables named;

`*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=`, `|=`: These operators combine assignment with one of the other operators that have been listed earlier. They provide a short-hand notation when the same thing would otherwise appear to the left and immediately to the right in an assignment. For instance `a = a+3` can be shortened to `a += 3`. The abbreviation has slightly more real value when the assignment concerned is to some variable with a name much longer than just `a`, or especially if it is into an array element, since the short form only has to evaluate the array subscript expression once. This can lead to a difference in meaning in cases where evaluating the subscript has a side-effect, as in the artificial fragment

```
int [] a = ...;
int [] b = ...;
int p = 0, q = 1;
for (int i=0; i<10; i++)
    a[p++] += b[q++];
```

where index variables `p` and `q` step along through the arrays `a` and `b` and it is important that they are each incremented just once each time around the loop.

## 4.2.1 Exercises

### Bitwise operations on integers

Investigate, either on paper or by writing test programs, each of the following operations. Explain what they all mean, supposing that the variables used are all of type `int`:

1. `~a + 1`;
2. `a++ + ++b`;
3. `a & (-a)`;

4. `a & ((1<<b)-1);`
5. `(a>>>i) | (a<<(32-i));`
6. `a + (a<<2);`
7. `(int)(byte)a;`
8. `(a & 0x80000000) != 0;`
9. `(a++ != b++) && (a++ == b++);`
10. `(--a != --b) | (--a == --b);`
11. `(a < 0 ? -a : a).`

### Counting bits in a word

Write a function that counts the number of non-zero bits in the binary representation of an integer. You can first do this using a test for each bit in the style `a & (1<<n) != 0`. Next see if any of the examples in the previous exercise give you a way to identify a single bit to subtract off and count. Consider also the expression

```
c[a & 0xff] + c[(a>>8) & 0xff] +
c[(a>>16) & 0xff] + c[(a>>24) & 0xff]
```

for some suitable array `c`.

### What does this do?

This exercise and the few following it introduce a few fragments of amazingly twisted and “tricky” code. Please do not view the inclusion of these programming techniques here as an indication that you will be examined on them or that you are being encouraged to use such obscure constructions in your own programs. It is more that puzzling through these examples can refine your understanding of the interactions between the Java arithmetic operations such as `+` and `%` and the ones that work on the binary representations of numbers, ie `&` and `>>`. In a few circumstances the ultra-cunning bit-bashing might save time in a really critical part of some program and so could be really important, but almost always clarity of exposition is even more important than speed. Certainly use of these tricks will not make your programs any shorter, in that the bulk of the comments needed to justify and explain them will greatly exceed the length of more straight-forward code that has the same effect!

Start off with `a` any positive value. Execute the following<sup>19</sup> and discuss what value gets left in `a` at the end. Hint: look at the binary representation of `a` to start with.

```
a -= (a>>>1) & 03333333333;
a -= (a>>>1) & 03333333333;
a = ((a>>>3) + a) & 0707070707;
a = a % 63;
```

**And this...**

```
a &= 0x3f;
a = ((a * 02020202) & 0104422010) % 255;
```

**And this...**

```
c = a & -a;
r = a + c;
a = ((r ^ a) >>> 2) / c | r;
```

[In each case it will help if you look at the numbers in binary.]

### Integers used to represent fractions

Consider `a` as a value expressed in binary but now as a positive fractional value in the range 0 to 1. This means that there will be an implicit binary point just to its left. Then `0xffffffff` will be just a tiny bit less than 1, `0x80000000` will stand for 1/2 and `0x40000000` for 1/4. In terms of this representation interpret the effect of executing the following four statements one after the other.

```
a += a >>> 2;
a += a >>> 4;
a += a >>> 8;
a += a >>> 16;
```

### Division and Shifts

If `a` is a positive integer then `a/2` and `a>>1` give the same result. What is the relationship between their values if `a` is negative. Carry on the analysis for division by 4, 8, 16,... and the corresponding right shifts.

<sup>19</sup>Discussions with Alan Mycroft caused this and some of the other curious examples here to re-surface. For a collection of real programming oddities including some of these try searching for “Hakmem” on the World Wide Web or find “The Hacker’s Delight”[15] in a library

### Some Exclusive-Or operations

What is the final effect on `a` and `b` of the sequence

```
a ^= b;  
b ^= a;  
a ^= b;
```

### Sieve for primes

Create an array of type `boolean` and length 1000. Set all elements to `true` to start with. Then set items 0 and 1 to `false`

Repeat the following two steps

1. Find the first `true` item in the array. If there are none left then exit. Print out the index of the value you have found, and call it `p`.
2. Set each item in the array that is at a position that is a multiple of `p` to be `false`, for instance as in

```
for (i=p; i<1000; i+=p) map[i] = false;
```

The numbers you have printed should be the primes up to 1000.

If you wanted to find the primes up to several million (for instance to count them rather than to tabulate them) it would make sense to make the array represent just the odd numbers not all numbers. It might also save significant amounts of space to represent the array as an array on `int` rather than `boolean` and pack information about 32 odd-numbers into each `int`. You might note that some programming languages can implement boolean arrays in that way without much user intervention — Java does not.

## 4.3 Control structures

### 4.3.1 Exercises

#### Ambiguous If

Consider the sample code fragment

```
if (a == 0)  
    if (b == 0) System.out.println("Both 0");  
    else System.out.println("Some other case");
```



Figure 4.3: Keep control structures simple!

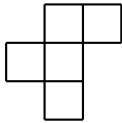


## Life

The world consists of an infinite sheet of graph paper. Each square may at any one time be either black or white. Every square has eight neighbours. Every so often all squares simultaneously follow the following rules:

1. A square that is black at present remains black if it has two or three black neighbours. Otherwise it turns white;
2. A white square becomes black if it has exactly three black neighbours.

3	4	5
2	⊙	6
1	8	7



These rules define a behaviour which was invented by John Conway (who at the time was in the Mathematics department here) and which is known as Life. One starts off with a board that has a small number of black seed points and display the position as the generations go by. There are many astonishingly complicated things that can happen and people have designed starting positions that illustrate them. The challenge here is to make the computer run the rules and display the world-state. A useful starting configuration to try has just five black cells arranged as at the head of this paragraph. It explodes and seethes for quite a long while before the situation stabilises. One thing to note is that all decisions about the next generation are expected to be taken simultaneously, so any program that updates the world incrementally is liable to get wrong answers. A further problem is that the ideal playing surface for Life is infinite, while computers tend not to be. Two resolutions to this are usually considered. One places an immutable wall of white cells as a border around the world so that all activity is contained within them. The other scheme often used is to use a finite playing area but considers its left hand column to be adjacent to its right most one and its top to be adjacent to its bottom row. This amounts (depending on how you think of it) to playing Life on a torus or to ensuring that all initial positions are replicated in a periodic manner across an infinite plane.

The easiest program for this will set up *two* boolean arrays. The first holds the current generation, while the second will be filled in with the next. My version of a program that does this, complete with code to set up the initial pattern that I have suggested and to draw the board positions in an applet window is around 75 lines long. I used a 200 by 200 board and kept the outermost rows and columns permanently blank. That means that when accessing neighbours I can read from them without going outside my array. Clearly the first exercise here is to reproduce something like that.



There are then three follow-up challenges. The first looks back to the optional part of the binomial coefficient tickable exercise: can you get away with just one boolean array rather than two, possibly keeping a boolean vector to store just one row of backup information but mostly updating the world in place. To do so would save around half of the memory that the simple program uses.

The second challenge observes that representing the playing area as arrays of type `boolean` is probably wasteful. This would be a typical application where packing 32 cells into an `int` and using lots of bitwise `and`, `or` and `shift` operations to deal with them would be common practice. It would of course be possible to achieve this by having nice abstract procedures to reference the bit at position  $(x,y)$  in an array even though the array was being represented in a packed way. But it would perhaps give big speed savings to look for ways to exploit the fact that bitwise operations on integers can handle 32 bits all at once and to try to use this to compute new values for several cells at the same time.

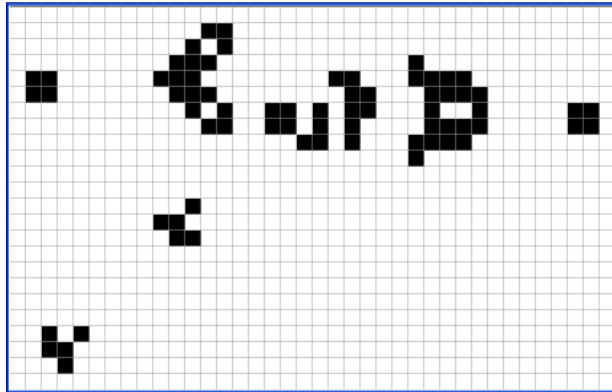


Figure 4.4: Gosper's Glider Gun.

Finally, and given that this program tends to run a little slowly, one looks at where the time goes. Much of it will be wastage on parts of the board that are totally white and hence where nothing is going to happen. Try to speed your code up by avoiding as much of such wasted as is reasonable.

### Eight Queens

Count the number of ways of placing eight queens on a chess board so so that no pair are in the same row, column or diagonal as each other. This is a classical puzzle to go in an introduction to programming and there are lots of clever tricks that can be used. It is the sort of thing that most supervisors will have come across before so I will not provide a fully worked through solution here, but I might observe that the search might well be done by a recursive function that when called at depth  $n$  will try to place a queen on row  $n$  of the chess-board.

## Permutations

In Java arrays can be passed as arguments and newly created arrays can be returned as results. Write a function that accepts an array of strings as its argument, and which hands back an array whose elements are themselves arrays of strings giving all possible permutations of the input. For instance if I use curly brackets to denote arrays here one might like `{"a", "b", "c"}` to turn into `{{"a", "b", "c"}, {"a", "c", "b"}, {"b", "a", "c"}, {"b", "c", "a"}, {"c", "a", "b"}, {"c", "b", "a"}}`.

As with several of the other Java exercises I might suggest that you design and test an ML version first.

## 4.4 Control structures Part 2

There are two aspects of syntax that I will put off until a yet later section. One is the syntax associated with the word `class` that we have seen wrapped around every program we have written. The other is the matter of the “.” that appears between or possibly within so many names, eg `System.out.println` and `g.drawString`. A few other bits of syntax will just not be covered in this first course, although you may find traces of them in the grammar and discussion of them in textbooks — and possibly also in next year’s “Concurrent Systems and Applications” course.

But I will talk through each of the important components of the syntax and give at least one illustration of each.

### 4.4.1 Expression Statements

Certain sorts of Java expression can be used as a statement — all that is necessary is to stick a semicolon on the end of it. The cases permitted are where evaluating the expression might have a side-effect. Thus an assignment expression, a function call or a pre- or post-increment expression can be used. As an example, consider the statement `x++;` which just increments `x`. An example such as `1+2+3;` is not considered valid: it would calculate the value 6 and then throw it away!

### 4.4.2 Blocks

Several statements can be placed one after the other to make a single large statement. Braces `{ ... }` are used around the statements to group them. In various earlier languages the keywords `begin` and `end` were used instead of braces, but Java prefers the version where you type in fewer key-strokes. If you see a block

with semicolons in the semicolons are just parts of expression statements and nothing special to do with the fact that there is a block there. Again some earlier languages differed by using semicolons between statements in a block rather than as termination of expression statements. Blocks can be nested any way you want. You may insert extra braces to stress the grouping of any collection of statements you feel deserves that, in much the same way that extra parentheses can always be used to emphasize the grouping within expressions. I think there are enough examples of blocks throughout these notes that I do not need to give a special one here.

### 4.4.3 Null statements

If you insert a stray semicolon into a Java program it (mostly) does not matter much, since a semicolon alone can be interpreted as an empty statement that does nothing. The most striking example of the use of a null statement is in something like

```
if (a > 7);  
else System.out.println("Gotcha");
```

where the `if` needs a statement before its `else` part but no real action is needed. If you *really* need such a place-holder I would suggest that the following is clearer and flags your unusual intent more clearly.

```
if (a > 7) { /*NOTHING*/ }  
else System.out.println("Gotcha");
```

Better yet rearrange your code to make tests happen in a positive sense:

```
if (a <= 7) System.out.println("Gotcha");
```

### 4.4.4 `if`

It takes a little while to get used to the fact that the condition tested by `if` is written in parentheses. Some people prefer a style where the statement after an `if` is always written as a block, even if it is only a single statement, so that the range that the `if` controls is made very explicit. This point of view has some sense behind it, especially if the statement after the `if` is more than half a line long.

The control expression used by `if` must be of type `boolean` and so equality tests are written as in `a==0` and not `a=0`<sup>22</sup>. Using a single rather than double equals sign is a common slip.

---

<sup>22</sup>Which would be an assignment and would have type `inti`.

### 4.4.5 while, continue and break

A `while` loop executes its command repeatedly for so long as the guarding expression remains `true`. Its syntax is very much like that of `if`. Within the iterated command you can embed a statement “`break;`”, and execution of that will cause a premature exit from the loop. The command `continue`;<sup>23</sup> can be useful if the iterated command is a long block, and it causes the loop to proceed at once to its next cycle. Both `break;` and `continue;` are very convenient at times, but it is often good style to avoid them when reasonably convenient so that the boolean expression at the top of the `while` loop represents a total statement about the circumstances in which it will loop and when it will terminate. The following sample shows a fairly typical `while` loop. Look back at your Discrete Mathematics notes for explanation of why it computes a highest common factor and to give clues to a reason for carrying out the extra computations. You may also like to code up an extended Euclidean algorithm as a function that calls itself (say in ML rather than Java) and observe that use of `while` loops does not always lead to the shortest or most transparent code.

```
int a = 72, b = 30;
int u = 1, v = 0;
while (b != 0)
{   int q = a / b;
    int r = a - q*b;
    a = b;
    b = r;
    int t = u - q*v;
    u = v;
    v = t;
}
// Here a is the HCF. What are u, v?
```

Note that `break` can be used to exit other loops, and it is also used with `switch` statements, which will be described soon.

### 4.4.6 do

Sometimes the most natural way to write a loop puts the test of a termination condition at the end of the loop rather than at the start. This circumstance is supported by the `do` statement, although I find it much less useful than `while`. In fact I will often express

do

---

<sup>23</sup>Observe that the syntax for each of these command includes a semicolon, The identifier mentioned in the full grammar is something I will not discuss here.

```

{
    ...
} while (xxx);

```

by writing it instead as

```

while (true)
{
    ...
    if (!xxx) break;
}

```

since I think that `do` puts the details of what the loop is about rather too far down the page. Anyway that also gave me a chance to include an example of a `break` statement for you! The issues of programming style here could give rise to a variety of discussions. A good policy is to try rather hard to make it very clear and obvious just when each loop you write is going to terminate, and indeed to make it clear (in comments as necessary) why you know it will eventually terminate.

#### 4.4.7 for

Iteration with `for` has been seen in several examples. What is shown in the Java syntax is that each of the three components within the parentheses and separated by semicolons is optional. The most extreme case is when none are present: `for (;;) { ... }` means just the same as `while (true) { ... }`.

In `for (A;B;C)` the expression `A` is an initializer evaluated just once at the start of the loop. `B` is a boolean expression and is used just as in a `while` statement to determine when to terminate. Finally `C` gets evaluated between each cycle of the loop, and it often increments some variable. The idiom `for (i=0;i<N;i++)` executes its command `N` times counting from 0 to `N-1`. The alternative way of writing things is `for (i=1;i<=N;i++)`. It loops the same number of times but is maybe slightly less commonly used. Of course with the second version the variable `i` starts at 1 not 0: this typically makes it less suitable for use as an array subscript because in Java subscripts start at 0.

#### 4.4.8 switch, case and default

There are occasions when one wants to dispatch to many different code fragments based on the value of some expression. This could be achieved by writing a chain of `if .. else` statements, but often `switch` provides a much neater way of expressing things.

The construction starts with `switch (Expression)`. The expression given must be of type `char`, `byte`, `short` or `int`. Note that `long` is not allowed. The

`switch`-header is followed by a block enclosed in braces, and within this block there can be special switch labels. The usual sort reads “`case Constant:`” and control arrives just after the colon if the integer value of the switch expression agrees with the constant after `case`. It is often useful to specify what action should be taken if none of the cases that have explicit coverage happen, and for this a label “`default:`” can be set. Case (and default) labels do not disturb the usual sequential execution of statements, and so unless something special is done after one case is processed control will proceed to the next one. This is usually not what is wanted. A `break;` can be used to exit from the entire switch block. Many programmers would count it is good style to put an explicit comment in whenever a `break` is not being used, to show that its omission was deliberate and not an accident.

If no explicit default label is given but a switch is executed in such a way that none of the cases match it just acts as if there had been a default label just before its final close brace.

It is generally a good thing to use `switch` whenever you have more than three or four options to select between, in that it tends to be much clearer and easier to understand than length strings of nested `if` statements. In the following rather silly example it is imagined that the user has provided the function `show`. Observe that the case labels do not have to be in any special order, and that a single statement can be attached to several labels.

```
switch ((int)n)
{
case 2: show("the only even ");
        // drop through
case 3: case 5: case 7:
case 11: case 13: case 17:
        show("prime\n");
        break;
case 4: case 9:
case 16: show("square\n");
        break;
case 8: show("cube\n");
        break;
default: show("dull or too large\n");
        // now just drop off the end
}
```

### 4.4.9 return

When a function has done all it needs to it will want to return a result. This is achieved using the `return` statement. Function definitions (see later) always indicate what type of result is required. They may have used the keyword `void` to indicate that no result is needed. Such is the case with `main`. For `void` functions one just writes “`return;`”, while in all other cases the syntax is “`return Expression;`”.

### 4.4.10 try, catch and throw, finally

Real programming languages need to be able to implement code that can recover from errors and handle unusual cases tidily. The handling scheme in Java uses the `throw` statement to raise exceptions. Throw statements specify an object which should generally<sup>24</sup> be of type `Exception`<sup>25</sup>. The effect is that control exits from the current block or procedure and any enclosing ones, all the way until a suitable handler is found. If no such handler is present the computation is terminated. The system has a number of built-in exceptions it will generate. For instance an attempt to divide by the integer 0 raises an exception of type `ArithmeticException`. Various functions that read from files can raise exceptions to indicate that the file did not exist, the current user did not have permission to read it or an attempt was made to read data beyond its end.

Handling exceptions involves prefixing a block with the word `try` and adding on the end of it one or more clauses that describe what to do in unusual cases. A clause that starts `catch (Argument)` is followed by another block which gets obeyed if the system detects an exception whose type matches that declared for the *Argument*. A single `try` may be followed by `catch` handlers for several different types of exception.

```
try
{ z = 1/0; } // raises an exception!
catch (ArithmeticException e) { ... }
```

After all `catch` clauses you can put the keyword `finally` followed by another block. The intent here is that this block will get executed come whatever, and it will usually be used to tidy files or data-structures that the program might otherwise have left in a mess. A typical scheme to provide robust access to files would go something like

```
<open the file for reading>
try
```

---

<sup>24</sup>I do not want to give the full and precise rules here!

<sup>25</sup>To be more precise of some type derived from `Exception`.

```
{   while (true) <read-more-from-file>
}
catch (<end-of-file-exception>)
{   // whole file read here. Good!
    <success code>
}
finally
{   // must tidy up even if some failure
    // other than end-of-file intruded
    <close the file>
}
```

Later on I will give concrete examples that fill in the function calls and so on in this framework.

Some programmers view `catch` and `throw` as neat and convenient language features to be used wherever they fit. Certainly the file-handling example above makes very good use of them. Others, and I tend to fit into this category, would like to see them used rather sparingly in code since they can result in all sorts of loops and functions terminating unexpectedly early and therefore undermine attempts to make absolute statements about their end results.

#### 4.4.11 `assert`

A statement of the form `assert Expression;` will evaluate the expression (which really ought not to have any side-effects. If its value is *false* and if some magic flag was supplied when the Java launcher was run then an exception is raised. Assertions can have a second expression that can be used to give more details of what you thought had gone wrong. It is proper style to include them at places in your code where there is some reasonably cheap consistency check that you could apply and when used well they are a huge aid to testing and debugging.

If you run your program normally the assertions will not be checked, and furthermore having them in your source file will not hurt performance enough to notice. If however you run the `java` command with the extra flag `-ea` the extra checks will be done. Usage such as

```
java -ea:CheckThisClass SomeClass
```

will arrange to check just the assertions in the named class.

#### 4.4.12 Variable declarations

Variable declarations can occur anywhere within a block. They are also allowed in the first component of a `for` statement. The scope of a variable that is declared



within a block runs from the declaration onwards until the end of the block. A declaration made in a `for` statement has a scope that covers the remainder of the `for` statement, including the end-test and increment expressions as well as the iterated block. In fact the scope of a declaration appears to include the initialiser for that variable, but if you try to use the variable there you should expect at least a warning message. So things like

```
{   int x = x+5;
    ...
}
```

should not be attempted! A local consists of a type, then the name of the variable being declared, and optionally one or more pairs of square brackets (to denote the declaration of an array). Any initializer follows an “=” sign, and for arrays the initializers are written in braces so that many individual values can be given so as to fill in the whole array. A local variable declaration can be preceded by the word `final`, and this marks the variable that is being declared as one that will not subsequently change. A convention is that constants should be spelt entirely in upper case, as have the examples `PI` and `PLAIN` that have been seen so far. Here is an example:

```
final double E = 2.718281828459045235;
E = 1/E; // NOT valid because of "final"
```

#### 4.4.13 Method definitions

A function definition starts with some optional qualifier words. The available words are

```
public      protected  private    static
abstract   final       native     synchronized
```

and if present these can be written in any order. I will explain what they mean later on. Next comes the type of result the function will return, which is either an type or the special word `void` to indicate “no result”. Next is the name of the function that is being declared, followed by a list of formal arguments (in parentheses). A formal argument must be given a type, and may be preceded by `final` if the body of the function will never update it. The grammar shown earlier indicated that pairs of square brackets may be written after the formal parameter list, but this should not be used in any new code<sup>26</sup>. If the execution of the function can cause an exception to be raised and this exception is to be caught somewhere then the fact must be mentioned by following the list of formal parameters by the

---

<sup>26</sup>It is a concession to some earlier versions of Java where it could be used for functions that returned arrays.

keyword `throws` and then a list of exception types. Finally there is a block (ie some statements within braces) that forms the body of the function that is being defined.

For the moment you will *still* have to take the qualifiers `public` and `static` on trust. They relate to the construction of the class that the whole file defines.

#### 4.4.14 Exercises

##### Concerning $3n+1$

Take any number  $n$ . If it is even then halve it, while if it is odd replace it with  $3n+1$ . Repeat this process to see what happens in the long run. For various very small integers you will find that you end up in a cycle  $1 \rightarrow 4 \rightarrow 2 \rightarrow 1 \dots$  but it is not at first clear whether this is the ultimate fate when you start from an arbitrary integer.

Write a program that generates the sequence starting from each integer from 1 to 1000. If the sequence ends at 1 record the number of steps it took to get there. If you have taken over 10000 steps on some particular sequence then stop and report just that value: after all maybe the sequence starting from that seed goes on for ever, either by diverging to infinity or by finding a cycle different from the one that includes 1. If on the way you generate an odd number larger than  $(\text{Integer.MAX\_VALUE}-1)/3$  you should also stop the calculation there since otherwise you would suffer from integer overflow and subsequent work would be nonsense. The constant `Integer.MAX\_VALUE` is another Java built-in constant useful in cases such as this.

Arrange that you only print anything when a new record is broken for the length of a sequence or when you would reach integer overflow. For each record-breaker display the seed, the number of steps taken before 1 is reached (or the fact that an overflow occurs) and the largest value in the sequence concerned.

##### Tickable Exercise 4

As you start this exercise note that ticks 1, 2, 3 and 4 are probably fairly easy. Tick 5 is going to be a somewhat larger piece of work so as soon as you have finished this one you might like to look ahead and get started on it!

In ML a function called `quicksort` could be defined as

```
fun select ff [] = []
  | select ff (a :: b) =
    if (ff a) then a :: select ff b
    else select ff b;
```

```

fun quicksort [] = []
  | quicksort (a :: b) =
    quicksort (select (fn p => p < a) b) @
    [a] @
    quicksort (select (fn p => p >= a) b);

```

The idea is to use the first element of the input list as a *pivot*. One then selects out first all the remaining values that are less than this pivot, and all the values that are at least as large. Recursive calls sort the two sub-lists thus generated, and a final and completely sorted list is obtained by concatenating the various parts that have been collected.

The ML version is very elegant and shows some of the important ideas behind the Quicksort algorithm. However it misses out several other things that are important in the real Quicksort method, mostly issues concerning use of memory. In this exercise you are to implement a version of Quicksort in Java. You should write a procedure with signature<sup>27</sup>

```
void quickSortInner(int [] v, int i, int j)
```

which will sort that part of the array `v` that has subscripts from `i` to `j`. It will be up to you to decide if these limits are inclusive or exclusive. The procedure should work by first seeing if the sub-array it has to work on is empty. If so it can return without doing anything! Otherwise it should take the first (active) element of the array as a pivot and rearrange<sup>28</sup> the remaining items so that the array gets partitioned at a point `k` such that the pivot has been moved to position `k`, all items to the left are smaller than the pivot and all items to the right are at least as large as it. It should do this re-arrangement without using more than a few extra simple variables: ie it is not acceptable to create a whole fresh array and copy material via it. `quickSortInner` can then call itself recursively in a way suggested by the ML code to complete the sorting process.

You should also define a function called just `quickSort` that takes only one argument — the array to be sorted. Remember that the `.length` selector can tell you how large the array is.

To show that your code works you should demonstrate the following tests:

1. Create an array of length 10 and show the effect of sorting it when its initial contents are (a) the numbers 1 to 10 starting of in the right order to begin with, (b) 1 to 10 in exactly the opposite order to begin with, (c) ten numbers generated by `nextInt()` from the random number package (d) ten numbers all of which are zero;

---

<sup>27</sup>The signature of a function is just the specification of the types of its arguments and result.

<sup>28</sup>Remember the National Flag exercise.

2. Measure the time taken to sort various length vectors of random data where you should use lengths 16, 32, 64, ... up until the sorting run takes several seconds. For each test compute the quotient of the time taken and the value  $N\log(N)$  where  $N$  is the number of items being sorted.

**Optional** part for those who are keen: Read the Java documentation for the `Array.sort(int [])` method that Java 2 provides. Write code to time it and compare the results with the code you wrote yourself. When measuring times work with arrays long enough that each test takes several seconds. Observe that the fact that the Java libraries provide you with sorting methods (see also `Collections.sort`) means that most Java users will never need to implement their own Quicksort: you are doing it here as an exercise and because it is good for Computer Scientists to understand what goes on inside libraries, since next time around it may be their job to implement libraries for some new language.

As a further *optional* extension to this exercise consider the following and adjust your code accordingly, then repeat all your tests:

1. The ML quicksort partitions items by comparing them with the value that happened to be first in the list. In the plausible cases where the original data is already in ascending or descending order this leads to excessive cost. Selecting as the “pivot” the median of the first, last and middle element from the array being sorted<sup>29</sup> does better;
2. It is probably best to stop `quickSort` from recursing once it gets down to sub-arrays of length 3 or 4. The end result is that it *almost* sorts the array, but a final pass of bubble-sort can finish off the job nice and fast. Is this born out in your code?
3. The partitioning code here can be delicate! Unless you are careful it can escape beyond the bounds of the array, or it can get muddled about whether the two final values in the middle of the array need exchanging or not. Simple implementations can be made safe by making all the end-conditions in your loops composite ones rather like

```
while (k>=i && v[k] > pivot) ...
```

while if we could get away with it it should be faster to go something more like

```
while (v[k] > pivot) ...
```

---

<sup>29</sup>Always supposing there are at least 3 items in the list.

Investigate how well you can trim down your inner loops while retaining code that always works! The Part IB course on Data Structures and Algorithms and the textbook by Cormen et al[9] are where this level of detailed study really belongs!

*(End of tickable exercise)*

### Highest Common Factors

Implement code to compute Highest Common Factors using the Euclidean Algorithm. Extend it to use the extended algorithms that at the end will allow you to solve equations of the form

$$Au + Bv = 1$$

### Tickable Exercise 5

The work called for here will be done in sections, and it is expected that while working towards the tick you will be able to design, code and test each section before moving on to the next. The idea involves creating a package of routines that can compute with (univariate) polynomials. For the purposes of this exercise a polynomial

$$(a_0 + a_1x + a_2x^2 + \dots + a_nx^n)/b$$

will be represented as an instance of a the class:

```
class Poly
{
    private String variableName;
    private long [] coeffs;
    private long denominator;
    ... constructors and methods as needed
}
```

where `variableName` holds “*x*”, the array called `coeffs` stores the coefficients  $a_0$  to  $a_n$  and the `long denominator` holds the value shown as  $b$  above. Because Java lets you enquire as to the length of an array it is not necessary to store the degree  $n$  explicitly. In this representation common factors should be cancelled out between numerator and denominator, and the highest degree coefficient  $a_n$  should never be zero. In this exercise all polynomials will be in terms of the same variable,  $x$ , so the `variableName` should always be set to “*x*” and it will not play much of a part in any of the calculations! Step by step carry out the following tasks, testing what you have done as best you can as you go:

**Create simple polynomials:** Write functions that can create the “polynomial” that represents just a given integer, a given fraction and the simple polynomial “ $x$ ”;

**Debug-quality printing:** Write code that takes a polynomial and displays its coefficients. For this part of the exercise it is not at all important that the display format you design be tidy or that it respects line-lengths. So for instance you may generate output such as

$$(1*x^0 + 0*x^1 + -3*x^2)/2$$

with various unnecessary symbols in there. The object is to be able to see your polynomials clearly enough that you can test and demonstrate what comes next!

**Special-case multiplication:** Write code to multiply a polynomial by an integer, to divide it by an integer, and to multiply it by  $x$ . Note that in the first two cases you will need to do calculations (involving greatest common divisors<sup>30</sup>) to reduce the coefficients to lowest terms. In the latter case the result will be of one higher degree than the input and so will be represented with a coefficient vector one item longer. These routines should not alter their input, but should create new polynomial data to represent the results;

**Addition and Subtraction:** Take two polynomials and create another that represents their sum (or difference). This involves more fun with ensuring that the result is over a common denominator, and subtracting two polynomials can lead to a result of lower degree if the leading terms cancel (as can adding if the leading terms start off as similar but with opposite signs);

**Multiply:** If you have one polynomial of degree  $m$  and one of degree  $n$  then their product is of degree  $m + n$ . Write code that computes it;

**Differentiate:** in fact differentiation of a polynomial by its variable is rather an easy operation (and so would be integration, which you would need in an optional extra to this exercise). If the polynomial contains an original terms  $a_i x^i$  then the derivative contains just  $(i a_i) x^{i-1}$ ;

**Proof of pudding part 1:** Let  $P_0 = 1$ ,  $P_1 = x$  and from there on define a sequence using the recurrence relationship

$$P_n = ((2n - 1)xP_{n-1} - (n - 1)P_{n-2})/n$$

---

<sup>30</sup>Otherwise known as highest common factor.

Using your polynomial manipulation program calculate and tabulate the values up to (and including)  $P_{12}$ ;

**Proof of pudding part 2:** Now instead define

$$P_n = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n$$

(This is known as Rodrigues formula, in case you wondered, and the polynomials you are computing are Legendre Polynomials)

Using this recipe compute and display values up to  $P_{12}$ . The two sequences of polynomials you have computed ought to match!

**Testing to destruction:** extend your tables until the values computed by the two recipes for  $P_n$  are incorrect because of some internal integer overflow, and report where your program first displays a result that is certainly incorrect;

**Optional extra (a):** Write code that evaluates a polynomial at an integer value of its variable, ie at  $x = n$ . Write code that computes the (indefinite) integral of a polynomial with respect to its variable. Combine these two to allow you to evaluate definite integrals. Display a table showing values of

$$\int_{x=-1}^1 P_i(x)P_j(x)$$

for  $i$  and  $j$  running from 0 to 5 (say);

**Optional extra (b):** Let  $y$  be one of the polynomials ( $P_n$ ) that you have just computed. Evaluate

$$(1 - x^2)y'' - 2xy' + n(n + 1)y$$

Tabulate this for various small values of  $n$ .

Note: the examples worked with here are Legendre polynomials, and provide an example taken from Sturm-Liouville theory. Optional extra (a) shows that they are orthogonal over the range from  $-1$  to  $+1$  and this in fact makes them useful for producing certain sorts of good numerical approximations to functions. Optional extra (b) is showing you that these polynomials are solutions of a differential equation: many other interesting sequences of functions satisfy recurrence formulae, have orthogonality properties *and* are solutions to differential equations! Abramowitz and Stegun's book of tables[1] is probably the easiest place to suggest you look to find out more.

(End of tickable exercise)

### Pollard Rho integer factorisation

In previous years this was Tickable exercise 5. There are in fact a few delicacies with regard to integer overflow (which do not greatly damage it as an exercise but which could raise questions about it). You may still like to try it!

The explanation of this exercise is quite long, and it maybe looks messy, but I can assure you that the code that has to be written is tolerably short and manageable once you have sorted out what needs doing.

Randomised factorisation: Implement the following algorithm that (possibly) finds a factor of an integer that it is given:

A single trial that looks for a factor of  $N$  is performed by selecting a random positive number  $R$  and computing  $S = R \% N$ . This is a number between 0 and  $N - 1$ . If the number is 0 deem this trial a failure. Next compute the highest common factor of  $S$  and  $N$ . If this is 1 then again the trial is deemed a failure. However if the HCF is not 1 then it is a factor of  $N$  and because it is also a factor of  $S$  it must be less than  $N$ . This counts as success!

The complete factorisation algorithm works by running a number of trials. If for a number  $N$  the first  $\sqrt{N}$  trials all fail then we will pretend that  $N$  is prime. Otherwise a factor of it has been found, and dividing this into  $N$  gives us its co-factor. Smaller factors of each of these can then be sought using the same technique.

Use this procedure to try to factorise the numbers  $2^i - 1$  for  $i$  from 2 upwards, stopping when your program starts to take more than a second or so to run.

The Birthday problem: Suppose we have a sequence of numbers all of which are less than  $N$ , and these values are generated in some way such that each number  $x_n$  is some fixed function of  $x_{n-1}$ . A concrete example would be if  $x_n = (x_{n-1}^2 - 1) \% N$ . For most  $N$  and for  $x_0 = 2$  this sequence<sup>31</sup> is in fact pretty unpredictable.

Any such sequence must eventually repeat a value, and once it has it necessarily continues in a loop. If consecutive values behave well enough as if they are random up to this point then the expected length of the sequence

---

<sup>31</sup>There is a significant delicacy here: when you compute  $x_{n-1}^2$  its value can be almost as large as  $N^2$  even though the remaindering will rapidly bring it down to a smaller range. This can lead to integer overflow and a particularly un-wanted effect is that a value you generate may end up unexpectedly negative (when  $N^2$  is outside the valid range of integers). I suggest you mostly ignore this here (!) and at most take an absolute value to ensure that the sequences you generate consist of positive numbers. Also there is not much special about starting with  $x_0 = 2$  and other randomish starting values might work just as well.



before a repeat is related to the problem of how many people you have to have in a room before you should expect to find that two of them share a birthday. In this case the room is on a planet in a galaxy far far away, where the length of the year is  $N$ , and the statistics suggest that we need around  $\sqrt{N}$  of our aliens.

For this exercise you are to imagine one algorithm that detects a cycle and implement a second and much better one.

The algorithm you just have to imagine guarantees to find a cycle as soon as it arises. It allocates a big array and stores values in this array as they are generated. As each one is generated it also checks through the ones already seen to see if the new value has occurred before, and if so declare the loop detected. This method is easy to visualise but it needs an array as long as the longest potential loop, and the search means that before finding a loop at step  $n$  it has done about  $n^2/2$  comparisons with old values. This is slow.

The second method, which you should implement, records the value of  $x_i$  each time  $i$  reaches the next power of 2 and compares newly generated values against this one stored value. It argues that if there is a loop then eventually the loop will be totally traversed between consecutive powers of 2, and thus will be detected. Furthermore this will be at worst a factor of two beyond the place where the first repeat happened.

Having coded the second loop-detection algorithms try it on sequences generated by  $x_n = x_{n-1}^2 - 1 \pmod N$  for various  $N$  and verify that for a reasonable proportion of values of  $N$  and  $x_0$  a loop is found after very roughly  $\sqrt{N}$  steps.

**Pollard Rho:** This builds on the previous two parts, so please do not start it until you have completed them. But then re-work the loop detection code so that instead of comparing each new  $x_n$  with a saved value  $x_{2^k}$  using an equality test compute the HCF of  $N$  and  $x_n - x_{2^k}$ . Stop if this is not 1, ie if a factor of  $N$  has been found. In the case when  $x_n = x_{2^k}$  the method has failed: you may either give up in that case or try starting again with a different value for  $x_0$ .

Implement this using the Java `long` type. If the number  $N$  is composite it is probable (although not guaranteed) that this will find a factor of  $N$  within around  $\sqrt[4]{N}$  trials, and will thus be able to find a factor any Java `long` value quite rapidly. Of course if  $N$  starts off as a prime this scheme will never manage to find a factor of it! To test this you should probably create numbers that are known to be composite by multiplying together two `int`-sized values.

*Optional:* The scheme above does not of itself find a complete decomposition of an integer into prime factors — it just splits composite numbers into two. A complete factorisation method needs to extend it with first a filter so that numbers that are prime are not attacked, and secondly with recursive calls that try to factor the two numbers that Pollard Rho split our number into. Investigate the Java `BigInteger` class that provides arithmetic on long integers and which also provides a test for (probable) primality. Re-implement your code to use `BigInteger` rather than `long` and to use `isProbablyPrime` to avoid trying to factor when it is futile. Thus produce code that can produce complete factorisations of reasonably large numbers. How many digits long a number can you factorise in say 20 seconds?

Some of you no doubt consider yourselves to be Java experts! You may like to arrange that the calculation  $x_{n-1}^2 - 1 \bmod N$  is computed exactly even when  $N$  is almost as large as a Java `long` can be, and that overflow does not interfere. An easy way to do this is to use the Java library big integer support, but what I would prefer here would be code expressed entirely in terms of use of `long` arithmetic.

## 4.5 Java classes and packages

What has been described thus far should provide a foundation for understanding the small-scale structure of Java programs. If you have understood it you are equipped to write programs that have up to (say) half a dozen sub-functions and that are limited to living in a single source file. So far the data that Java can work with has been limited to the primitive types `int` and so on, together with arrays of them. The time has now come to discuss the Java idea of a *class*. This is used both to support the construction of user-defined data-structures and to impose an order on programs that are large enough that they should properly be spread across several source files. A discussion of Java classes will include an explanation of what all the “.” characters are doing in the sample programs seen so far. All of this counts as “Object Oriented Programming”.

One of the aspects of programming language design that has proved to be especially important is that control of the visibility of names. This whole issue tends to look rather frivolous — a distraction — while your programs are only a page or two long but it makes a critical difference to big (and perhaps especially collaborative) projects. There are several interlocking reasons to want to keep name-spaces under control. One as so that a large chunk of code can be given a cleanly defined interface consisting of the functionality that it makes visible to the outside world. Everything not so exported is then deemed private to the group who maintain that body of code, and they may change internal parts of their design

... old-fashioned approaches  
to software construction ...



Figure 4.5: Classes and Packages make Java “modern”.

with complete confidence that this can not hurt anybody else.

A second reason for keeping name-spaces well controlled is so that different parts of a large program are free to re-use the most obvious names for their functions and variables, secure that this can not introduce unexpected clashes.

Related to both of these is the fact that when trying to understand code limits on the visibility of names can allow you to concentrate on just the range in the code where something is relevant.

Java controls access to names at three levels. At the finest grain it has scope rules that are much like those of most other programming languages. If a local variable is declared within a block that variable can only be referenced using code textually within that block. Java understands the idea that a re-declaration of a variable in an inner block would create a different variable with the same name, and that within the inner block the new variable would shadow the old one, as in

```
int func(int a)
{
    {   int a = 4, b = 5;           // ???
        for (int a=0; a<10; a++) b++; // ???
        System.out.printf("%d %d\n", a, b);
    }
    return a;
}
```

and it views this as something that could be codified and that to a computer has a totally logical interpretation. But that it is a potential cause of real confusion to human programmers so it should be prohibited! Thus the above example will be rejected by the Java compiler and all the interesting computer-science discussion of exact rules about scope can be set aside. You may like to note, however, the variables do not clash in any way if their scopes do not overlap, so the following is valid:

```
int func(int a)
{
    {   int i = 4;
        for (int j=0; j<a; j++) i++;
        System.out.println(a + " " + i);
    }
    for (int i=0; i<10; i++) a *= 2;
    for (int i=0; i<a; i++) a--;
    return a;
}
```

The scopes associated with each declaration of `i` are disjoint.

The other two aspects of Java name-space control are more interesting. The important words used here are *class* and *package*.

All names of Java variables and procedures<sup>32</sup> live in some class. In general you have to gain access to the class before you can use its members<sup>33</sup>. A member of the current class can be referred to just by giving its unqualified name, but in other cases you need to have access to an object of the required class and refer to the member using a dot “.” as a selector on it. This is what was happening in cases such as `g.drawLine` where `g` was a variable of type `Graphics` and `drawLine` was a member of that class. When a class is defined the user can arrange which of its members can be referenced by other classes in this manner, so that internal details of the class can not even be accessed using this sort of explicit naming. The word `public` flags a component of a class that should be universally visible while `private` marks one that should not.

Classes thus contribute in two ways to the avoidance of confusion over names. Firstly they mean that most references to things outside the current class will include a dot selector that indicates fairly explicitly what context the name is to be taken from, and secondly they can arrange that some names are kept totally local to the class within which they are used and can *never* be accessed from anywhere else. It is perhaps worth reminding you at this stage of the qualifier `final` that can turn a variable declaration into the definition of a constant. There are further refinements in the control of name visibility and use that Java provides, and the keywords `protected`, `abstract` and `static` relate to some of them: these will be discussed later on.

Classes themselves have names, and so a scheme is needed to structure the name-space that they live in. A collection of classes can be placed in a “package”. When classes are declared only those that have been given the `public` attribute<sup>34</sup> are visible outside the package. Furthermore since the idea is that any other Java code<sup>35</sup> can access the public classes of a package there is a somewhat curious linkage between package names and the filing system on your computer. This linkage is mediated by a thing called the “class path” which can list the places that Java should search to find the compiled code if you refer to a class defined in some package. You can expect that any reasonable default Java setup will have your class path set up for you already so that you can access all of the standard Java libraries and so that code in the current directory can be used. The full names of classes generally contain dots. Various names starting with the component `java` are reserved for the system, and ones starting with `sun` are for use by Sun

---

<sup>32</sup>From now on I will increasingly move towards the Java notation and call these “methods”.

<sup>33</sup>We will see later that in some cases, when the name has been declared as `static`, one can refer to the item via the name of its containing class. But in the more general case it will be necessary to have an instance of the class and access the item via that.

<sup>34</sup>Making a class `public` is a similar idea to making a member of that class `public`, but of course we are talking now about a different level in the structure of a program.

<sup>35</sup>Ideally anywhere in the world!

Computers, who designed Java. The various further parts to package names are intended to group packages into hierarchies. For instance every package whose name starts with `java.awt` is to do with the Java Abstract Window Toolkit, which is the part of Java that provides facilities to pop up windows on your display. The package `java.awt.event` is the sub-part of this that contains classes relating to events — we have seen an example where these could be caused by the user clicking with the mouse but there are others. The Java documentation contains a list of all the predefined packages that are part of the Java core, and then lets you browse the complete set of classes defined in each. Each class of course provides a number of variables and methods: the number of standard library methods is huge!

Specifying full names in the package hierarchy could become very tedious, so Java provides a user-configurable way of setting up shorthand forms of reference. Recall that various sample programs we have seen began with a collection of `import` statements

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

This adjusted Java's name resolution scheme so that the class `MouseEvent` (say) could be referred to by that short name. Without the `import` it would still have been possible to write the same program but it would have been necessary to use a fully spelt-out "`java.awt.event.MouseEvent`" to name the class, and that would involve knowing exactly which of the standard parts of the Java library `MouseEvent` belongs in. The "\*" in the `import` statements show tells Java to support short names for all the classes in the packages named. It is also possible to put a single class name in an `import` statement. This could be useful if only one member of a package was to be used and you did not want to risk confusion with other names from that package. Some Java programmers take the view that `import` with a "\*" introduces risk of giving them access to classes other than ones they know about, and so always spell their imports out fully, despite that being a little more verbose. Note that the syntax of Java only allows the wild-card "\*" to be placed right at the end of an `import` where it means "all the classes in this package".

If you issue `import` statements that attach to two or more packages that define identically named classes then Java will refuse to get muddled: it just insists that you use fully spelt-out names for the classes that could otherwise be ambiguously resolved. This is probably safer than a scheme where the first (or possibly the last?) `import` statement took precedence.

Java comes with around 200 and huge numbers of pre-defined classes, and so getting to know them all is a big job. You are not expected to do so especially

as future Java releases will add yet more, and it is probable that when you work on any big Java project you may find yourself using substantial third-party class libraries. But it does make a lot of sense for you to have a good overview of what is available to you so that when relevant you can use existing well-tested library code rather than starting to write something of your own.

As well as being a way of organising the name-space all Java classes count as data-types. When something's type is a class it is usual to refer to the thing as an "object". Thought of in terms of objects, a class defines a data structure that contains fields that are the variables defined in it and happens also to be able to contain definitions of functions that will access these fields. If the variables were declared public then any code anywhere can access them and so there do not really have to be any (explicit) methods defined within the class. There will in fact always be a few implicitly defined ones that are to do with the creation and deletion of objects.

Taking a minimal approach<sup>36</sup> to class definition I can now set up a definition that would let me represent binary trees where each node in the tree contains an integer:

```
// Compare the ML version, which would be
//  datatype Tree =
//      nullTree |
//      makeTree of int*Tree*Tree;
// or some such.

class BinaryTreeOfIntegers
{
    public int value;
    public BinaryTreeOfIntegers left;
    public BinaryTreeOfIntegers right;
}
```

Comparison with the ML version reminds us that it is important to be able to have some way of telling when the left and right children of such a tree do not really exist. In ML that was achieved with an explicit alternative constructor, which I called `nullTree`. In Java *any* variable which has a class<sup>37</sup> as its type can either hold a proper instance of that class (ie an object) or it can hold the special value `null`. This value is provided as a keyword in Java. Ie the word `null` is hard-wired into the Java language and not just some curious pre-defined variable. It also has the odd property that the same value may be used with any sort of class

---

<sup>36</sup>The class I define here would work, but it misses out on exploiting a lot of structuring and security features that classes can provide, and so is *just* a minimal start.

<sup>37</sup>Or array.

or array variable to set the variable to a state where it “does not hold an any object at all”.

Once a class has been defined it will be useful to declare variables using it and create objects to go in them. Here I will create a rather small tree using the above class definition:

```

...
BinaryTreeOfIntegers a1, a2, a3;
// a1, a2, a3 are all un-initialised here, and
// Java complains if you try to compile a program
// that relies on the values of variables that
// might not have been given a value.
a1 = new BinaryTreeOfIntegers();
a2 = new BinaryTreeOfIntegers();
a3 = new BinaryTreeOfIntegers();
a1.value = 1;
a2.value = 2;
a3.value = 3;
a1.left = a2;
a1.right = a3;
// the next 2 lines are not needed in that
// null is the default value given to a field
// that would hold an object.
a2.left = a2.right = null;
a3.left = a3.right = null;
...

```

Note (but do not worry about, for now) the parentheses after the class name following `new`. And also observe how dreadfully clumsy all this is.

Note that Java provides default initialisers for instance variable in classes and elements in arrays, but not for local variables within methods. The default values used are zero for numeric fields, false for booleans, `'\0'` for characters and `null` for all references.

Anybody who is a C or C++ programmer is liable to have a question to ask at this stage, but those who have mostly seen ML should see all this as reasonable. You can also see `“.”` being used as a selector to access the components of a class object. The C programmers can read my footnote<sup>38</sup>! Java objects are created in much the same way as Java arrays are, using `new`, and there is no need to take

---

<sup>38</sup>In C or C++ one would distinguish rather carefully between a structure and a pointer to the structure. And in C terms all Java class variables hold pointers. However in Java it is not really useful to think this way since all Java operations have been designed to prevent any explicit tricks involving pointers. Please try to think of Java objects as more in the style of ML data. In C you the explicit visibility of the difference between a structure that is directly at hand and one that is referred to via a pointer leads to a distinction between the use of `“.”` and `“->”` to access



any special action when you have finished with one. The Java run-time system is expected to tidy up memory for you. However grossly excessive object creation can either consume time or utterly run you out of memory. The first loop in the following code does not do anything very useful with the objects it creates, and it discards them all rather rapidly. It may be a bit inefficient. The second loop creates a million objects and chains them all together so that none of space concerned can be recycled. At one million you may get away with this, but if you tried to do this a few hundred times more your computer's memory would not be able to keep up with the demands of the program and an exception would be raised to report this fact.

```
for (int i=0; i<1000000; i++)
{   BinaryTreeOfIntegers x =
    new BinaryTreeOfIntegers();
    x.value = i;
    // x is implicitly discarded here
}
BinaryTreeOfIntegers w;
for (int i=0; i<1000000; i++)
{   BinaryTreeOfIntegers x =
    new BinaryTreeOfIntegers();
    x.right = w; // chain on to w
    w = x;
}
```

There are very few cases in Java where it would be considered good style to define a class that only had variables defined within it<sup>39</sup>. Mostly an attempt will be made to collect almost all of the methods that work with the class as part of it. Very frequently the variables in the class can then be made `private`, and the `public` methods provide a clean and abstract interface to everything. There is something of a convention about providing and naming methods to access the data stored in an instance of a class: methods that update variables have names starting with `set`, ones that retrieve boolean values start `is` while others that retrieve information start with `get`. Here is the previous example expanded to follow these conventions, and adjusted so that the case of boolean variables can be illustrated:

---

components. Again Java does not need this and so only has one notation, even though in some sense it uses dot where a C programmer would naturally reach for an arrow.

<sup>39</sup>The most plausible good case I can think of is when all the variables are marked as `final` so they are constants and the class is just used to encapsulate the name-space within which these constants are defined.

```
// Compare the previous Java version where
// the variables were public but there were
// no methods.
class BinaryTreeOfBools
{
    private boolean value;
    private BinaryTreeOfBools left;
    private BinaryTreeOfBools right;

    public void setValue(boolean n) { value = n; }
    public void setLeft(BinaryTreeOfBools t)
    { left = t; }
    public void setRight(BinaryTreeOfBools t)
    { right = t; }

    public boolean isValueTrue()
    { return (value==true); }
    public BinaryTreeOfBools getLeft()
    { return left; }
    public BinaryTreeOfBools getRight()
    { return right; }
}
```

For small classes this just adds way too much extra verbiage and feels silly. However for a large and complicated class with many other methods having a regular and predictable naming can be a real help. It also provides a way that you can give read-only access to some variables or you can check the sanity of values to be assigned to others, ending up with much finer-grained control over access than even use of the `public` and `private` qualifiers give you. The term “bean” is sometimes used for Java classes that follow this set of conventions, and some programming tools exploit it. Because it makes small programs so much bulkier I will not use this style in every example in these notes, but you can notice that many of the Java library classes clearly have and you might think about it again when you move on to writing large classes for yourself.

Here is a sample Java class that might be useful within other programs and that illustrate methods that actually do something useful. It is a start at code that will enable Java code to work with complex numbers. An odd-looking programming style that it illustrates is one where to combine two complex numbers, say `a` and `b`, one will call a method associated with one of them, passing the other as argument. Thus the sum of the two values will be requested as `a.plus(b)`. It is not possible (in Java) to redefine or extend the basic “+” operator to make it “add” objects from some new user-defined class, hence use of a method name such as `add` is

necessary here<sup>40</sup>.

```
public class Complex
{   private double x, y;
    // define setX, setY, getX, getY here if you want.

    public Complex(double realPart, double imagPart)
    {   x = realPart;
        y = imagPart;
    }
    public double modulus()
    {   return Math.sqrt(x*x+y*y);
    }
    public Complex plus(Complex a)
    {   return new Complex(x + a.x, y + a.y);
    }
    public Complex times(Complex a)
    {   return new Complex(x*a.x - y*a.y,
                          x*a.y + y*a.x);
    }
}
```

This would be placed in a file `Complex.java` and compiled using `javac` in the usual way to make a file `Complex.class`. Because I have not put in a package statement this class will live in a default package, and when other Java programs run and they want a class called `Complex` they might manage to find this one if its class file is still in the current directory.

The `Complex` class illustrates one new concept. Observe the method definition that uses the name of the class as its own name and which does not specify a separate return type:

```
public Complex(double realPart, double imagPart)
{   x = realPart;
    y = imagPart;
}
```

It has no return statement in it. A method whose name matches that of the class is a *constructor* and you will typically use it with `new` to create fresh instances of the class thing concerned. If you do not specify an explicit constructor function then a default one is supplied — it has no arguments and does not leaves all variables in their default state. It is valid to have several constructors provided that the types

---

<sup>40</sup>In contrast the language C++ does allow you to extend the meaning of all the operators that are denoted by punctuation marks. Many people believe the conciseness and elegance that can be achieved that way is more than balanced out by the potential for severe confusion.

of their arguments are different. Observe here how the methods that are members of the class all have access to the `private` variables, but no code outside the class will have.

Sometimes when referencing a variable it is useful to stress that you are talking about one in the current instance. The keyword `this` always refers to the object from which you invoked a method, and so the constructor and the `plus` methods above could have been written out in a way that some would consider clearer:

```
public Complex(double x, double y)
{
    this.x = x;
    this.y = y;
}

public Complex plus(Complex a)
{
    return new Complex(this.x + a.x, this.y + a.y);
}
```

Explicit use of `this` can be used to avoid mixups if the name of a formal parameter for a method matches the name of a variable in the class. Consider the following and the muddle that would arise without the use of `this`, but also note how much nicer it is to select names that avoid any hint of a clash.

```
public Complex plus(Complex x)
{
    return new Complex(this.x + x.x, this.y + x.y);
}
```

## 4.5.1 Exercises

### Complete the `Complex` class

The class as shown here does not support division, and does not have an equality test. If you define a method called `toString()` in it then will be called to “print” the number when you use “+” to concatenate it with a string. Finish off the `Complex` class adding in these and whatever other facilities you feel will be generally useful.

### Polar Complex Numbers

The `Complex` represents complex numbers in Cartesian form, ie as  $x + iy$ . But the internal variables `x` and `y` that it uses are both `private` so nobody outside the class can tell this! An alternative representation of complex numbers would store a number as a pair  $(r, \theta)$  where the complex value concerned had modulus  $r$  and argument  $\theta$ . In other words one would have  $z = re^{i\theta}$ . At the cost of computing a few arc-tangents and the like it is possible to create a re-worked `complex`

class that has exactly the same external behaviour as the original one but which stores internal values in polar form. The constructor function and addition become messier, multiplication becomes easier and the modulus function becomes utterly trivial. Implement and test the polar version of the class.

### Wolves and Caribou

On a certain island there live some wolves and some caribou. In year  $n$  there are obviously  $w_n$  wolves and  $c_n$  caribou. What happens the next year depends...

- Wolves hunt, and the total number of dinners they get is proportional to  $w_n k_n$ . The number of baby wolves is automatically proportional to the number of dinners their (potential) parents are able to eat over and above the amount needed to keep the parents active. The wolf minimum feed intake and reproductive capability may be modelled as

$$w_{n+1} = w_n + k_1 w_n (c_n - k_2)$$

- In each year the stock of caribou would increase by a factor  $k_3$  were it not for the depredations of the wolves, since each dinner for a wolf is one less member of the herd. Thus

$$c_{n+1} = k_3 c_n - k_1 w_n c_n$$

- Baby wolves eat, hunt and reproduce as from year  $n + 1$ , and there are no losses of caribou other than as described above. In particular we do not have to worry about over-grazing etc.

At the beginning of time the island is stocked with a herd of 100000 caribou, and a medium-sized pack of ravening wolves. Over a number of years various things could happen. Either wolves or caribou or both could die out, or the populations could stabilise. For some values of the constants and initial wolf population various of these do indeed occur. For instance if at the start there are twice as many wolves as caribou the next year there will only be wolves left (one should adjust the equations given so that negative populations get turned into zero ones), and the year after that the wolves all expire of hunger.

In fact for many configurations the populations do not stabilise, but they do often get locked into stable cycles that last several years. This improbable situation has been observed by real naturalists not only in the situation described here but with regard to disease spread (mumps and children say) and other natural systems. Write java code to investigate.

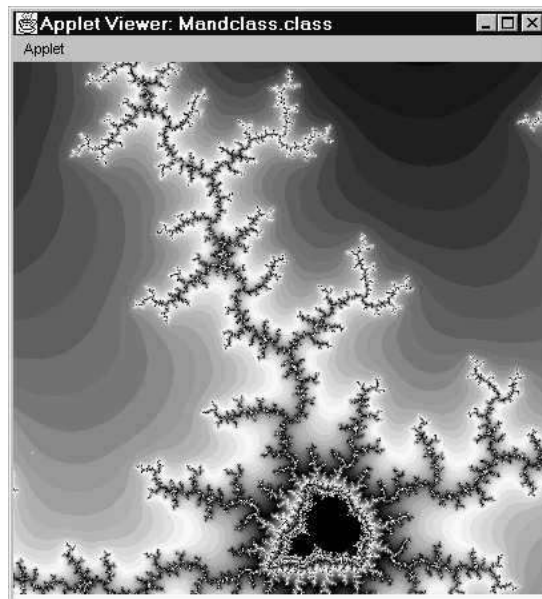
### Packages and jar files

Make a sub-directory called (say) `ex251` and put some Java source files there. Put package `ex251` at the top of the files. Now compile the code, eg saying `javac ex251/*.java`. By unless you explicitly set a *classpath* Java looks for classes that are in a given package by using the package name as if it described a chain of sub-directories down from the current directory. So now set up several different packages and create files that illustrate the use of `protected` and others that fail to compile because you have not made allowance for suitable cross-package visibility. Now look up about *jar* files and prove to yourself that you can take a complete Java program (consisting of many classes) and consolidate it into a single (jar) file that can then let anybody else run it in a simple and convenient way.

These activities are not essential for any of the example programs that you have to write for this year's Java course, but starting to investigate and practise now will put you in a good position for some of next year's work, and particularly the Group Project. I am also aware that this exercise is asking you to read ahead in these notes...

### Tickable Exercise 6

The following definition of a `paint` method uses the rudimentary `Complex` class as shown earlier in this section. The Mandelbrot set can be drawn by considering the sequence defined by  $z_0 = 0$  and  $z_{n+1} = z_n^2 + c$  where both  $z$  and  $c$  are complex numbers. For most values of  $c$  eventually values of  $z_n$  become large. If one counts and finds the smallest  $n$  such that  $|z_n| > K$  for some suitable  $K$  then that  $n$  will depend on the value of  $c$  that was used. The well-known pictures arise by using different colours to display the values of  $n$  associated with values of  $c = x + iy$  as  $x$  and  $y$  vary. Because drawing this involves a significant calculation for every single point within the applet's window it can be painfully time-consuming. To arrange that the screen looks more interesting this



code arranges to draw a crude blocky version first and then gradually refine it into the correct high-resolution image. You may have seen some web browsers do similar things to give better apparent responsiveness when loading and displaying pictures from web sites! The code draws a part of the Mandelbrot set centred around `(midX, midY)` and with width `range`, these referring to the values of the constant  $c$  in the iteration. If the value of  $z$  has not grown large within `LIMIT` steps it is supposed that it never will. The code illustrates use of the `Color` class. Colours are sometimes specified in terms of the amounts of Red, Blue and Green that go to make them up. Printers will tend to think in terms of Cyan, Magenta and Yellow<sup>41</sup> while in yet other circumstances one uses Hue (running through the colours of the rainbow), Saturation (eg white through pinks up to a full-blooded rich red) and Brightness (all colours fading to black at zero brightness, just as all wash out to white (or grey) at zero saturation).

Insert this program in a suitable framework and investigate other areas of the display by altering the relevant variables. You should be aware that if you increase the screen size or `LIMIT` the code can become very time-consuming. Indeed it might very well be sensible while testing to decrease the finest resolution used to say 8 rather than 1. And because the `paint` method computes the whole picture each time it is called any disturbance of the screen is liable to provoke a complete re-calculation (at great cost). I find that the `appletviewer` does not exit until the end of a call to `paint()` and so even quitting from it can involve an amazingly long delay!

```
public void paint(Graphics g)
{ // I Paint first in crude 16*16 blocks and then
  // in finer and finer tiles. This is so that
  // SOMETHING appears on the screen rather rapidly.
  for (int resolution=16; resolution>=1; resolution/=2)
  { double midX = -0.25, midY = 0.85; // Adjust these
    double range = 0.004;           // Adjust this
    int screenSize = 400;           // Match .html
    int s2 = screenSize/2;
    for (int y=0; y<screenSize; y+=resolution)
    for (int x=0; x<screenSize; x+=resolution)
    { int n = 0;
      int LIMIT = 250; // Maybe adjust this?
      Complex z = new Complex(0.0, 0.0);
      Complex c =
        new Complex((range*(x-s2))/s2 + midX,
                    (range*(y-s2))/s2 + midY);
      // Important loop follows.
```

---

<sup>41</sup>Printing inks favour analysis in terms of subtractive colours rather than additive ones.

```

while (n++ < LIMIT && z.modulus() < 4.0)
{
    z = z.times(z); // z = z * z;
    z = z.plus(c); // z = z + c;
}
// Draw in black if count overflowed
if (n >= LIMIT) g.setColor(Color.black);
// ... otherwise select a colo(u)r based on
// the Hue/Saturation/Brightness colour model.
// This gives me a nice rainbow effect. If
// your display only supports 256 (or fewer)
// colours it will not be so good.
else g.setColor(Color.getHSBColor(
// cycle HUE as n goes from 0 to 64
(float)(n % 64)/64.0f,
// vary saturation from 0.2 to 1.0 as n varies
(float)(0.6+0.4*
Math.cos((double)n/40.0)),
// leave brightness at 1.0
1.0f));
// screen coords point y downwards, so flip to
// agree with norman human conventions.
g.fillRect(x, screenSize-y, // posn
resolution, resolution); // size
}
}
}

```

Complete the program based on the above and test it.

Next check `Graphics.getClipBounds` and `Rectangle.contains` in the Java documentation. Adjust the program so that when `paint` is called it first finds the clip rectangle associated with the re-paint operation. This is a rectangle on the screen such that only points within this area need to be re-displayed. Arrange that the loop on `x` and `y` that at present re-computes the colour for every point on the whole screen just loops round doing nothing for points outside the clipping rectangle and so only does the expensive operations for points inside it. Try the new version, and in particular move other windows to obscure small parts of it and then move them away so you can see the effect of the partial re-draw operations.

Note that the above program will display best if your screen is set up to support lots of colours. On a display with either 16-bit colour (65536 colours) or true-colour (24 or 32 bit) and at high resolution the effect is fairly stunning. If only 256 colours are supported the shapes will remain nice and wiggly but the delicate shading will be lost. While preparing these notes I have adjusted the program



to display a 1200 by 1200 image at best-possible resolution in 16-bit colour, and although it takes utterly ages for the screen to refresh I think it is almost worth it!

The program that I give has a bug that you can see if you watch carefully when it re-paints at the various different resolutions. It relates to the fact that in Java the x-co-ordinate increases from left to right (as expected) but the y-co-ordinate is zero at the top of the screen and largest at the bottom. Identify and correct the behaviour that I count as a defect.

*Optional:* Add a `BufferedImage` to make the re-painting of the screen cleaner. I might like to be able to reset the view to some standard one at the click of a mouse, and to be able to drag with the mouse to select a sub-part of the current picture for zooming in on. Those who are feeling keen can investigate these possibilities.

There is also quite some incentive to find ways of speeding up drawing of the images here!

*(End of tickable exercise)*

### Fractions

Create a class similar to the `Complex` one but that implements rational numbers, is fractions. You will probably want to make the internal representation a pair of `long` values rather than just `int`, and keep everything reduced to lowest terms by cancelling out highest common factors.

### Series for $\tan(x)$

It may be well known that

$$\tan(x) = x + \frac{1}{3}x^3 + \frac{2}{15}x^5 + \frac{17}{315}x^7 + \frac{62}{2835}x^9 + \dots$$

but fewer people are happy about being able to predict what the next few coefficients in the expansion are. However if we have a computer program able to compute with rational numbers it is in fact easy to generate as many more coefficients as are desired. The coefficients satisfy a recurrence formula

$$\begin{aligned} t_0 &= 0 \\ t_1 &= 1 \\ t_n &= \frac{1}{n} \sum_{i=0}^{n-1} t_i t_{n-i-1} \end{aligned}$$

Use this to confirm the series as I have tabulated it and display the next few terms. The result here may be derived from the fact that the derivative of  $\tan(x)$  is  $1 + \tan^2(x)$ , and is also related to (but rather harder than!) the discussion of “generating functions” in the probability course.

### Complex elementary functions

Perhaps you already did this when making your complete version of the complex numbers class...

It might be useful to be able to construct complex numbers either by specifying real and imaginary parts or by giving argument and modulus<sup>42</sup>. Failing any better scheme you could distinguish between the two constructors by adding declarations

```
public static final int CARTESIAN = 0;
public static final int POLAR     = 1;
```

in the Complex class and then having the constructor take an extra argument that specifies which option is being used. It would then also make sense to provide data access methods that make it equally easy to access the number in polar or cartesian interpretation.

If that is done it becomes reasonably easy to support complex versions of several of the elementary functions. Observe the identities:

$$\begin{aligned} \sqrt{re^{i\theta}} &= \sqrt{r}e^{i\theta/2} \\ \log(re^{i\theta}) &= \log(r) + i\theta \\ \exp(x + iy) &= \exp(x)e^{iy} \\ \sin(z) &= (\exp(iz) - \exp(-iz))/2i \\ \cos(z) &= (\exp(iz) + \exp(-iz))/2 \\ p^q &= \exp(q \log(p)) \end{aligned}$$

The expressions for sin and cos can be inverted to find ways of writing the inverse trigonometric functions as messy complex logarithms. And it may be seen that the neatest way of using these formulae to implement complex-valued versions of the elementary functions really does benefit from being able to slip very comfortably between the cartesian and polar views of the values. Implement it all.

I should observe carefully that the code you have just written is liable to be a very long way from the last word in elementary function libraries, for the following reasons, which are given in descending order of importance:

1. Several of the complex-valued elementary functions have branch-cuts. For instance the square root function has a principal value which is discontinuous as you cross the negative real axis, and the various inverse trig functions will also have cuts. Your code can not automatically be assumed to implement these cuts in the way that will be considered proper by experts in the field. Probably the most readily accessible description of which cuts are desirable is in *Common Lisp, the Language* by Guy Steele[21];

---

<sup>42</sup>ie by giving the polar version.

2. Your implementation will probably suffer from arithmetic overflow (and hence give back answers that are infinities or NaNs) substantially before the desired result would overflow. For instance the identity given for  $\cos$  computes an intermediate result that is twice as big as the final answer, and hence can suffer in this way thereby returning incorrect answers;
3. In many cases the naive use of the formula given can lead to serious loss of numerical accuracy when values of similar magnitude are subtracted one from the other. For instance this problem would arise in the calculation of  $\sin(z)$  for  $z$  near zero;
4. Direct use of these formulae will not even give an efficient set of recipes for the desired functions!

however the numerical analysis to address these problems is certainly beyond the scope of this course.

### Binary Trees

Start from the `BinaryTreeOfIntegers` class sketched above and extend it so that as well as defining variables in the class it provides a set of methods to work with them. The methods you introduce should arrange that any binary tree built is always structured so that all integers stored in the left sub-tree that hangs off a node are smaller than the integer in the node itself, while all integers in the right tree are greater (or equal). You should provide a constructor that creates such a tree out of all the integers in an integer array, and another function that first counts the size of a tree, then allocates an array that big and finally copies all the integers back into the array so that they end up in ascending order. I would fairly strongly suggest that you design and implement the key parts of this in ML before you move on to the Java version. Your code is an implementation of tree-sort: you should compare it with quicksort for clarity, amount of code that has to be written, robustness (ie are there any truly bad sorts of input it can be given) and performance.

## 4.6 Inheritance

There is one more major feature of the Java class mechanism. It provides yet further refined control over name visibility and it can often be a huge help when organising the structure of large projects. It is called *inheritance* and the idea of it is to allow the user to define new classes as variants on existing ones. When this happens the new class starts off with all the components and methods of the one

upon which it is based, and it counts as having defined a sub-type. It can however define extra variables and/or methods and implement more specialised versions of some of the methods already present in its parent class. This is what has been happening every time we have used the word `extends`, and so for instance every applet we have written has defined a new class extending the library class `Applet`. This library class implements all the major functionality for getting a window to appear, and to get the visual effects we wanted all that was needed was to provide our sub-class with its own version of a `paint` method.

There seem to be three interlocking reasons why inheritance is important when large programs are to be written:

1. Class libraries can be provided in forms that implement all the generic behaviour of really quite complicated programs, but by making a new program that inherits from such a class and that overrides some of its methods lots of flexibility is left for the programmer to create a system that does exactly what they want. Prior to languages that supported inheritance there was a severe conflict between having libraries that contained large enough components to give large time-savings and those that were adaptable enough to be realistically useful;
2. Class inheritance serves a linguistic purpose in Java. If you start from a single base class it is possible to derive several other classes from it. All these count as specialisations of the original one, and a variable capable of holding a member of the base class can therefore automatically refer to instances of any of the derived ones. This is how Java can support data-structures that can have several variants. Furthermore the name-visibility rules in Java can use the way in which inheritance groups classes into families to further refine access to class members.
3. It often becomes possible to implement a set of basic classes first, and test them, and then leave those alone (and hence stable) while deriving new classes that add extra functionality. This both provides a respectable strategy for organising system development, and means that there is a significant chance that the basic classes that are developed will be useful in the next project;

I will try to illustrate these three points in turn.

### 4.6.1 Inheritance and the standard libraries

The richest and most valuable place where this happens in the libraries relate to applications that pop up windows. Examples given before show user code being

derived from a class called `Applet`. One of the things that has been seen about `Applet` and hence any class derived from it is that the method `paint` has a special status, in that it is invoked whenever the screen needs to be refreshed. The fact that by deriving a new class you get an opportunity to write your own `paint` method and that in your new class your own definition takes the place of a standard one (which probably does nothing much!) is obviously critical. If you could not alter the re-painting behaviour of an applet the whole structure would lose its point. If you look at the documentation for the `Applet` class you will find that it is listed as having around a couple of dozen associated methods. Each of these will define a default behaviour for an `Applet` and each can be replaced<sup>43</sup> in a derived class if some special behaviour is needed. However these two-dozen methods are very far from being the whole story. For instance `paint` is not listed among them. This is because `Applet` is descended from `java.awt.Panel` which in turn is derived from `java.awt.Container` which itself inherits from `java.awt.Component` and `java.lang.Object`. Each of these super-classes define (often many) methods of their own. The lower-down ones sometimes replace a few of the higher level methods with more specialised versions, but they also tend to provide lots of new methods of their own. Thus in this case the `paint` method is defined as an aspect of a `Container`, and is only part of `Applet` via inheritance. The end effect is that something that is as easy to get started with as an `Applet` in fact comes complete with perhaps hundreds of bits of pre-defined behaviour almost any of which can be adjusted by the simple expedient of overriding some method.

Sometimes of course this arrangement whereby library facilities are structured into hierarchies of classes means that the very simplest thing one might want to do involves explicit construction of objects from various classes in a way that looks less smooth. To print simple text as the output from a simple Java stand-alone application one can invoke `System.out.println`. The long name is because `System` is a class (its full name is `java.lang.System`), and `out` is then a variable in that class. The field `out` has as its type `PrintStream` and the class `PrintStream` provides a method called `println`. It is possible to reference the variable `out` just by giving its class (without having to have a variable whose type is that class) because it was defined as being `static`. The recipe as typed in by the programmer is not too bulky but the full explanation of why it works is a bit clumsy. “Simple” input is if anything worse. There is a static variable `System.in` which is of type `InputStream`, and for an application to accept input from the keyboard one needs to use it. However the class `InputStream` only provides the most basic reading functions, and various derived classes are needed if flexible, efficient and convenient reading is to occur. A suggested protocol for a single

---

<sup>43</sup>The fuller story is that any member of a class that has been marked as `final` can not be redefined in a derived class. The use of `final` thus provides the designer of a class with a way to guarantee some aspects of class behaviour even in derived classes.

integer from the standard input ends up something like

```
BufferedReader in =
    new BufferedReader(
        new InputStreamReader(System.in),
        1);
int n;
try
{   n = Integer.parseInt(in.readLine());
}
catch (IOException e)
{   n = -1; }
catch (NumberFormatException e)
{   n = 0; }
System.out.println("I got: " + n + "....");
```

This creates an `InputStreamReader` out of `System.in`, and then builds from that a `BufferedReader` where here I have indicated that a buffer size of 1 should be used. For reading directly from the keyboard a ridiculously small buffer size means that the program gets characters as soon as they are available. If the “, 1” was omitted the `BufferedReader` would use some default buffer size and you would have to have keyed in that many characters before anything ever happened! The `BufferedReader` class then provides a `readLine` method, and the string that it returns can be interpreted as an integer by the static method `parseInt` in the `Integer` class. Both `readLine` and `parseInt` may raise exceptions if anything goes wrong, and so a proper program should be prepared to handle these. The above tends to look very heavy-handed because “real” programs will generally want to decode much more complicated input than just the single number shown above, and will really need to put in the `catch` clauses so that they can respond cleanly to erroneous input. Even the buffering control is really quite important — direct keyboard input may need to be unbuffered so that interaction works well while input of large amounts of input from a file may be *much* faster if buffering is used.

Java in fact provides another rather larger class than `BufferedReader` which may be useful in many applications that want to accept free-format input. This is the class `java.io.StreamTokenizer`<sup>44</sup> which can help you read in a mixture of numbers and words. Here is a demonstration:

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

---

<sup>44</sup>Actually I think that `StreamTokenizer` is very useful while you are getting started, but although it can be customised quite substantially it is not flexible enough for most really serious uses. In the Compiler Construction course in Part IB you may learn about a package called `JLex` that is harder to set up but which provides enormously more power and flexibility.

```

...

StreamTokenizer in =
    new StreamTokenizer(
        new BufferedReader(
            new InputStreamReader(System.in),
            1));
in.eolIsSignificant(true); // see newlines
in.ordinaryChar('/');      // '/' is not special
in.slashSlashComments(true); // '/' for comment
try
{   int type;
// The next line loops reading tokens until end of file.
    while ((type = in.nextToken()) !=
            StreamTokenizer.TT_EOF)
        {   switch (type)
            {
// There are a number of predefined "token types" in
// StreamTokenizer, so I process each of them.
                case StreamTokenizer.TT_WORD:
                    System.out.println("word " + in.sval);
// If the user says "quit" then do so. NB "break" only
// exits the switch statement here.
                    if (in.sval.equalsIgnoreCase("quit"))
                        break;
                    continue;
// in.sval and in.nval get set when string or numeric
// tokens are parsed and contain the value.
                case StreamTokenizer.TT_NUMBER:
                    System.out.println("number " + in.nval);
                    continue;
// the method lineno() tells us which line we are on.
                case StreamTokenizer.TT_EOL:
                    System.out.println("start of line " +
                                        in.lineno());

                    continue;
// quotes and doublequotes contain strings.
                case '\\': // drop through
                case '\"':
                    System.out.println("string " + in.sval);
                    continue;
// Other characters end up here. Eg +, - etc.
                default:

```

```

        System.out.println("sym " + (char)type);
        continue;
    }
    break; // here if "quit" typed in
}
}
catch (IOException e)
{   System.out.println("IO exception");
}
}

```

The level of complexity here seems much more reasonable! The initial code that sets up a `StreamTokenizer` is not very different from that which set up the simpler buffered stream before, and is clearly a small overhead to pay to be able to have Java split your input up into words and numbers. The `StreamTokenizer` provides methods that allow you to customise its behaviour so that it can recognise one of several possible styles of comments and accept various string delimiters. The calls

```

in.eolIsSignificant(true); // see newlines
in.ordinaryChar('/');      // '/' is not special
in.slashSlashComments(true); // '//' for comment

```

illustrate a little of this. The first call tells the tokenizer that newlines should be returned to the caller. By default they are counted as whitespace and so not passed back. The second call makes a single / into an ordinary character, where by default it introduces a comment if followed by a second / or a \*. The final line enables recognition of comments that are started by //. As always you need to browse the full documentation to discover what all the other options are!

Two lessons emerge. The first is that the bigger and more powerful classes in the Java libraries may really save you time if you find out how to use them, while direct use of very low level facilities may end up feeling pretty clumsy. The other is that these high level facilities are often very flexible, but if you need some feature that they do not support you may have to drop down a level. For instance `StreamTokenizer` does not know how to handle numbers expressed in hexadecimal or octal, and it always reads numbers in type `double` which is not good enough if what you needed was a `long` value.

## 4.6.2 Name-spaces and classes

When you derive one class from another it is sometimes desirable if the methods and fields of the base class are visible in the derived one, but in other cases it may not be. This aspect of name visibility needs to be considered in conjunction with the consequences of classes falling into different packages. Java confronts all this by defining four levels of name visibility within classes:





Figure 4.6: Classes and inheritance are a sort of magic.

`private`: is the most restrictive one. A method or variable that has been declared as `private` can be referenced from within the class in which it is defined, but not from anywhere else. In particular code that is in another class can not see it regardless of whether the other class is in the same package as or was derived from the original one;

`package`: relaxes things so that code in any class that is in the same package can reference a value. This is the default arrangement, and is indicated by *not* using any of the other visibility qualifiers. Note that the keyword `package` is used at the head of a file to specify which package that class will reside in, and it is not valid in method or variable declarations;

`protected`: When a name is declared as `protected` it becomes visible in derived classes even if they are in other packages. Because during this first course you will probably not be creating new packages yourself this case will mostly be relevant where a library class has some `protected` members and you derive a few class from it. Your class will probably be in the default package but despite that you will be able to access the members involved;

`public`: is the final case, and it makes names generally available regardless of packages and inheritance.

It seems tidy to document the other possible qualifiers for declarations here, even though they are not concerned with name visibility. Indeed their consequences are rather mixed, and since this is a first Java course it is not essential to be fully comfortable with them all.

`final`: When a variable is declared `final` nobody will be allowed to assign a new value to it. When a method is `final` then it can not be overridden in any derived class. In both cases the effect is to make the definition in its visible form the one that can be relied upon everywhere else;

`static`: The default situation for items defined within classes is that the items only come into existence when an object of the class-type is created. This makes obvious sense for data fields. For instance after the declaration

```
class IntList
{   public int head;
    public IntList tail;
}
```

it is clear that the only context in which the `head` and `tail` fields can be used is in association with an object of type `IntList` as in

```

int sum(IntList x)
{
    int r = 0;
    while (x != null)
    {
        r += x.head;
        x = x.tail;
    }
}

```

For consistency the same access rule is then applied to member functions (ie methods) in a class. If however an item in a class has been declared `static` it is as if a single globally allocated instance of the class gets created automatically, and the field can then be referred to relative to just the class name. For instance (a nonsense code fragment!)

```

class MyConsts
{
    static final double ZETA2 =
        1.6449340668482264365;
    static final double CATALAN =
        0.91596559417721901505;
    static int square(int x)
    {
        return x*x;
    }
}
...
double a = MyConsts.CATALAN -
           Myconstcs.ZETA2 +
           (double)MyConsts.square(1729);
...

```

**abstract:** Sometimes it is useful to define a base class not because it is useful as such, but because the various other classes that get derived from it might be. Consider the ML declaration

```

datatype option = A of int | B of double;

```

One way of producing a Java equivalent would be to start by defining a rather vacuous class called `Option` and then deriving from it two new classes one to correspond to each of the two cases in the ML version:

```

abstract class Option
{
}
class OptionA extends Option
{
    int a;
}

```

```
class OptionB extends Option
{   double d;
}
```

The base class here only exists to be extended, and it would be silly to create an object that was of that type<sup>45</sup>. The qualifier `abstract` prevents anybody from creating objects of the base class. It marks things that must be inherited from before meaningful use can be made of them. In cases such as this it is often useful to discriminate as to which derived class a particular instance belongs to. The `instanceof` operator can be used to do this. Again my illustrative code is artificial:

```
Option x = new OptionA(); // or maybe OptionB?
...
if (x instanceof OptionB) ...
else ...
```

It is very often neater and easier to define different overridings of a common (abstract) method in the two derived classes so that the correct behaviour is achieved for each. If that is done<sup>46</sup> the `if` statement and use of `instanceof` could be replaced by a simple call to the method concerned. It is of course not essential to make a base class in such examples `abstract`, but doing so prevents any possible embarrassment if some code created an instance of it in its raw and useless form, so it is generally considered to be good style.

**native:** If a method is defined as `native` then Java somehow expects there to be an implementation of it that was coded in some language other than Java. This can be used by system builders to interface Java code down to lower level and perhaps machine-specific system calls, but will not be discussed further in these notes.

**synchronized:** related to Java code where several threads of computation may be active at once. Although the very basic aspects of this will be covered in this course a proper treatment needs to wait until you have had a Part IB course on concurrent systems.

**interface:** The keyword `interface` is not a modifier for use in class definitions but a keyword whose use is very much like that of `class`. An interface

---

<sup>45</sup>Of course objects of type `OptionA` and `OptionB` are also of type `Option`, so what I mean is it would be silly to go `new Option()`.

<sup>46</sup>A similar stylistic issue arises in ML where user of pattern-matching in function definitions can often reduce the number of explicit `if` statements that have to be written.

can be declared much as an abstract class is. Classes can be defined to extend other classes, but a restriction that Java applies is that a new class can only be an extension of a single parent class. Interfaces provide an approximation to being able to extend several parent classes — a new class can specify that it `implements` one or more interfaces. When a class indicates that it will implement an interface it has to contain (concrete) definitions of all the (abstract) methods that the interface specifies.

At (very) long last we have covered all the magic that arose in the initial `Hello.java` program and can see what each keyword present there was indicating.

### 4.6.3 Program development with classes

In Java, as in other Object Oriented languages, the whole shape of a large program needs to be designed in terms of terms of the packages and classes that will be built. It is worth putting particularly careful thought into the way in which hierarchies of classes will be derived from one another via inheritance.

There are two application areas that were pioneers in illustrating the benefits and strengths of object oriented programming (which is what this is). It can thus be worthwhile considering examples of these as some of the earliest ones you work with when getting used to the idiom. The first application area was that of simulation<sup>47</sup>, while the other was graphics and especially the display of geometric figures in windows. The following example, which is taken from *Java in a Nutshell* and shows how use of several classes rather than just one may allow the programmer to keep distinct aspects of their task separate. But doing this the size of unit that has to be debugged is reduced, and the possibility of re-using parts of the code later on in another project is increased. The example supposes that a graphical design and modelling package is being written. Within it it will keep data-structures that represent circles, squares and other shapes. For much of its time it will work on these busily computing their areas, their circumferences, whether they intersect and similar properties. It may also adjust their sizes and positions. As well as performing all these calculations the complete package will also have a user interface that can draw the objects. There will be options to control the colour of each individual circle (and so on) as well as to determine whether the items are drawn just as outline figures or as filled-in shapes.

Without use of inheritance and thus without serious use of the Java class mechanism the code would probably have to consist of a single class, say called `Shape`, which would contain a master variable indicating what sort of shape was involved,

---

<sup>47</sup>Indeed the way that object-oriented C++ developed from the simpler language C was initially specifically for use in this area.

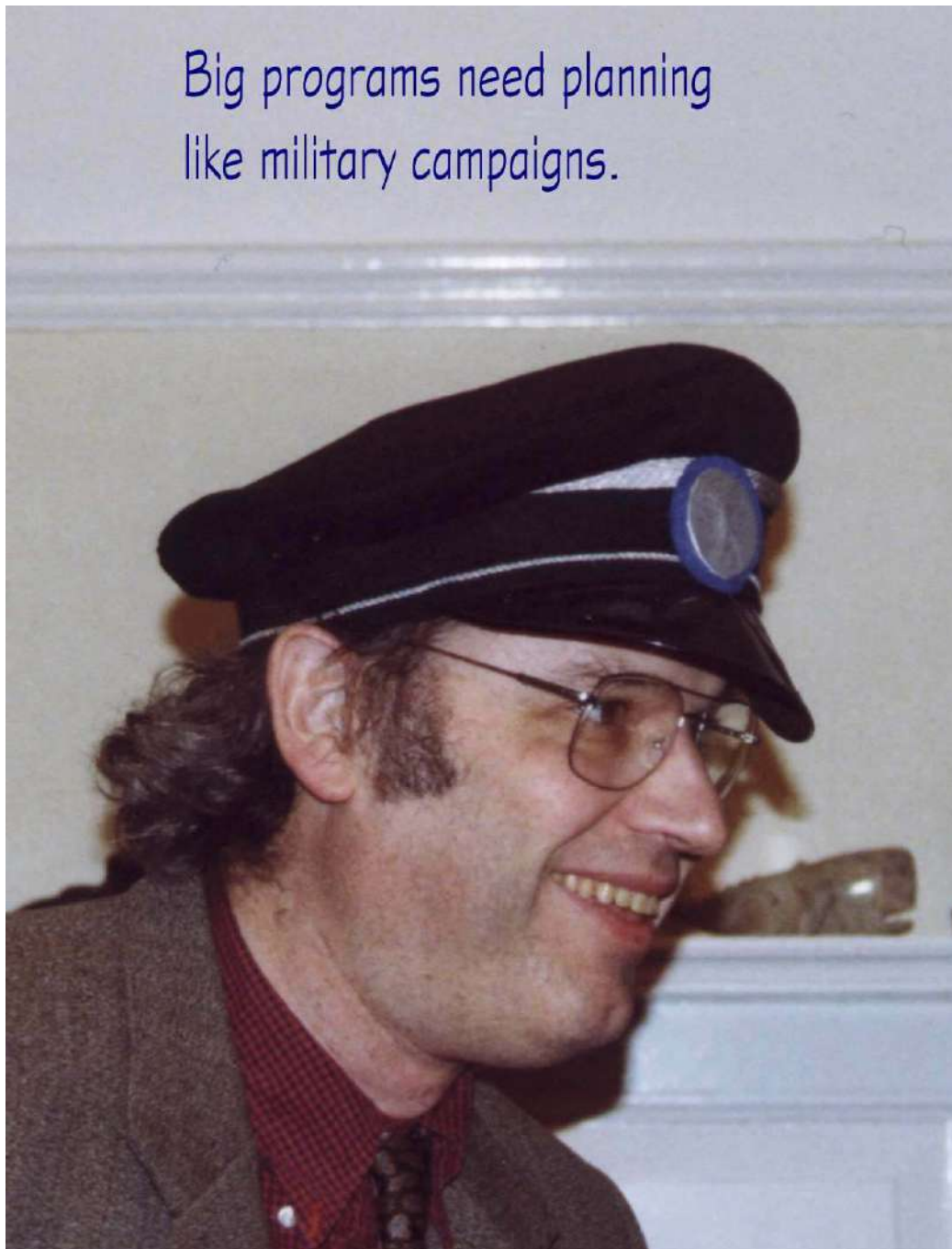


Figure 4.7: See also the “Software Engineering” courses.

then other variables that could be used to specify the exact parameters of that shape (eg its radius if it was a circle). The method functions such as `area` would need to dispatch on the type of the figure and do different calculations in each case. Further code would arrange to be able to draw pictures to represent the data. All the geometric and graphical parts of the code would be in the same class and thus the same source file — something which would not cause trouble in tiny cases but would become clumsy for a fully elaborated version.

With inheritance it would be natural to start with a basic class (again I will call it `Shape`) which will probably be `abstract`. Its purpose is to allow the program to declare variable of type `Shape` and then store circles, squares, stars and all other possible sorts of shape in that single sort of variable. The methods declared for `Shape` can be given as just declarations, rather than as full definitions:

```
public abstract class Shape
{
    public abstract double area();
    public abstract double circumference();
}
```

which makes these methods available in any object of class `Shape` but expects that concrete variants on the class will provide the real implementations.

For each sub-class of `Shape` a new class could then be derived:

```
class Circle extends Shape
{
    protected double radius;
    public Circle() { radius = 1.0; }
    public area() { return Math.PI*radius*radius; }
    public circumference()
    { return 2.0*Maths.PI*radius; }
    public double getRadius() { return radius; }
}
```

Note that this can introduce new public members that are not relevant for general `Shape` quantities, but which do make sense when you know you have a `Circle`.

Next an `interface` would be set up, defining the methods relevant for drawing<sup>48</sup> things on the screen:

---

<sup>48</sup>Java is an American language, and so the character of being Red, White or Blue is `Color` rather than `Colour`. Given that the library uses this spelling it seems best to swallow nationalistic pride and adopt it elsewhere in the code...

```
public interface Drawable
{
    public void setColor(Color c);
    public void draw(Graphics g);
    // etc etc.
}
```

Now it is reasonable to derive a new class for a version of each sort of shape but in a form that supports the drawing operations:

```
class DrawableCircle extends Circle
    implements Drawable
{
    Color c;
    public void setColor(Color c)
    {   this.c = c; }
    public void draw(Graphics g)
    {   ... // whatever, maybe
        g.drawOval(...);
    }
    // etc etc.
}
```

It is now possible to use the drawing methods as well as the data manipulation methods in one of these ultimate data-structures.

Often when producing a derived class and overriding a method the newly extended method needs to use the corresponding operation from its parent class. For instance if a class defines a method that is used to initialise its variables then a derived class may add extra variables that need initial values too, but it would be clumsy to insist that it also had to repeat all the code to setup the variables in the base class. And indeed if some of those were `private` or `protected` it might not be able to. The solution is hidden in the keyword `super`. This is a bit like `this` in that it always refers to the current object, but it views it as a member of the immediate parent class. Thus code like

```
class SubClass extends MyClass
{   private int variable;
    public void init()
    {   super.init(); // init as a MyClass
        variable = -1; // finish off as SubClass
    }
}
```

and the word `super` is only of relevance when extending a class and overriding methods. In the case of some library classes and methods the documentation will



explain to you that you must use it, see for instance the method `paint` in the class `Container`.

## 4.7 Generics

The material here is now for Java 1.5 and I expect my coverage of it to grow over the next year or so. This year I will do hardly more than just mention it and let Part Ibb coverage consider filling in the gaps. This seems especially reasonable since textbooks that catch up with this are still somewhat rare.

In ML you got used to having types that were polymorphic. For instance a sort function that took a predicate and a list might have had type

$$(\alpha * \alpha \rightarrow \text{bool}) * \alpha \text{list} \rightarrow \alpha \text{list}$$

to indicate that the elements of the input and output lists had the same type and the ordering predicate was compatible with that. A particular feature of ML to recall is the availability of parameterised types such as  $\alpha \text{list}$ . In Java instead of saying “type” we will say “class”, and instead of saying “polymorphic” we say “generic”. A generic class is established by putting type variables within angle brackets. You can then use the type variable within the class as if it were a regular type name: small

```
class MyClass<E>
{
    E myMethod(E arg1, int arg2)
    {   MyClass<String> newvar = ...
        ...
    }
}
```

With your ML experience of polymorphism you can now probably see at once how to use this capability to write implementations of various generic data structures (trees, lists and the like) and provide useful functions that traverse, search or sort them. In fact that Java libraries have done a great deal of that in their so-called Collection classes.

In ML polymorphism is all-or-nothing. If you have a type-variable  $\alpha$  it can stand for absolutely any ML type. To improve security you may sometimes like to have a way of expressing more limited flexibility (eg generic over all sorts of numbers, but not over non-numeric data). Java provides a capability using a type wildcard written as question mark, and can limit the range of the wildcard using notation like

```
public void sum(List<? extends Number> arg)
{   for (n:arg)
    {   ...   }
}
```

Here the `sum` method takes an argument that is some sort of `List`<sup>49</sup> but it insists that the polymorphism that `List` provides has been used in a way that means you know that all the objects in the list are some subclass on `Number`.

You will use generics every time you use the Java Collection Classes. You can use it in your own code too. There is a fair amount more that I could say about exactly how it interacts with the type-hierarchy that class inheritance provides and when a generic class is a sub-class of another, but I believe that the details there do not belong in a *first* Java course!

## 4.7.1 Exercises

### Objects everywhere

The Java libraries make extensive use of classes in hierarchies (and also a more modest number of interfaces). The arrangement in the basic set of classes is that *everything* is ultimately descended from a base class called `Object`. The most immediate consequence is that an object of *any* class from the basic libraries may be stored in a variable of type `Object`. It is exactly as if whenever you define a new class and do not give an explicit `extends` clause as part of its definition Java just sticks in “`extends Object`” for you. Of course when you extend some other class it in turn will somehow have `Object` as an ancestor-class so this way as previously stated *every* instance of *any* class is an `Object`.

A few basic methods are defined for `Object`, of which perhaps the most interesting at present is `getClass` which returns an thing from the class `Class`. If `x` is any `Object` then `x.getClass().getName()` is a string that is the name of the class of `x`! The general parts of the Java libraries that allow you to investigate the classes that `Objects` belong to and then retrieve lists of the variables and methods that they provide are referred to as *Reflection*: as it were a Java program can look at itself as if in a mirror.

Check the documentation and write Java code that accepts an `Object` and prints out as detailed and as readable description of it as you reasonably can.

Note that `Object` underpins the polymorphism of Java generics, but now that generics are available programmers will use `Object` directly much less than they used to.

---

<sup>49</sup>A Collection class that does just what you expect!

**Primitive is second class?**

The ability to treat things as “Objects” does not (directly) extend to the Java primitive types. To work around that the libraries contain classes with names that are rather like those of the primitive types except that they are capitalised. Ie `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float` and `Double`. As of the most recent revision of Java you will find that the compiler arranges to convert between `int` and `Integer` (and the other primitive types and their associated wrapper classes) when it believes that that will help you. The conversion naturally involves some run-time cost so it is perhaps advisable to be aware when it happens. The sort of circumstance where it is especially convenient that this happens is when you want to store a primitive object (eg an integer, floating point number or character) in a `HashMap` or a `Vector` (or indeed any of the collection classes).

It was then natural for the Java designers to set methods associated with these to implement a wide range of basic conversions and tests on the values, as in `Integer.doubleValue` and `Double.isNaN` (and many more).

The numeric types the classes `Integer` etc do not inherit directly from `Object` but via a class called `Number` Eg

```
Number a, b;
a = 2;      // new Integer(2);
b = 11.7;  // new Double(11.7);
```

can be written as shown, but behaves as if the constructors in the comments have been used. If the `Number` objects are used in a context where primitive numbers are needed (eg you try to perform arithmetic using them) the values will be unpackaged for you.

Write a class that defines lists of `Numbers`, with suitable set of facilities for constructing such lists and a method `sum` that can add up the values in a list returning the result as a `double`. You may need to use “`x instanceof Integer`” to sort out which flavour of number is present in some particular node.

**Some text output using Objects**

Since `Object` is an almost universal type it can be used to pass arbitrary data to a function. This is in fact what happens with `printf`, but one extra thing happens there. If a method is declared with three dots after the type at the end of its argument list, as in

```
PrintStream printf(String format, Object... args)
{ // whatever definition you need
}
```

indicates that the final argument to `printf` will actually be passed as an array of `Object` values. But the *calls* to it will just appear to permit a variable number of arguments, and each argument will be converted to (if a primitive type) or interpreted as (if a class type) `Object`.

While this scheme can be used in your own code to support variable numbers of arguments, and it can also be used with more restrictive types than `Object` it will almost always count as poor style since it can easily reduce type-safety and cause confusion if you mix it with method overloading. But where it is useful it really helps make code concise. Without it instead of writing

```
int i=1, j=2;
System.out.printf("%d, %d", i, j);
```

you would need to wrap `i` and `j` up in the type `Integer` explicitly, and create an array to pass the multiple arguments explicitly.

```
int i=1, j=2;
myOwnMethod("%d, %d",
    new Object [] {new Integer(i), new Integer(j)});
```

But note very well that within the code that implements things such as `printf` everything has to work understanding that the concise calls are in fact mapped by the java compiler onto the clumsy looking code that packaged up primitive types and makes an array.

Now seems a good time to provide a summary of more of the formatting options available with `printf`, and also to note that the method `String.format` does exactly the same job of layout but returns a formatted string rather than doing any direct printing. We have already seen `"%d"` for laying out integers, and know that `"%n"` generates a newline.

Within a format string the character `"%"` introduces a format specifier. After the percent sign a number of optional elements can appear:

- An argument index followed by a dollar sign (`$`). Without one of these the values to be converted are taken one at a time from the arguments provided. An index such as `(2$)` tells the formatter to use the second argument now, even if that is out of order. Often you may want to display the same data several times, eg in different formats. In that case `(<)` is very useful: it tells the system to re-use the argument most recently dealt with;
- Some flag characters. Just what is valid here will depend on just what sort of layout is being performed, but various punctuation marks as flags can, for instance, force left-justification of text within a field (`-`), ensure that numbers are always displayed with an explicit sign (`+`), include leading zeros (`0`) or be more fussy about the actual types of arguments (`#`). You need to check fine details in the documentation when you use flags!

- a field-width, written as an unsigned non-zero integer. You should expect that if this is specified that the output from the conversion will have exactly that number of characters;
- A dot followed by a integer precision. Eg (.4). For some conversions this sets an upper limit on the number of characters to be generated. For floating point conversions it controls either significant figures or the number of digits after the decimal point.
- (and finally!) a character (or in some cases a pair of characters) that indicated just what sort of conversion is to be performed. Perhaps the more important cases are the letters *s*, each of which is discussed briefly below!

The full set of format letters and options can be found in the online documentation, but key cases are

- s, S:** This takes any value at all and tried to convert it] to a string. If the argument implements the `Formattable` interface then its `formatTo` method is used to do the conversion, otherwise its `smalltoString` method is used. When you define a class of your own you may often wish to override or define one or both of these methods so that you can easily print instances of your class. Many of the Java library classes implement these methods in ways that at least try to be helpful. If you write a capital *S* the material that is displayed is forced into upper case. Similar effects apply for other use of upper case format letters;
- d:** This is the case most often seen in these notes so far, and prints an integer. But you can also display `BigInteger` values with this (and the *x*) format;
- x, X:** Integers can be displayed in hexadecimal rather than normal decimal notation this way;
- c:** character
- e, f, g, E, F, G:** Floating point and their display involve lots of complication! The “e” formats always use scientific notation with an explicit exponent. The “f” formats use the specified precision as the number of digits to display after the decimal point (eg it is a good thing to use for printing pounds and pence with “%.2f”), while “g” tries to select between those two formats to select one that will be natural and will take up as little space as possible.
- %:** If you want to print a percent sign you will need to write two in a row!

**n:** Unix and Windows have different ideas about what constitutes a “newline”. The format code `%n` makes allowance for that for you.

**tx:** Java provides an amazingly rich range of ways of formatting times and dates. You can use these formats when printing objects of type `Long`, `Calendar` or `Date`. I think there is too much to list here, but a very few of the options available are

**tY** year displayed as 4 digits, eg “2005”;

**tA** full name of day of the week, eg “Monday”;

**TA** as above, but upper case: “MONDAY”;

**ta** short name of day: “Mon”;

**tM** minute within the hour, as 2 digits;

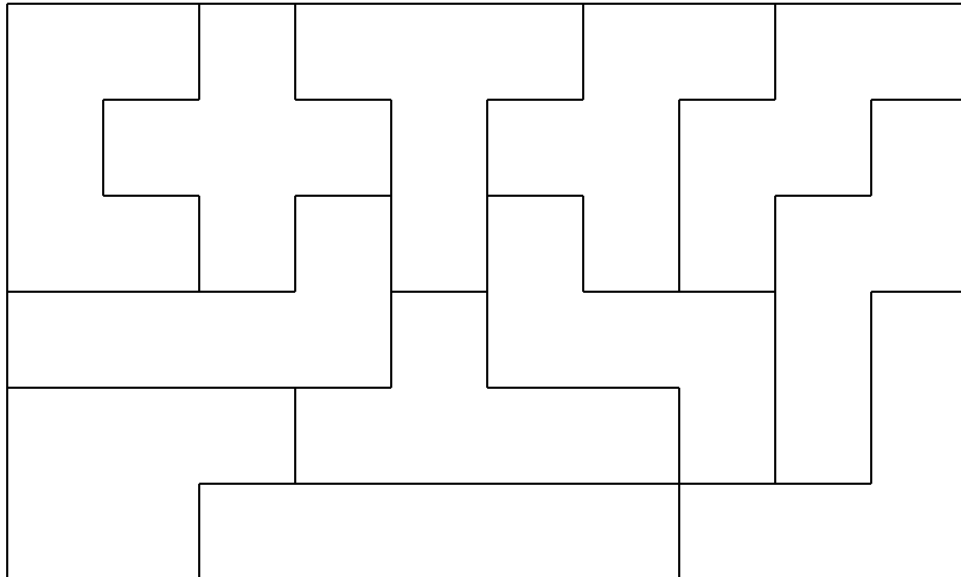
**tm** number of the month as 2 digits, counting January as number 1;

**tT** time formatted for the 24-hour clock as “%tH:%tM:%tS”;

**tD** date formatted as “%tm/%td/%ty”.

### A bigger exercise

There are twelve shapes that can be made by joining five squares together by their edges to get a connected unit. It is possible to pack these shapes (the *pentominoes*) into a six by ten<sup>50</sup> rectangle in a number of ways. Here is one such packing, which will also serve to show you the shapes of all the pieces:



<sup>50</sup>also into a five by twelve or three by twenty.

The object of this exercise is to find other solutions to the puzzle.

The suggested strategy is to represent the 10 by 6 board using 60 of the 64 bits in a `long`. You can then treat these as if they are arranged as a rectangular array, and then a single `long` value can represent a possible position of a piece. In this representation the twelve pieces can be described by the array:

```
final long [] rawPieces =
{   0x000001f, 0x0100407, 0x000040f, 0x0300403,
    0x0401c01, 0x2008007, 0x0201c02, 0x0000c07,
    0x0301808, 0x0000c0e, 0x000100f, 0x0301802
};
```

where the values look pretty ugly but are at least all in quite a small table. A bulkier but perhaps cleaner way to set up the initial table of shapes would be to use a function such as:

```
long piece(String line1, String line2, String line3)
{   return (row(line1) << 2*boardWidth) |
        (row(line2) << boardWidth) |
        row(line3);
}
long row(String line)
{   long r;
    for (int i=0; i<line.length(); i++)
    {   r <<= 1;
        if (line.charAt(i) == 'X') r |= 1;
    }
}
...
piece("X  ",
      "XXX",
      " X ");
```

The `init` method for the applet should start by setting up a table first of all the twelve pieces normalised so that they are down in one corner of the board, and then a larger table showing each piece in every location on the board that it could possibly be. Doing this will involve writing code that reflects and rotates pieces — not especially nice when using bits packed into a `long` — and which avoids setting up entries that are redundant because of symmetry. The code that is involved in getting this far is quite messy enough to keep you busy for a while.

The overall structure of the code that searches for solutions might then be

```
// search(i) looks for ways of placing piece
// i on the board. The array entry maps[i][j]
// is a bitmap showing the j-th place that
// piece i could bit, and the variable "board"
// shows which parts of the board have already
// been filled. There are 12 pieces, known as
// 0 to 11.

void search(int i, long board)
{   if (i == 12)
    {   // Here a solution has been found
        ... record it somehow ...
        return;
    }
    for (int trial=0; trial<maps[i].length; trial++)
    {   if ((maps[i][trial] & board) == 0)
        {   // no overlap with existing pieces
            // so put this in and next try to
            // fit in piece i+1.
            search(i+1, board | maps[i][trial]);
        }
    }
}
```

The first challenge would be just to count the solutions, and so the place in the above which is incomplete could be replaced by a single statement that incremented a variable. But since it is easy to use `fillRect` to draw filled-in rectangles in Java it would seem natural to try to draw some of the solutions and that would mean doing something distinctly harder.

The `search` function I have sketched tries the twelve pieces one after each other, and at each stage considers each piece at every position on the board where it would still fit. A different search strategy would be to scan the board at each stage and find the first vacant square. The program would then identify and try every piece that could be used to fill in that square. I believe that this second search strategy is rather closer to the one most people would use than my original one was.

### A curve to plot

Use Java to plot a picture of the following curve as  $t$  varies from 0 to  $2\pi$ :

$$\begin{aligned}x &= \cos(t)(1 + \cos(40t)) \\ y &= \sin(t) + \cos(t) \sin(40t)\end{aligned}$$



Find a copy of *A Book of Curves*, E. H. Lockwood, Cambridge 1963, and in a similar style re-create variants on as many of the pictures as you can.

### Reading hexadecimal numbers

We have seen various ways of decoding numeric input, eg `Integer.parseInt` and the whole set of joys associated with `StreamTokenizer`. You can note from the full documentation that there is a two-argument version of `parseInt` that allows you to specify what radix the input string was supposed to be in. You may also like to check details of the class `SmallScanner` which has a method `nextInt` that can also accept an argument indicating what radix to read in.

Now imagine that these facilities did not exist, or that for some strange reason you could not use them. Implement your own functions that can be given strings as arguments and which will make it possible to convert the strings into `int` and `long` values, allowing for the possibility of octal or hexadecimal specifications.

### Displaying floating point numbers

In versions of Java prior to 1.5/5.0 the functionality of `printf` was not available. This exercise is to re-create some of it thereby getting a chance to feel what work is involved in making worthwhile extensions to the existing libraries.

The method `Double.toString` allows you to generate a printable representation of a floating point number. However compared to the floating point layout flexibility available in many other languages it seems pathetically simple-minded. A typical programming language will provide for three ways of printing floating point values:

F format: here numbers are written as illustrated in the following examples

```
-1.000  
1234567890000000000.0  
0.000000005656
```

and even if the values are very large or very small their magnitude is indicated by having suitable numbers of leading or trailing zeros. It is typically possible to specify how many digits will be printed following the decimal point, and to indicate the width that the whole number will be padded to with either leading or trailing blanks.

E format: For very large or small numbers it may be convenient to use scientific notation. So with *E* format an explicit exponent will always be displayed:

```
-1.0e000
1.234568E018
5.656000e-009
```

Observe that there is always exactly one digit before the decimal point (sometimes a scaling option is provided to allow the user to specify a different number of digits before the point), and the exponent is always present and probably always displayed in a way where the largest possible exponent value could be fitted in. A “precision” specifier can indicate how many significant digits are to be shown, and the number will be padded with zeros or rounded to meet that requirement. Numbers close to 1.0 tend to look a bit ugly this way! Again it is useful to be able to place the number in a fixed-width field, either right or left-justified.

**G format:** Large numbers are best shown in *E* format while modest size ones do best in *F*. So *G* is a composite scheme that looks at the value of a number and decides which of the other two formats would lead to the most compact representation, and it then uses that. It is roughly what Java’s `Double.toString` method provides, but again we would really like options to indicate precision and field width.

Implement functions which convert Java `double` values to strings in each of the above formats.

I might suggest that you start by using `toString` to do the basic conversion and then let your code restrict its worry to unpicking that string and re-formatting the characters. If you decide you want to do the numeric to string conversion from scratch you should be aware that preserving numeric accuracy is quite hard!

Write a test-suite that compares the strings your code generates with the ones that `String.format` produces. Then worry about NaNs, infinities, careful rounding and the like!

### Double as bit-patterns

`Double.doubleToLongBits` takes a `double` as an argument and returns a `long`. The `long` is the internal IEEE-format bit-pattern that represents the `double`. The matching function `longBitsToDouble` accepts a `long` and manufactures a `double` in the same dubious sort of way. Investigate whether there is any `double` value `x` in Java so that

```
(double)Double.doubleToLongBits(x) == x
```

**Continued fractions**

Any positive number can be expanded as a *continued fraction* as in

$$x = x_0 + \frac{1}{x_1 + \frac{1}{x_2 + \dots}}$$

where the values  $x_i$  are called *partial quotients* and are all positive integers. If the original number is rational the continued fraction terminates at some stage. Otherwise it goes on for ever, and can be viewed as providing an alternative to the usual decimal expansion of numbers. Instead of writing a value as say 1.414213562... the partial quotients would be listed [1, 2, 2, 2, ...]. Gosh in fact for this number it looks as if the continued fraction is astonishingly regular!

The sequence of partial quotients in the expansion of a number are easy to compute - the first is just obtained by casting the number to an `int`. The rest can be obtained from the reciprocal of what you get by subtracting that value from the original number. Write code to do this and tabulate the first dozen partial quotients you get in the expansions of the following numbers:

$$\sqrt{3}$$

$$(\sqrt{5} + 1)/2$$

$$\sqrt{7}$$

$$e = 2.71828\dots$$

$$\pi$$

**4.8 Important features of the class libraries**

The coverage thus far has shown the use of some small parts of the Java libraries, but has also missed a great deal out. In this course I will not have anything like enough time to describe everything that is available. However there are a few bits of functionality that either seem to be generally useful enough or sufficiently fun to be worth covering. The little bits of explanation given here are thus to be viewed as a sampler of what Java can do for you. If you can work through all these demonstrations and navigate the documentation of the classes that they introduce you should have got a reasonably broad idea of the system, and in looking up the documentation details while working on these cases you will as a side-effect be noticing what other classes are present. I will only give the most basic possible demonstrations of the things illustrated here. Full competent use of them can only come with serious work on rather larger bodies of code. I will also totally ignore

several of the newer parts of the Java class libraries, or to be more precise, I will leave fuller details of some of these facilities and of the other ones to next year's "Concurrent Systems and Applications" course and/or your own private study.

### 4.8.1 File input and output

The character input and output shown so far has used the pre-defined "standard" streams `System.in` and `System.out`. Obviously in many real applications it is necessary to access named files.

In many programming languages there is a logical (and practical) distinction made between files that will contain text and those which will be used to hold binary information. Files processed as binary are thought of as sequences of 8-bit bytes, while ones containing text model sequences of characters. In Java this distinction has two main manifestations, one of which is somewhat frivolous but can matter on an every-day basis in the UK while the other is of wider importance but will not impinge on immediate coursework:

1. Windows and some internet protocols use a pair of characters, carriage-return and line-feed, to mark the end of a line. Unix and Linux use a single character (newline). In text mode Java makes whatever adjustments are needed so that external files adhere to platform conventions, but your Java code sees just the character `'\n'`.
2. In many parts of the world (and in particular in the Far East) text documents need to cope with alphabets that involve many thousands of symbols. Unicode is designed to be able to cope with these, but there can be a variety of ways of encoding text as streams of bytes. When working in such an environment Java can be configured so it knows how to pack and unpack Unicode using various of the major encoding conventions. But obviously it will only even try to do this when it knows that the programmer wants data to be viewed as character-based rather than binary.

Java uses names typically based on the word `Stream` for binary access, and `Reader` and `Writer` for text. So when you read the documentation expect to find two broadly parallel sets of mechanisms, one for each style of access!

Java input and output can seem clumsy to start with because almost all of the functions involved are able to throw exceptions, and it is expected that Java code using them should be prepared to handle these. This is in fact *good* because experience with earlier languages indicates that most programmers do not find it easy or natural to put error-checks after simple I/O operations, even though logically almost any of them could fail. For instance writing a single character

to a file could unexpectedly fail if the disc on which the file lived became full<sup>51</sup>, or if it was on a floppy disc and the disc was removed from the drive or had a scratch, or if there was a hardware glitch in a disc interface. Different but equally delicate issues arise with output that goes directly to a printer, across a network to a remote file-system, or with input from a semi-reliable device such as a bar-code scanner. The Java use of exceptions encourages all programmers to consider I/O failure right from the start.

There is one final complication about Java input and output that ought to be mentioned up front. One use of Java is in applets to be embedded within web pages, and hence sometimes fetched from remote web-sites. It could be bad if code from an untrusted site could read and write all your files! So Java introduces the idea of a security manager and can classify code as either trusted or untrusted. Untrusted code will not be permitted to access the local filing system. The short form way around this is to make everything you do an application not an applet: security restrictions are then (by default) not imposed. If you do need to make applets that access disc or do other things that default controls lock out you need to impose security on an application then you will need to find out about the creation of custom Security Policies and signed Java code. I will not describe that here.

Java provides a rich and somewhat elaborate set of capabilities, but perhaps a good place to start will be simple reading of text files. The class `FileReader` does almost everything you are liable to need: here is a minimal demonstration that shows that you can use a method called `read` to read characters, and that it returns the integer `-1` at end of file.

```
import java.io.*;

public class ReadDemo
{
    public static void main(String [] args)
        throws IOException
    {
        Reader r = new FileReader("filename");
        int c;
        while ((c = r.read()) != -1)
        {    System.out.printf(
            "Char code %x is \"%<c\"%n", c);
        }
        r.close();
    }
}
```

---

<sup>51</sup>Or the user's quota expired.

There are a significant number of things about this small bit of sample code that deserve further explanation, and by trying to be minimal the code is not really very good: an improved version is given soon.

Firstly note that `FileReader` is in the `java.io` package so we have an `import` statement to make use if it easy. Next observe that almost all input and output functions can raise exceptions, and this code just admits defeat and notes that its `main` method might therefor fail. I view it as bad style to do this and strongly believe that exceptions should be handled more locally.

Now `FileReader` is a subclass of `Reader`, which is the general class that reads from character streams. So I create a `FileReader` using a constructor that takes a file-name as its argument but store what I get as just a `Reader`. This helps stress to me and remind me that the rest of my code would be equally valid if using some other sort of `Reader`, such as one that gets its input from a pipe, from a string, from characters packed in an array, from a network connection, by running a character decoder on a stream of bytes or otherwise. The way I do things here supposes that the data in the file concerned is encoded in the standard local character-set that Java has been set up for. For reading files imported from elsewhere in the world you have to do things a more complicated way!

The `read` method hands back either the numeric code for a character, or the value `-1` to denote end-of-file. It perhaps seems odd that it returns an `int` not a `char`, but doing so allows it to hand back `-1` which does not stand for any normal character. You can of course case the `int` to a `char` any time you want to!

After having read the file you are expected to call the `close` method. If you fail to do this for an input file you may just leave some machine resources cluttered and unless you try to open and read very many files without closing any of them you will probably not feel any pain. However for output files it may sometimes be that the last bit of your data is not actually sent to the file until you do the `close`. You should get into the habit of ensuring that every file you open does get closed.

A much improved version of the same code can be arrived at by handling the possible exceptions. You may note that `FileNotFoundException` is a subclass of `IOException` which is why the `throws` clause above was sufficient, but which also allows us to see how the improved code is more precise. When you get an exception out of Java it can often be useful to print it, in that it is liable to carry some text that explains further what went wrong. I use `finally` to guarantee that the `close` method of the `Reader` will always be invoked.

```
import java.io.*;

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

```
{
    Reader r;
    try
    {    r = new FileReader("filename");
    }
    catch (FileNotFoundException e)
    {    System.out.printf(e);
        return;
    }
    int c;
    try
    {    while ((c = r.read()) != -1)
        {    System.out.printf(
            "Char code %x is \"%<c\"%n", c);
        }
    }
    catch (IOException e)
    {    System.out.printf("Reading failed (%s)%n", e);
        return;
    }
    finally
    {    r.close();
    }
}
}
```

Output to a file is somewhat similar, and if you only ever want to write individual characters and simple strings then `FileWriter` will suffice. However you may like to be able to use `println` and `printf` when writing data to your file, and they come in a class called `PrintWriter`. Unlike the class `FileWriter`, `PrintWriter` hides all exceptions so you do not need to catch them, but you can check for error using the `checkError` method and you still need to ensure that `close` is called.

```
import java.io.*;

public class PrintDemo
{
    public static void main(String [] args)
    {
        try
        {    PrintWriter w = new PrintWriter("filename");
            try
```

```

        {    w.printf("Hoorah%n");
          assert !w.checkError();
        }
        finally
        {    w.close();
        }
    }
    catch (FileNotFoundException e)
    {    System.out.println("Sorry!");
    }
}
}

```

This time you must make `w` a `PrintWriter` and not just a `Writer` to gain access to `printf` and so on.

If errors arise on a `PrintWriter` the flag marking them persists so you do not need to use `checkError` after every single print statement – every so often and once when you have generated all that you want to end up in the file will suffice. Although I have used `assert` here I probably feel that error checking should be done always and that something along the lines of

```

    if (w.checkError())
        throw new IOException("failure on PrintWriter");

```

might well be better policy.

The long-winded but more flexible way to access files is to start by creating an instance of `java.io.File`. An object of this type can be created using either a constructor that takes a single `String` that names the file (as a complete path, if necessary), or with a two-argument constructor where one argument specifies the directory to look in and the other the file-name within that directory. A `File` object supports methods `exists`, `canRead` and `canWrite` and also one called `isFile`, which test for a “normal” file, ie one that is not a directory or any of the exotic things that in Unix masquerade as sort-of-files. You can pass a `File` rather than a string when opening a `FileReader` or `FileWriter`.

Other methods available via the `File` class include ones to check the length of a file<sup>52</sup>, rename it, create new directories, list all the files in a directory and delete files. You can also create a file by giving just a local name (eg such as `"java.tex"`) and call `getAbsolutePath` to obtain a fully-rooted file-path that identifies it. The exact result you get will clearly be system-dependent, and on one computer I tried that I got back

---

<sup>52</sup>The length reported is liable to count in bytes, and so for text files it can well be that the length reported differs from system to system.



```
"e:\UNIV\notes\java\java.tex"
```

while on another it was

```
"/home/acn1/javanotes/java.tex".
```

The fact that all these facilities are so conveniently supported may make Java one of the more useful programming languages for writing file-management utilities. Once again if you look at Java code and compare it against other languages for very tiny tasks and where previously you would have missed out all error handling Java can look clumsy — but when you look at more realistic and well-engineered examples it starts to feel much nicer.

Binary data access are useful for cases when your data really is raw data and not composed of characters. There classes called `java.io.FileInputStream` and (of course) `FileOutputStream` that take a `File` or a string as an argument and create streams. They of course throw exceptions if the files can not be opened as requested. Once a file has been opened you should in due course call the relevant `close` method to tidy up.

Earlier examples have shown an extra layer of Java constructor arranging to buffer input in the expectation that that may speed it up. I have done that here too.

Putting these together we might arrive at something like this to copy a file in binary mode:

```
String fromName = "source.file";
String toName = "destination.file";
File fromFile = new File(fromName),
    toFile = new File(toName);
if (!fromFile.exists() ||
    !fromFile.canRead() ||
    toFile.exists() ||
    !toFile.canWrite())
{   System.out.println("Can not copy");
    return;
}
InputStream fromStream =
    new BufferedInputStream(
        new FileInputStream(fromFile));
try
{   OutputStream toStream =
        new BufferedOutputStream(
            FileOutputStream(toFile));
    try
    {   for (;;)
        {   toStream.write(fromStream.read());
```

```

        }
    }
    catch (EOFException e)
    {} // Use exception to break out of for loop
    finally
    {    toStream.close();
    }
}
catch (IOException e)
{    System.out.printf("IO error " + e);
}
finally
{    fromStream.close();
}

```

This code is in fact not yet complete! It needs yet more try blocks to guard against `FileNotFoundException` cases where the two streams are created. But it illustrates how the `EOFException` can be used to stop processing at end of file, and demonstrates very clearly that in real file-processing applications most of what you write will be to do with setting everything up and arranging to handle exceptions, while the central interesting bit of the code may be as short as just

```

    for (;;)
    {    toStream.write(fromStream.read());
    }

```

Overall it may seem pretty grim, but in large programs the complication will still remain at the level of the dozen or so lines shown above, rather than growing out of control. It is also probable that the visible pain is because writing high quality file-manipulation code is in fact nothing like as easy as earlier programming languages have tried to make it out to be!

There is a potential down-side in Java being so very insistent that you catch all these errors, in that it can encourage a style of cop-out that just wraps all your code in

```

try
{    ...
}
catch (Exception e)
{}

```

where the block is set up so it catches *all* sorts of `Exception` not just the very special ones that you know are liable to arise, and rather than doing anything it just ignores the error. This very much defeats the purpose Java is trying to achieve! If you are (quite reasonably!) in a rush some time at least go:

```
try
{
    ...
}
catch (Exception e)
{
    System.out.printf("Exception: %s%n", e);
    System.exit(1);
}
```

so that the exceptions you catch are reported and make your program stop.

The above example used `BufferedInputStream` which should not have any effect at all on what your program actually does, but may have an impact on performance when you work with big files. For binary data there are more interesting classes that you could use just as easily: ones to compress and decompress data, encryption and checksumming capabilities. For text data you can use `LineNumberReader` in place of `BufferedReader` and it will keep track of which line you are on in your input. See the classes `FilterInputStream` and `FilterReader` in the documentation for further details.

## 4.8.2 Big integers

The Discrete Mathematics course had an extended section where it discussed highest common factors, modular arithmetic and eventually the RSA encryption scheme. To refine your understanding of all that you could quite properly want to code it all up. To make any pretence at all of reasonable security this means that you need to do a lot of arithmetic on integers that are between 1024 and 1536 bits long. This sort of range of values is about what is required because there is a serious possibility that numbers smaller than that might be factorisable by the best current algorithms and fastest current computers. An implementation of RSA will also need to generate a couple of primes, each with around half that number of bits.

Java has thought of that and it provides a class `java.math.BigInteger` which does essentially everything you could need! And note that `printf` lets you print these big values easily.

In this class there are half a dozen constructors. The more obvious ones construct big integers from strings or `byte` arrays, and a `valueOf` method allows you to create a big integer from a `long`. The two interesting constructors create random big numbers. They both accept an argument that is an object from the class `Random` which actually gives them their randomness. One creates an arbitrary  $n$ -bit number while the other creates an  $n$ -bit number which is (almost certainly) prime. For the second of these it is possible to tune the degree of certainty that a prime has indeed been found by giving a “certainty” argument that tells the con-

structor how hard to work to check things. I might suggest that a value of 50 would be sufficient for all reasonable purposes.

I should provide a rather heavy health warning here. If you use the Java-provided `Random` class to help you create private keys or other values of cryptographic significance you will be throwing away almost all the security that the RSA method could give you, since this random number generator comes too close to having a predictable behaviour. Specifically there is a chance that to arrange to get the same “random” values that you do it may suffice for somebody to run a similar Java program having reset their computer so that their run appears to happen at the exact time of day that yours did. This may be hard but is nothing like as hard as factorising 1536-bit integers. If you ever wanted to use serious encryption you *must* instead use `java.security.SecureRandom`. Anybody really serious about security would think at length before trusting even the things in `java.security`: how is it possible to tell that they do what they are supposed to and that they do not include secret weaknesses? And even if they are honest it is astonishingly easy to lose all the security you thought you had by some apparently minor clumsiness in how you use your cryptographic primitives. A course on security later in the Tripos gives much more information about all of this!

Note that some of the functionality in the Java security and encryption packages may be missing or limited unless your installation has provided some level of assertion that you are not a national of a country that the US Government does not like and that you are not a terrorist. But as is the way of any such attempt at blocking access to technology, there are easy to find drop-in replacements not hampered by (so many) export license issues. You might still be aware that good encryption is viewed by some as something with significant security implications and that it should not be given any opportunity to cross international borders until you at the very very least know what all the rules are! Java itself provides ways that those who satisfy the correct eligibility conditions can use the standard Java libraries and obtain unlimited security, via the installation of special “JCE policy files”.

Once you have made suitable objects of class `BigInteger` the library provides you with methods to add, subtract, multiply and divide them, to even raise one big number to a huge power modulo another number (what a give-away about the expected use of this class!). The function that computes what the Discrete Mathematics notes called a Highest Common Factor is here known as a Greatest Common Divisor (`gcd`), but the change of name does not hide any change of behaviour<sup>53</sup>.

---

<sup>53</sup>The `javasecurity` package provides easy to use functions for generating keys and computing message digests and digital signatures. There is a standard extension to Java that supplies encryption and further functionality: this part may be subject to export regulation and has to be fetched and installed as a separate item from the main SDK.

```
import java.math.*;
import java.util.*;
...
Random r = new Random(); // inadequate!
    // use the SecureRandom class instead!!!
// Create two big primes p and q
BigInteger p =
    new BigInteger(768, // length in bits
        50, // only 1 in 2^50 prob of non-prime
        r); // random number source
BigInteger q = new BigInteger(768, 50, r);
// form their product, n, which can be public
BigInteger n = p.multiply(q);
// compute phi = (p-1)*(q-1)
BigInteger bigOne = BigInteger.ONE;
BigInteger pMinusOne = p.subtract(bigOne);
BigInteger qMinusOne = q.subtract(bigOne);
BigInteger phi = pMinusOne.multiply(qMinusOne);
// select a random exponent whose HCF with phi
// is 1.
BigInteger e;
do
{   e = new BigInteger(1536, r);
} while (!phi.gcd(e).equals(bigOne));
// now (n, e) is the public key
...
// Set up a message to encrypt
BigInteger msg =
    new BigInteger("12341234"); // silly message
// Encrypt with public key. One line of code!
BigInteger k = msg.modPow(e, n);
...
```

The code is clearly about as short as it possibly could be. Again let me warn you that cryptographically satisfactory random number generators are hard to come by, and that such issues as managing the secure distribution of public keys and keeping private ones truly private mean that security involves very much more than just these few lines of code. But Java is clearly making it easy to make a start on it.

How do you know that the Authorities and not bugging your computer while you run the above code? How do you know that no traces of the secret information remain anywhere when you have logged off or even powered down the computer? The Computer Lab's security group has a fine track-record of demonstrating that

even apparently safe computing habits leak information to a sufficiently skilful and ingenious snooper.

### 4.8.3 Collections

Java has an interface called `Collection` and a whole range of interesting classes derived from it. The general idea is that `Collection` covers ideas like “set”, “list” and “vector”. In some cases the elements in a collection can be ordered (in which case the objects included must all implement the `Comparable` interface<sup>54</sup>), but might not be. Collections may be implemented as linked lists or as vectors, but the library classes arrange that when a vector is used it will be enlarged as necessary so that the user does not have to specify a limit to the size of the collection too early. One sub-case of a `Collection` is a `Map`, which provides for general ways to organize various sorts of table or dictionary. I am not providing any sample code using `Collections` in this little section since I believe that when you browse the documentation you will find them easy to cope with. However it may make sense for me to list more of the names of classes worth looking at: `Collection`, `Collections`, `Set`, `HashSet`, `TreeSet`, `Vector`, `LinkedList`.

The collection classes are keyed to the Java `for` statement to make it trivial to iterate over the values in a collection: as has been seen in various of the sample programs here.

Very typically when you create an instance of a `Collection` Class you will use the generics capability to indicate the type of the objects you will keep in it, eg

```
Vector<String> v = new Vector<String>();
```

and if you do so Java will know that the values you extract will be of the type indicated.

### 4.8.4 Simple use of Threads

A *thread* is a stream of computation that can go on in parallel with others. The term is used when the activities are part of a single program, and where there is no need for security barriers to protect one thread from the next. The more general term used when the extra overhead of protection is needed is *process*. Java is one of the first languages to make a big point of having threads supported as

---

<sup>54</sup>An especially interesting issue here is the way that Java can compare strings. To support the needs of different nationalities a class `Collator` is provided, and methods in it can order strings based on proper `Locale`-specific understandings of where accented or non-English letters should go. Alphabetic ordering with international texts is a more complicated business that almost all of you would have imagined.

a standard facility. Many systems in the past have had threads, but usually in rather non-portable form. Almost any program that has to implement a complicated windowed user interface or which accesses remote computers<sup>55</sup> will need to use threads so that one high priority thread can ensure that the user always gets responses to requests, while several lower priority ones keep on with some bigger calculation. There are very many subtleties in any program that exhibits concurrency. I will not describe these here, and in consequence I expect that people who try to make substantial use of threads based on just these notes will get themselves into deep water. There are two typical bad effects that can arise. In one the system just locks up as a chain of threads each wait for the others to complete some task. In the other two threads both attempt to update some data structure at around the same time and their activities interfere, leaving data in a corrupted state. The Java keyword `synchronized` is involved in some of the resolutions of these sorts of problem.

The example here is supposed to do not much more than to alert you to the possibility of writing multi-threaded Java programs, and to show how easy it is. I will start by defining a class that will encapsulate the behaviour I want in the rather silly thread that I will use here:

```
class Task extends Thread
{
    boolean resultShown;
    String result;
    int identification;
    Task(int i)
    {   identification = i;
        resultShown = false;
    }
    public void run()
    {   try { sleep(20+100*identification % 77); }
        catch (InterruptedException e) { return; }
        result = String.valueOf(identification);
    }
}
```

The two critical things are that my class extends `Thread` and that it implements `run`. The method `run` will be to a thread much what `main` is to a complete program. In this case I make my thread do something rather minimal. It goes to sleep for an amount of time that depends on the argument that was passed to its constructor, and it then sets one of its variables, `result`, to a string version of that value. When created my task also sets a flag that I will use later on to record whether I have picked up its result.

---

<sup>55</sup>Often a slow business.

To demonstrate use of threads I will create half a dozen instances of the above, and then wait around until each has finished its work. As I notice each task completing I will pick up its `result` and display it. When I have done that I will set the `resultShown` flag so that I do not display any result twice. I could surely find a cleverer way of achieving that, but the solution I use here is at least quite concise. Once all my threads have finished I will let my main program terminate. I let my top-level class inherit from `Thread` just because I want to use `sleep` in it so that while waiting for the sub-tasks to finish I am mostly idle.

```
public class Threads extends Thread
{
    static final int THREADCOUNT = 6;

    public static void main(String[] args)
    {
// Create and start six threads
        Task [] spawn = new Task [THREADCOUNT];
        for (int i = 0; i<THREADCOUNT; i++)
        {   spawn[i] = new Task(i);
            spawn[i].start();
        }
        System.out.println("All running now");
        int stillGoing = THREADCOUNT;
// Scan looking for terminated threads
        while (stillGoing != 0)
        {   for (int i=0; i<THREADCOUNT; i++)
            {   if (!spawn[i].isAlive() &&
                !spawn[i].resultShown)
// print result the first time I notice a thread dead
                {   System.out.println("Result from " +
                    i + " = " + spawn[i].result);
                    spawn[i].resultShown = true;
                    stillGoing--;
                }
            }
            System.out.println("One scan done");
// sleep for 7 milliseconds between scans to avoid waste
            try { sleep(7); }
                catch (InterruptedException e) { break; }
        }
        System.out.println("All done");
    }
}
```

Observe that `sleep` can raise an exception if the sleeping task receives an in-



interrupt from elsewhere, and I (have to) catch this and quit. The results I obtain follow, and you can see traces that show the main program scanning round looking for threads that have finished and also you can see that the different threads terminate in some curious order. Of course a more worthwhile example would put real computation into each of the threads and their termination would be based on how long that took rather than on the artificial sleeping I have used here!

```
java Threads
  All running now
  One scan done
  One scan done
  One scan done
  Result from 0 = 0
  Result from 4 = 4
  One scan done
  Result from 1 = 1
  One scan done
  One scan done
  Result from 5 = 5
  One scan done
  Result from 2 = 2
  One scan done
  One scan done
  One scan done
  Result from 3 = 3
  One scan done
  All done
```

The reason my example is respectably simple and trouble-free is that the threads only communicate by receiving data when first created and by delivering something back when they have finished. Inter-process communication beyond that can be astonishingly hard to get right.

### 4.8.5 Network access

Java really hit the news as a language for animating your own web pages. One part of doing this is the set of graphical operations that it supports. Another less instantly visible but equally important thing is the ability to connect to remote computers and retrieve data from them. The set of rules that make up HTTP<sup>56</sup> are what defines the World Wide Web. Standard Java libraries provide various degrees of ability to connect using it. The small program shown here links through to a

---

<sup>56</sup>HyperText Transfer Protocol.

fixed location named as its fire command-line argument and displays the data found there. This data comes out as an HTML document with lots of tags that are enclosed in angle brackets.

```
// Read file from a possibly remote web server

import java.net.*;
import java.io.*;

public class Webget
{

public static void main(String [] args)
{
    URL target;
    try
    { target = new URL(args[0]);
    }
    catch (MalformedURLException e)
    { return; } // Badly formed web-page address
    try
    { URLConnection link = target.openConnection();
      link.connect(); // connect to remote system
// Now just for fun I display size and type information
// about the document that is being fetched. Note that
// documents might be pictures or binary files as well
// as just text!
      System.out.println("size = " +
        link.getContentLength());
      System.out.println("type = " +
        link.getContentType());
// getInputStream() gives me a handle on the content, and
// I rather hope it is text. In that case I can get the
// characters that make it up from the InputStream.
      Reader in = new InputStreamReader(
        link.getInputStream());
      int c;
// Crude echo of text from the document to the screen.
// It will have lots of HTML encoding in it, I expect.
      while ((c = in.read()) != -1)
        System.out.print((char)c);
    }
// A handler is needed in case exceptions arise.
    catch (IOException e)
```

```
        { System.out.println("IO error on link"); }  
// I am lazy here and do not close anything down.  
}  
  
}  
  
// end of Webget.java
```

The data stored on the CL teaching support pages in mid February 1998 started off as follows, apart from the fact that I have split some of the lines to make the text fit neatly on the pages of these notes. It has of course changed by now! I keep this old material in the notes out of nostalgia.

```
<HTML>  
<HEAD>  
<TITLE>Comp.Sci. Teaching pages</TITLE>  
</HEADER>  
<BODY>  
  
<H1> Computer Science teaching material on Thor</H1>  
  
<P>  
<UL>  
<LI><A HREF="Java/java-index.html"> Some Information  
    about Java </A> (on this server)  
    ...
```

The main message here is that accessing a remote web-site is just about as trivial as reading from a local file.

### 4.8.6 Menus, scroll bars and dialog boxes

Back when Java 1.2 was released Sun finalised a whole set of windows management code that they called Swing. This extended and in places replaced earlier windowing capabilities that were known as AWT. I believe that by now it is proper to use the Swing versions of things even in those cases where older AWT versions remain available. You can tell that you are doing that when you use a lot of class names that start with a capital “J”!

The code presented here is called `MenuApp` and is a pretty minimal demonstration of menus! I will use this example to show how something can be both an application and an applet. The “application” bit of course defined a method called `main`, and this just sets up a window (frame) that can contain the applet stuff. There is a bit of mumbo jumbo to allow the application to stop properly

when the window is closed. As usual I will show the inner bit of the code first – the fragment that actually does the work:

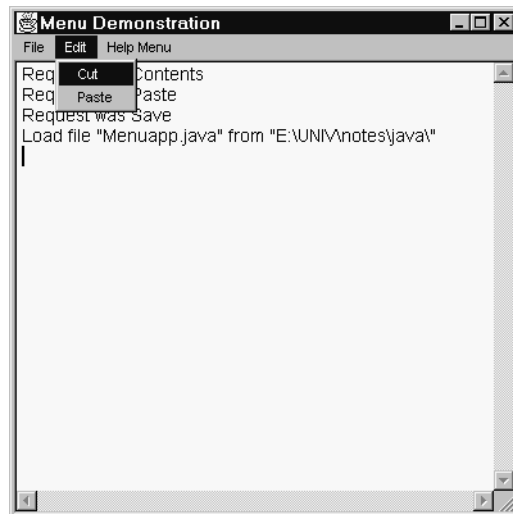
```
public static void main(String[] args)
{
    Menuapp window = new Menuapp();
    JFrame f = new JFrame("Menu Demonstration");
    f.addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    });
    f.getContentPane().add(window, BorderLayout.CENTER);
    f.setSize(600, 400);
    f.setVisible(true);
}
```

I should point out the syntax

```
new WindowAdapter()
{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}
```

which extends the class `WindowAdapter` to produce a new (anonymous) class. In this new class it overrode the `windowClosing` method. It then creates an instance of the new anonymous class. This arrangement is known as an “Inner Class” and can be very handy when you need a small variant on some existing class and will use it just once so that giving it a name would be over-clumsy.

The constructor for `Menuapp` will establish a menu bar at the top of itself, then makes menus on that bar, and places menu items on each menu. In much the way that mouse events were dealt with by registering a handler for them it is necessary to implement an



Menuapp running



```
        f.setSize(600, 400);
        f.setVisible(true);
    }

// All real work happens because of this
// constructor. I create a JTextPane to hold
// input & output and make some menus.

    JTextPane text;
    Container cc;

    public Menuapp()
    {
        cc = getContentPane();
        text = new JTextPane();
        text.setEditable(true);
        text.setFont(
            new Font("MonoSpaced", Font.BOLD, 24));
        JScrollPane scroll = new JScrollPane(text,
            JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
        cc.add(scroll);

// Menus hang off a menu bar and contain menu items
        JMenuBar bar;
        JMenu mFile, mEdit, mHelp;
        JMenuItem mNew, mOpen, mSave, mCut,
            mPaste, mContents;

// Create a menu bar first and add it to the Frame
        bar = new JMenuBar();          setJMenuBar(bar);
// Create a menu and add it to the MenuBar
        mFile = new JMenu("File");    bar.add(mFile);
// Create menu items and add to menu
        mNew = new JMenuItem("New");  mFile.add(mNew);
        mOpen = new JMenuItem("Open"); mFile.add(mOpen);
        mSave = new JMenuItem("Save"); mFile.add(mSave);

        mEdit = new JMenu("Edit");    bar.add(mEdit);
        mCut = new JMenuItem("Cut");  mEdit.add(mCut);
        mPaste = new JMenuItem("Paste");mEdit.add(mPaste);

        mHelp = new JMenu("Help Menu");
```

```
        bar.add(mHelp);
        mContents = new JMenuItem("Contents");
        mHelp.add(mContents);

// Each menu has to be activated to be useful.
        mNew.addActionListener(this);
        mOpen.addActionListener(this);
        mSave.addActionListener(this);
        mCut.addActionListener(this);
        mPaste.addActionListener(this);
        mContents.addActionListener(this);
    }

// When a menu item is selected this gets called,
// and getActionCommand() retrieves the text from
// the menuItem. Here I clear the area when New
// is used, and do something with Open, but otherwise
// just display a message.
    public void actionPerformed(ActionEvent e)
    {
        String action = e.getActionCommand();
        try
        {
            if (action.equals("New")) text.setText("");
            else if (action.equals("Open")) openFile();
            else
            {
                StyledDocument s =
                    text.getStyledDocument();
                s.insertString(s.getLength(),
                    "Request was " + action + "\n", null);
            }
        }
        catch (BadLocationException e1)
        {}
    }

void openFile() throws BadLocationException
{
    JFileChooser d = new JFileChooser("Open a file");
    if (d.showOpenDialog(cc) ==
        JFileChooser.APPROVE_OPTION)
    {
        StyledDocument s = text.getStyledDocument();
        s.insertString(s.getLength(),
            "Load file \"" +
            d.getSelectedFile().getAbsolutePath() +
```

```

        "\n", null);
    }
}

// end of Menuapp.java

```

You should expect that extending the above example or writing your own code that sets up controllable visual effects will cause you to have to do rather a lot of reading of the class library documentation to plan which classes you will derive items from. Also when you have mastered the basics of GUI construction by hand you will very probably want to take advantage of one of the Java development environments that can set up frameworks for user-interfaces for you in really convenient ways.

## 4.8.7 Exercises

### Replacement for “ls”

On Unix the command `ls` lists all the files in the current directory. With a command-line flag `-R` it also lists members of sub-directories. Investigate the Java `File` class and see how much of the behaviour of `ls` (or the DOS/Windows `dir`) you can re-create.

### RSA

The code fragment above suggests how to create a public key and then how to use it to encrypt a message once that message has been reduced to a `BigInteger` of suitable size. Flesh this out with code that can use the private key to decrypt messages, and with some scheme that can read text from the standard input (or a file, maybe), split it up into chunks and represent each chunk as a `BigInteger`.

You might also want to investigate the Java Cryptographic Architecture and find out what is involved in creating cryptographic-grade random values. You should be made very aware that the ordinary Java random number generator does not pretend that the values it returns to you are suitable for use in security applications.

Then do a literature search to discover just what you are permitted to do with an implementation of an idea that has been patented<sup>57</sup> and also what the Secu-

---

<sup>57</sup>The main RSA US patent expired on the 20th September 2000, but that does not necessarily mean that all associated techniques and uses are unprotected. Also note that there are other public key methods for both digital signatures and for encryption where the original patents have



curity Policies of various countries are about the use, import and export of strong encryption technology.

Note again that Java provides the specification of a security library that would do all this for you if it were not for the USA government's export restrictions, and if these restrictions do not apply to you you could use the Java strong security. There are international re-implementations of this library that can be fetched and used here. See for instance [www.openjce.org](http://www.openjce.org). But also be aware that exporting code that includes strong encryption may be subject to government restriction.

### Big Decimals

The class `BigDecimal` builds on `BigInteger` to provide for long precision decimal fractions. When a `BigDecimal` is created it will behave as if it has some specified number of digits to the right of its decimal point, but as arithmetic is carried out there can be any number of digits generated before the decimal point.

For any number  $z$  one can define a sequence of values  $x_i$  by  $x_0 = 1$  and  $x_{i+1} = (x_i + z/x_i)/2$ . This sequence will converge to  $\sqrt{z}$ , and once it gets even reasonably close it homes in rapidly, roughly doubling the number of correct significant values at each step. For finding square roots of numbers between 0.5 and 2 (say) the starting value  $x_0 = 1$  will be quite tolerable.

If one wanted the square root of a number larger than 2 or smaller than 0.5 it would make sense to augment the recurrence formula by use of the identity  $\sqrt{4z} = 2\sqrt{z}$  to pre-normalise  $z$ .

Implement the method and use it to create a table of square roots of the integers from 1 to 10 all to 100 decimal places.

If you can perform high-precision arithmetic on floating point numbers you should try the following calculation. I am going to use informal syntax with `double` constants written where you will need to use `BigDecimal`s, and I will use the ordinary operators to work on values not calls to suitable methods in the `BigDecimal` class. I am also not going to tell you here either how many cycles of the loop you should expect the code to obey or what the values it prints out will look like! But I suggest that you work to (say) 1000 decimals and see what you get.

```
a = 1.0;
b = 1/sqrt(2.0);
u = 0.25;
x = 1.0;
pn = 4.0;
do
```

---

also expired, and so using these can also be of interest without patent worries about deployment. Investigate "El Gamal".

```

{   p = pn;
    y = a;
    a = (a+b)/2.0;
    b = sqrt(y*b);
    u = u-x*(a-y)*(a-y);
    x = 2*x;
    pn = a*a/u;
    print(pn);
} while (pn < p);

```

When you have investigated this for yourself try looking for arithmetic-geometric mean (and the cross-references from there!) in Hakmem.

### **Mandelbrot set (again)**

Adjust the Mandelbrot set program, as considered in an earlier exercise, so that at the start (maybe in an `init` method?) it allocates a 400 by 400 array of integers all initially set to 0. Change the `paint` method so that it just displays colours as recorded in this array, and have a separate thread that (a) computes values to go in the array and (b) uses `repaint` at the end of each of its resolution cycles to bring make sure that the screen is brought fully up to date then. The objective here is to arrange that each pixel is only calculated once and that thus the `paint` method is a lot quicker.

Further extension of this code would add mouse support so that (as a start) a mouse click would cause the display to re-centre at the selected point and start drawing again using half the previous range. Those of you who get that far are liable to be able to design more flexible and convenient user-driven controls than that, I expect!

### **Tickable exercise 7**

Tick 5 was the largest single exercise. Tick 6 was a fairly easy example of getting an Applet working. This final exercise is intended to be a reasonably manageable one to end off the series of Java practicals.

The illustration of network code that I gave represents a minimal way of accessing a remote file.

1. Extend it so that it can be used as

```
java Webget URL destination
```

to fetch the contents of an arbitrary web document and store it on the local disc. If an explicit destination file is not specified your program should display the document fetched on the screen;

2. Investigate document types and encodings and try to arrange that text documents are fetched as streams of characters while binary documents come across the net verbatim;
3. *Optional:* Some files on the web are of type “zip”, being compressed. Java provides a set of library functions for uncompressing such files. Use it so that when a zipped file is fetched the data that arrives locally is the decompressed form of it.
4. *Very optional:* A `javax.swing.JEditorPane` can be set up so that it renders (a subset of) HTML for you. Using this perhaps you could go a significant way towards implementing your own light-weight web browser and you might only need a fairly modest amount of Java!

*(End of tickable exercise)*

### Yet further network-related code

The parts of the Java libraries that can fetch material from remote machines understand quite well that you will only occasionally be fetching raw text, and that fairly often the data to be accessed will be a picture or a sound. Investigate the documentation and sort out how to use these facilities. An image from a local student-run web site is shown here to illustrate the usefulness of these facilities. Further parts of the network classes allow you to detect (to some extent) what sort of data is about to be fetched from a web location so that different sorts of data can each be handled in the most appropriate manner.

If you could write code that located words enclosed in angle brackets within text, and lay text out in a window in such a way that you could tell which word was the subject of a mouse click you might find yourself half-way to writing your very own web browser!



Figure 4.8: [www.arthurnorman.org](http://www.arthurnorman.org)!

### A minimal text editor

The menu example already allows you to re-position the cursor and type in extra text. I have shown how to identify files to be loaded and saved. The `TextArea` class provides methods that would implement cut and paste. Put all these together to get a notepad-style editor.

### More Fonts

The Unicode example showed that it is easy to select which font Java uses to paint characters. The class `FontMetrics` then provides methods that allow you to obtain very detailed measurements of both individual characters and rows of them. Using all this you can get rather fine control over visual appearance. Using however many of these facilities you like create a Java applet that displays the word `LATEX` in something like the curious layout that the designers of that text-formatting program like to see.

### Two flasks

This was a puzzle that I found in one of the huge anthologies of mathematical oddities:

An apothecary has two spherical flasks, which between them hold exactly 9 measures of fluid. He explains this to his friend the mathematician, and adds that the glass-blower who makes his flasks can make them perfectly spherical but will only make flasks whose diameter is an exact rational number. The mathematician looks unimpressed and says that the two flasks obviously have diameters 1 and 2, so their volume is proportional to  $1^3 + 2^3 = 9$ . Hmmm says the apothecary, that *would* have worked, but I already had a pair like that. This pair of flasks has exactly the same total capacity, still has linear dimensions that are exact fractions but the neither flask has diameter 1.

Find a size that the smaller of his two flasks might have had.

This is clearly asking for positive solutions to  $x^3 + y^3 = 9$  with the solution being a neat fraction. In an earlier exercise you were invited to write a class to handle fractions. Before continuing with this one you might like to ensure that you have a version of it that uses `BigInteger` just to be certain that overflow will not scupper you.

Here is an attack you can make on the problem. We are interested in solutions to  $x^3 + y^3 = 9$  and know an initial solution  $x = 1, y = 2$ . Imagine the cubic equation drawn as a graph (I know it is an implicit equation, so for now I will be happy if

you imagine that there is a graph and do not worry about actually drawing it!). The point  $(1,2)$  lies on the curve. Draw the tangent to the curve at  $(1,2)$ . This straight line will have an equation of the form  $lx + my + n = 0$  for some  $l$ ,  $m$  and  $n$  which you can find after you have done a little differentiation. Now consider where the straight line meets the curve. If one eliminated  $y$  between the two equations the result is a cubic in  $x$ , but we know that  $x = 1$  is one solution (because the curve and line meet there). In fact because the line is a tangent to the curve that is a double solution, and thus the cubic we had will necessarily have a factor  $(x - 1)^2$ . If you divide that out you get a *linear* equation that gives you the  $x$  co-ordinate of the other place where the tangent crosses the curve. Solve that and substitute into the line equation to find the corresponding value of  $y$ . The pair  $x, y$  were found by solving what were in the end only linear equations, and so the numbers must be at worst fractions. This has therefore given another rational point on the cubic curve. What you get there is not a solution to the problem as posed, since one of  $x$  and  $y$  will end up negative. But maybe if you take a tangent at the new point that will lead you to a third possibility and perhaps following this path you will eventually manage to find a solution that is both rational and where both components are positive.

Write code to try this out, and produce the tidiest fractional solution to the original puzzle that you can, ie the one with smallest denominators.



A member of the audience checking a detailed point with the lecturer at the end of a session.

Figure 4.9: Odds and Ends follow: watch what the lecturer happens to cover.

## Chapter 5

# Designing and testing programs in Java

At this stage in the “Programming in Java” course I will start to concentrate more on the “programming” part than the “Java”. This is on the basis that you have now seen most of the Java syntax and a representative sampling of the libraries: now the biggest challenge is how to use the language with confidence and competence.

In parallel with this course you get other ones on *Software Engineering*. These present several major issues. One is that errors in software can have serious consequences, up to and including loss of life and the collapse of businesses. Another is that the construction of large computer-related products will involve teams of programmers working to build and support software over many years, and this raises problems not apparent to an individual programmer working on a private project. A third is that formal methods can be used when defining an planning a project to build a stable base for it to grow on, and this can be a great help. The emphasis is on programming in the large, which is what the term “Software Engineering” is usually taken to have at its core. Overall the emphasis is on recognition of the full life-time costs associated with software and the management strategies that might keep these costs under control.

The remaining section of this Java course complements that perspective and looks at the job of one person or a rather small group, working on what may well be a software component or a medium sized program rather than on a mega-scale product. The intent of this part of the course is to collect together and emphasise some of the issues that lie behind the skill of programming. Good programmers will probably use many of the techniques mentioned here without being especially aware of what they are doing, but for everybody (even the most experienced) it can be very useful to bring these ideas out into the open. It seems clear to me that all computer science professionals should gain a solid grounding carrying out small projects before they actually embark on larger ones, even though a vision of what

will be needed in big projects help shape attitudes and habits.

It is at least a myth current in the Computer Laboratory that those who intend to become (mechanical) engineers have as an early part of their training an exercise where they fashion a simple cube of solid metal, and they are judged on the accuracy of their work. Such an activity can be part of bringing them into direct touch with the materials that they will later be working with in more elaborate contexts. It gets them to build experience with tools and techniques (including those of measurement and assessment). Programming in the small can serve similar important purposes for those who will go on to develop large software systems: even when large projects introduce extra levels of complication and risk the design issues discussed here remain vital.

One of the major views I would like to bring to the art (or craft, or science, or engineering discipline, depending on how one likes to look at it) of programming is an awareness of the value of an idea described by George Orwell in his book “1984”. This is *doublethink*, the ability to believe two contradictory ideas without becoming confused. Of course one of the original pillars of doublethink was the useful precept *Ignorance is Strength*, but collections of views specifically about the process of constructing programs. These notes will not be about the rest of the association of computers with surveillance, NewSpeak or other efficiency-enhancing ideas. The potentially conflicting views about programming that I want to push relate to the prospect of a project succeeding. Try to keep in your minds both the idea *Programming is incredibly difficult and this program will never work correctly: I am going to have to spend utterly hopeless ages trying to coax it into passing even the most minimal test cases* and its optimistic other face, which claims cheerfully *In a couple of days I can crack the core of this problem, and then it will only take me another few to finish off all the details. These days even young children can all write programs*. The concise way to express this particular piece of doublethink (and please remember that you really have to believe both part, for without the first you will make a total botch of everything, while without the second you will be too paralysed ever to start actual coding), is

### **Writing programs is easy.**

A rather closely related bit of doublethink alludes both to the joy of achievement when a program appears to partially work and the simultaneous bitter way in which work with computers persistently fail. Computers show up our imperfections and frailties, which range through unwillingness to read specifications via inability to think straight all the way to incompetence in mere mechanical typing skills. The short-hand for the pleasure that comes from discovering one of your own mistakes, and having spent many frustrating hours tracking down something that is essentially trivial comes out as



## Writing programs is fun.

A further thing that will be curious about my presentation is that it does not present universal and provable absolute truths. It is much more in the style of collected good advice. Some of this is based on direct experience, other parts has emerged as an often-unstated collective view of those who work with computers. There are rather fewer books covering this subject than I might have expected. There is a very long reading list posted regularly on `comp.software-eng`, but most of it clearly assumes that by the time things get down to actually writing programs the reader will know from experience what to do. Despite the fact that it is more concerned with team-work rather than individual programming I want to direct you towards *the Mythical Man Month*[7], if only for the cover illustration of the La Brea Tar Pits<sup>1</sup> with the vision that programmers can become trapped just as



Figure 5.1: The La Brea tar pits.

the Ice Age mammoths and so on were. Brooks worked for IBM at a time that they were breaking ground with the ambitious nature of their operating systems. The analogous Microsoft experience is more up to date and can be found in *Writing*

---

<sup>1</sup>You may not be aware that the tar pits are in the middle of a thoroughly built-up part of Los Angeles, and when visiting them you can try to imagine some of the local school-children venturing too far and getting bogged down, thus luring their families, out for a week-end picnic, to a sticky doom.

*Solid Code*[18] which gives lots of advice that is clearly good way outside the context of porting Excel between Windows and the Macintosh. If you read the *Solid Code* book you will observe that it is concerned almost entirely with C, and its examples of horrors are often conditioned by that. You should therefore let it prompt you into thinking hard about where Java has made life safer than C and where it has introduced new and different potential pitfalls.

For looking at programming tasks that are fairly algorithmic in style the book by Dijkstra[10] is both challenging and a landmark. There are places where people have collected together some of the especially neat and clever **little** programs they have come across, and many of these indeed contain ideas or lessons that may be re-cyclable. Such examples have come to be referred to as “pearls”[4][5]. Once one has worked out what program needs to be written ideas (now so much in the mainstream that this book is perhaps now out of date) can be found in one of the big presentations by some of the early proponents of structured programming[19]. Stepping back and looking at the programming process with a view to estimating programmer productivity and the reliability of the end product, Halstead[14] introduced some interesting sorts of software metrics, which twenty years on are still not completely free of controversy. All these still leave me feeling that there is a gap between books that describe the specific detail of how to use one particular programming language, and those concerned with large scale software engineering and project management. To date this gap has generally been filled by an apprentice system where trainee programmers are invited to work on progressively larger exercises and their output is graded and commented on by their superiors. Much like it is done here! With this course I can at least provide some background thoughts that might help the apprentices start on their progression a little more smoothly.

When I started planning a course covering this material it was not quite obvious how much there was going to be for me to say that avoided recitations of the blindingly obvious and that was also reasonably generally applicable. As I started on the notes it became clear that there are actually a lot of points to be covered. To keep within the number of lectures that I have and to restrict these notes to a manageable bulk I am therefore restricting myself (mostly) to listing points for consideration and giving as concrete and explicit recommendations as I can: I am not including worked examples or lots of anecdotes that illustrate how badly things can go wrong when people set about tasks in wrong-minded ways. But perhaps I can suggest that as you read this document you imagine it expanded into a very much longer presentation with all that additional supporting material and with a few exercises at the end of each section. You can also think about all the points that I raise in the context of the various programming exercises that you are set or other practical work that you are involved with.

## 5.1 Different sorts of programming tasks

Java experience illustrates very clearly that there are several very different sorts of activity all of which are informally refereed to as “programming”. On style of use of Java — of great commercial importance — involves understanding all of the libraries and stringing together uses of them to implement user interfaces and to move data around. In such cases the focus is entirely on exploiting the libraries, on human factors and on ensuring that the code’s behaviour agrees with the manual or on-line documentation that its users will work from. At another extreme come tasks that involve the design of many new data-structures and algorithmic innovations in their use. Often in this second style of program there will also be significant concern over efficiency.

Given that there are different sorts of software it might be reasonable to start with a classification of possible sorts of program. There are three ways in which this may help with development:

1. Different sorts of computer systems are not all equally easy to build. For instance industrial experience has shown repeatedly that the construction of (eg) an operating system is very much harder than building a (again eg) a compiler even when the initial specifications and the amount of code to be written seem very similar. Thinking about the category into which your particular problem falls can help you to plan time-scales and predict possible areas of difficulty;
2. The way you go about a project can depend critically on some very high level aspects of the task. A fuller list of possibilities is given below, but two extreme cases might be (a) a software component for inclusion in a safety-critical part of an aero-space application, where development budget and timescale are subservient to an over-riding requirement for reliability, and (b) a small program being written for fun as a first experiment with a new programming language, where the program will be run just once and nothing of any real important hands on the results. It would be silly to carry forward either of the above two tasks using a mind-set tuned to the other: knowing where one is on the spectrum between can help make the selection of methodology and tools more rational;
3. I will make a point in these notes that program development is not something to be done in an isolated cell. It involves discussing ideas and progress with others and becoming aware of relevant prior art. Thinking about the broad area in which your work lies can help you focus on the resources worth investigating. Often some of these will not be at all specific to the

immediate description of what you are suppose to achieve but will concerned with very general areas such as rapid prototyping, formal validation, real-time responsiveness, user interfaces or whatever.

I will give my list of possible project attributes. These are in general not mutually exclusive, and in all real cases one will find that these are not yes–no choices but more like items to be scored from 1 to 10. I would like to think of them as forming an initial survey that you should conduct before starting any detailed work on your program just to set it in context. When you find one or two of these scoring 9 or 10 out of 10 for relevance you know you have identified something important that ought to influence how you approach the work. If you find a project scores highly on *lots* of these items then you might consider trying to wriggle out of having to take responsibility for it, since there is a significant chance that it will be a disaster! The list here identifies potential issues, but does not discuss ways of resolving them: in many cases the project features identified here will just tell you which of the later sections in these notes are liable to be the more important ones for your particular piece of work. The wording in each of my descriptions will be intended to give some flavour of how *severe* instances of the issue being discussed can cause trouble, so keep cheerful because usually you will not be plunging in at the really deep end of the pool.

**Ill-defined** One of the most common and hardest situations to face up to is when a computer project is not clearly specified. I am going to take this case to include ones where there appears to be a detailed and precise specification document but on close inspection the requirements as written down boil down to “I don’t know much about computer systems but I know what I like, so write me a program that I will like, please.” Clearly the first thing to do in such a case is to schedule a sub-project that has the task of obtaining a clear and concise description of what is really required, and sometimes this will of itself be a substantial challenge;

**Safety-critical** It is presumably obvious that safety-critical applications need exceptional thought and effort put into their validation. But this need for reliability is far from an all-or-nothing one, in that the reputation of a software house (or indeed the grades obtained by a student) may depend on ensuring that systems run correctly at least most of the time, and that their failure modes appear to the user to be reasonable and soft. At the other extreme it is worth noting that in cases where robustness of code and reliability of results are not important at all (as can sometimes be the case, despite this seeming unreasonable) that fact can be exploited to give the whole project a much lighter structure and sometimes to make everything very much easier to achieve. A useful question to ask is “Does this program have to work

correctly in *all* circumstances, or does it just need to work in *most* common cases, or indeed might it be sufficient to make it work in just *one* carefully chosen case?”

**Large** It is well established that as the size of a programming task increases the amount of work that goes into it grows much more rapidly than the number of lines of code (or whatever other simple measurement you like) does. At various levels of task size it becomes necessary to introduce project teams, extra layers of formal management and in general to move away from any pretence that any single individual will have a full understanding of the whole effort. If your task and the associated time-scales call for a team of 40 programmers and you try it on your own maybe you will have difficulty finishing it off! Estimating the exact size that a program will end up or just how long it will take to write is of course very hard, but identifying whether it can be done by one smart programmer in a month or if it is a big team project for five years is a much less difficult call to make.

**Shifting sands** If either project requirements or resources can change while software development is under way then this fact needs to be allowed for. Probable only a tiny minority of real projects will be immune from this sort of distraction, since even for apparently well-specified tasks it is quite usual that experience with the working version of the program will lead to “wouldn’t it be nice if . . .” ideas emerging even in the face of carefully discussed and thought out early design decisions that the options now requested would not be supportable. Remember that Shifting Sands easily turn into Tar Pits!



Figure 5.2: The museum frieze at La Brea.

**Urgent** When are you expected to get this piece of work done? How firm is the deadline? If time constraints (including the way that this project will compete with other things you are supposed to do) represents a real challenge it

is best to notice that early on. Note that if, while doing final testing on your code, you find that it has a bug in it there may be no guarantee that you can isolate or fix this to any pre-specified time-scale. This is because (at least for most people!) there is hardly any limit to the subtlety of bugs and the amount of time and re-work needed to remove them. If the delivery date for code is going to be rigidly enforced (as is the case with CST final year projects!) this fact may be important: even if there is plenty of the project as a whole a rigid deadline can make it suddenly become urgent at the last minute;

**Unrealistic** It is quite easy to write a project specification that sounds good, but is not grounded in the real world. A program that modelled the stock markets and thereby allowed you to predict how to manage your portfolio, or one to predict winning numbers in the national lottery, or one to play world-class chess. . . Now of course there are programs that play chess pretty well, and lots of people have made money out of the other two projects (in one case the statistics that one might apply is *much* easier than the other!), but the desirability of the finished program can astonishingly often blind one to the difficulties that would arise in trying to achieve it. In some cases a project might be achievable in principle but is beyond either today's technology or this week's budget, while in other cases the idea being considered might not even realistic given unlimited budget and time-scales. There are of course places where near-unreasonable big ideas can have a very valuable part to play: in a research laboratory a vision of one of these (currently) unrealistic goals can provide direction to the various smaller and better contained projects that each take tiny steps towards the ideal. At present my favourite example of something like this is the idea of *nanotechnology* with armies of molecular-scale robots working together to build their own heirs and successors. The standard example of a real project that many (most?) realistic observers judged to be utterly infeasible was the "Star Wars" Strategic Defence Initiative, but note that at that sort of level the political impact of even starting a project may be at least as important as delivery of a working product!

**Multi-platform** It is a luxury if a program only has to work on a single fixed computer system. Especially as projects become larger there is substantial extra effort required to keep them able to work smoothly on many different systems. This problem can show up with simple issues such as word-lengths, byte-arrangements in memory and compiler eccentricities, but it gets much worse as one looks at windowed user interfaces, multi-media functions, network drivers and support for special extra plug-in hardware;

**Long life-time** The easiest sort of program gets written one afternoon and is thrown away the next day. It does not carry any serious long-term support concerns with it. However other programs (sometimes still initially written in little more than an afternoon) end up becoming part of your life and get themselves worked and re-worked every few years. In my case the program I have that has the longest history was written in around 1972 in Fortran, based on me having seen one of that year's Diploma dissertations and having (partly unreasonably) convinced myself I could do better. The code was developed on Titan, the then University main machine. I took it to the USA with me when I spent a year there and tried to remove the last few bugs and make it look nicer. When the University moved up to an IBM mainframe I ran it on that, and at a much later stage I translated it (semi automatically) into BBC basic and ran it (very slowly) on a BBC micro. By last year I had the code in C with significant parts of the middle of it totally re-written, but with still those last few bugs to find ways of working around. If I had been able to predict when I started how long this would be of interest to me for maybe I would have tried harder to get it right first time! Note the radical changes in available hardware and sensible programming language over the lifetime of this program;

**User interface** For programs like modern word processors there is a real chance that almost all of the effort and a very large proportion of the code will go into supporting the fancy user interface, and trying to make it as helpful and intuitive as possible. Actually storing and editing the text could well be fairly straight forward. When the smoothness of a user interface is a serious priority for a project then the challenge of defining exactly what must happen is almost certainly severe too, and in significant projects will involve putting test users in special usability laboratories where their eye-movement can be tracked by cameras and their key-strokes can be timed. The fact that an interface provides lots of windows and pull-down menus does not automatically make it easy to use;

**Diverse users** Many commercial applications need to satisfy multiple users with diverse needs as part of a single coherent system. This can extend to new computer systems that need to interwork seamlessly with multiple existing operational procedures, including existing computer packages. Some users may be nervous of the new technology, while others may find excessive explanation an offensive waste of their time. The larger the number of interfaces needed and the wider the range of expectations the harder it will be to make a complete system deliver total satisfaction;

**Speed critical** Increasingly these days it makes sense to buy a faster computer if

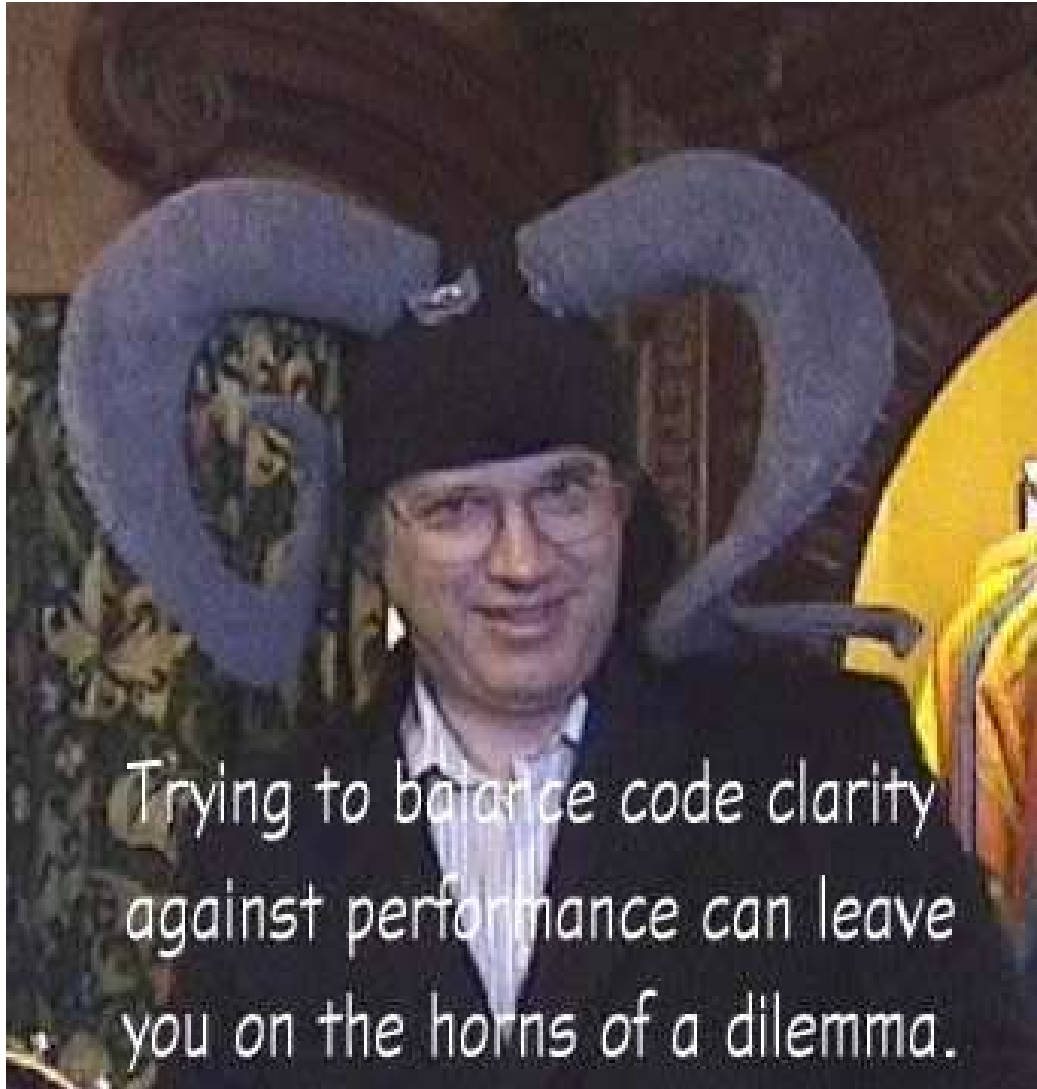


Figure 5.3: The search for speed can lead to eccentric-looking results.

some task seems to take a little longer than is comfortable. However there remain some areas where absolute performance is a serious issue and where getting the very best out of fixed hardware resources can give a competitive edge. The case most in my mind at present is that of (high security) encryption, where the calculations needed are fairly demanding but where there is real interest in keeping some control over the extra hardware costs that user are expected to incur. If speed requirements lead to need for significant assembly code programming (or almost equivalently to the design of task-specific silicon) then the resource requirements of a project can jump



dramatically. If in the other hand speed is of no importance at all for some task it may become possible to use a higher level programming system, simpler data structures and algorithms and generally save a huge amount of aggravation;

**Real time** Real-time responsiveness is characteristic of many control applications. It demands that certain external events be given a response within a pre-specified interval. At first this sounds like a variant on tasks that are just speed-critical, but the fine granularity at which performance is specified tends to influence the entire shape of software projects and rule out some otherwise sensible approaches. Some multi-media applications and video games will score highly in this category, as will engine management software for cars and flight control software for airports;

**Memory critical** A programming task can be made much harder if you are tight on memory. The very idea of being memory-limited can feel silly when we all know that it is easy to go out and buy another 64 Mbytes for (currently) of the order of £50<sup>2</sup>. But the computer in your cell-phone will have an amount of memory selected on the basis of a painful compromise between cost (measured in pennies), the power drain that the circuitry puts on the battery (and hence the expected battery life) and the set of features that can be supported. And the software developers are probably give the memory budget as a fixed quantity and invited to support as long a list of features as is at all possible within it;

**Add-on** A completely fresh piece of software is entitled to define its own file formats and conventions and can generally be designed and build without too much hindrance. But next year the extension to that original package is needed, or the new program is one that has to work gracefully with data from other people's programs. When building an add-on it is painfully often the case that the existing software base is not very well documented, and that the attempted new use of it reveals previously unknown bugs or limitations in the core system. Thus the effort that will need to be put into the second package may be much greater than would have been predicted based on experience from the first;

**Embedded** If the computer you are going to program is one in an electric egg-timer (or maybe a toy racing car, or an X-ray scanner) then testing may involve be a quite different experience from that you become used to when

---

<sup>2</sup>Last year these notes indicates 16 Mbytes for £50! I may have rounded prices up and down somewhat but still. . .

debugging ordinary applications that run on standard work-stations. In particular it may become necessary to become something of an expert in the hardware and electronics and also in the application area of the system within which your code will be placed;

**Tool-weak environment** This is a follow-on from the “embedded” heading, in that it is perhaps easiest to envisage an electric pop-up toaster where anything that slowed down or enlarged the code being run would perturb system timing enough to burn the toast, and where the target hardware is not automatically equipped with printers and flashing lights that can be used to help sense what is going on inside its CPU. For some such cases it is possible to buy or build real-time emulators or to wire in extra probes into a debuggable version of the hardware. There are other cases where either technology or budget mean that program development has to be done with a slow turn-around on testing and with only very limited ability to discover what happened when a bug surfaced. It is incredibly easy to simulate such a tool-weak environment for yourself by just avoiding the effort associated with becoming familiar with automated testing tools, debuggers and the like;

**Novel** One of the best and safest ways of knowing that a task is feasible is to observe that somebody else did it before, and their version was at least more or less satisfactory. The next best way is to observe that the new task is really rather similar to one that was carried out successfully in the past. This clearly leads to the obvious observation that if something is being attempted and there are no precedents to rely on then it becomes much harder to predict how well things will work out, and the chances of nasty surprises increases substantially.

There are two sort of program not listed above which deserve special mention. The first is the implementation of a known algorithm. This will usually end up as a package or a subroutine rather than a complete free-standing program, and there are plenty of algorithms that are complicated enough that programming them is a severe challenge. However the availability of a clear target and well specified direction will often make such programming tasks relatively tractable. It is however important to distinguish between programming up a complete and known algorithm (easyish) from developing and then implementing a new one, and uncomfortably often things that we informally describe as algorithms are in fact just strategies, and lots of difficult and inventive fine detail has to be filled into make them realistic.

The second special sort of program is the little throw-away one, and the recognition that such programs can be lashed together really fast and without any fuss

is important, since it can allow one to automate other parts of the program development task through strategic use of such bits of jiffy code.

## 5.2 Analysis and description of the objective

Sometimes a programming task starts with you being presented with a complete, precise and coherent explanation of exactly what has to be achieved. When this is couched in language so precise that there is not possible doubt about what is required you might like to ask why you are being asked to do anything, since almost all you need to do is to transcribe the specification into the particular syntax of the (specified) programming language. Several of the Part IA tickable problems come fairly close to this pattern, and there the reason you are asked to do them is exactly so you get practical experience with the syntax of the given language and the practical details of presenting programs to the computer. But that hardly counts as serious programming!

Assuming that we are not in one of these artificial cases, it is necessary to think about what one should expect to find in a specification and what does not belong there. It is useful to discuss the sorts of language used in specifications, and to consider who will end up taking prime responsibility for everything being correct.

A place to start is with the observation that a specification should describe what is wanted, rather than how the desired effect is to be achieved. This ideal can be followed up rather rapidly by the observation that it is often amazingly difficult to know what is really wanted, and usually quite a lot of important aspects of the full list of requirements will be left implicit or as items where you have to apply your own judgement. This is where it is useful to think back to the previous section and decide what style of project was liable to be intended and where the main pressure points are liable to be.

### 5.2.1 Important Questions

I have already given a check-list that should help work out what general class of problem you are facing. The next stage is to try to identify and concentrate on areas of uncertainty in your understanding of what has to be done. Furthermore initial effort ought to go into understanding aspects of the problem that are liable to shape the whole project: there is no point in agonising over cosmetic details until the big picture has become clear. Probably the best way of sorting this out is to imagine that some magic wand has been waved and it has conjured up a body of code and documentation that (if the fairy really was a good one!) probably does everything you need. However as a hard-headed and slightly cynical person

you need to check it first. Deciding what you are going to look for to see if the submitted work actually satisfied the project's needs can let you make explicit a lot of the previously slightly woolly expectations you might have. This viewpoint moves you from the initial statement "The program must achieve X" a little closer to "I must end up convinced that the program achieves X and here is the basis for how that conviction might be carried". Other things that might (or indeed might not) reveal themselves at this stage are:

1. Is user documentation needed, and if so how detailed is it expected to be? Is there any guess for how bulky the user manual will be?
2. How formal should documentation of the inner workings of the code be?
3. Was the implementation language to be used pre-specified, and if not what aspects of the problem or environment are relevant to the choice?
4. Is the initial specification a water-tight one or does the implementer have to make detailed design decisions along the way?

With regard to choice of programming language note that evidence from studies that have watched the behaviour of real programmers suggests that to a good first approximation it is possible to deliver the same number of lines of working documented code per week almost whatever language it is written in. A very striking consequence of this is that languages that are naturally concise and which provide built-in support for more of the high-level things you want to do can give major boosts to productivity.

The object of all this thought is to lead to a proper specification of the task. Depending on circumstances this may take one of a number of possible forms:

### **5.2.2 Informal specifications**

Documents written in English, however pedantically phrased and however voluminous, must be viewed as informal specifications. Those who have a lot of spare time might try reading the original description of the language C[16] where Appendix A is called a reference manual and might be expected to form a useful basis for fresh implementations of the language. Superficially it looks pretty good, but it is only when you examine the careful (though still "informal" in the current context) description in the official ANSI standard[22] that it becomes clear just how much is left unsaid in the first document. Note that ANSI C is not the same language as that defined by Kernighan and Ritchie, and so the two documents just mentioned can not be compared quite directly, and also be aware that spotting and making clear places where specifications written in English are not precise

is a great skill, and one that some people enjoy exercising more than others do! The description in section 5.18 is another rather more manageable example of an informal textual specification. When you get to it you might like to check to see what it tells you what to do about tabs and back-spaces, which are clearly characters that have an effect on horizontal layout. What? It fails to mention them? Oh dear!

### 5.2.3 Formal descriptions

One response to the fact that almost all informal specifications are riddled with holes (not all of which will be important: for instance it might be taken as understood by all that messages that are printed so that they look like sentences should be properly spelt and punctuated) has been to look for ways of using formal description languages. The ZED language (developed at Oxford<sup>3</sup>, and sometimes written as just Z) is one such and has at times been taught in Software Engineering courses here. The group concerned with the development of the language ML were keen to use formal mathematically-styled descriptive methods to define exactly what ML ought to do in all possible circumstances. Later on in the CST there are whole lecture courses on Specification and Verification and so I am not going to give any examples here, but will content myself by observing that a good grounding in discrete mathematics is an absolute pre-requisite for anybody thinking of working this way.

### 5.2.4 Executable specifications

One group of formal specification enthusiasts went off and developed ever more powerful mathematical notations to help them describe tasks. Another group observed that sometimes a careful description of what must be achieved looks a bit like a program in a super-compact super-high-level programming language. It may not look like a realistic program, in that it may omit lots of explanation about how objectives should be achieved and especially how they should be achieved reasonably efficiently. This leads to the idea of an *executable specification*, through building an implementation of the specification language. This will be permitted to run amazingly slowly, and its users will be encouraged to go all out for clarity and correctness. To give a small idea of what this might entail, consider the job of specifying a procedure to sort some data. The initial informal specification might be that the output should be a re-ordering of the input such that the values in the output be in non-descending order. An executable specification might consist of three components. The first would create a list of all the different

---

<sup>3</sup><http://www.comlab.ox.ac.uk/oucl/prg.html>

permutations of the input. The second would be a procedure to inspect a list and check to see if its elements were in non-descending order. The final part would compose these to generate all permutations then scan through them one at a time and return the first non-descending one found. This would not be a good practical sorting algorithm, but could provide a basis for very transparent demonstrations that the process shown did achieve the desired goal! It should be remembered that an executable specification needs to be treated as such, and not as a model for how the eventual implementation will work. A danger with the technique is that it is quite easy for accidental or unimportant consequences of how the specification is written to end up as part of the project requirements.

### 5.3 Ethical Considerations

Quite early on when considering a programming project you need to take explicit stock of any moral or ethical issues that it raises. Earlier in the year you have had more complete coverage of the problems of behaving professionally, so here I will just give a quick check-list of some of the things that might give cause for concern:

1. Infringement of other people's intellectual property rights, be they patents, copyright or trade secrets. Some companies will at least try to prevent others from creating new programs that look too much like the original. When licensed software is being used the implications of the license agreement may become relevant;
2. Responsibility to your employer or institution. It may be that certain sorts of work are contrary to local policy. For instance a company might not be willing to permit its staff to politically motivated virtual reality simulations using company resources, and this University has views about the commercial use of academic systems;
3. A computing professional has a responsibility to give honest advice to their "customer" when asked about the reasonableness or feasibility of a project, and to avoid taking on work that they know they are not qualified to do;
4. It can be proper to take a considered stance against the development of systems that are liable to have a seriously negative impact on society as a whole. I have known some people who consider this reason to avoid any involvement with military or defence-funded computing, while others will object to technology that seems especially liable to make tracking, surveillance or eavesdropping easier. Those of you with lurid imaginations can no

doubt envisage plenty more applications of computers that might be seen as so undesirable that one should if necessary quit a job rather than work on them.

## 5.4 How much of the work has been done already?

The points that I have covered so far probably do not feel as if they really help you get started when faced with a hard-looking programming task, although I believe that working hard to make sure you really understand the specification you are faced with is in fact always a very valuable process. From now onwards I move closer to the concrete and visible parts of the programming task. The first question to ask here is “Do I actually have to do this or has it been done before?”

There are three notable cases where something has been done before but it is still necessary to do it again. Student exercises are one of these, and undue reliance on the efforts of your predecessors is gently discouraged. Sometimes a problem has been solved before, but a solution needs to be re-created without reference to the original version because the original is encumbered with awkward commercial<sup>4</sup> restrictions or is not locally available. The final cause for re-implementation is if the previous version of the program concerned was a failure and so much of a mess that any attempt to rely on it would start the new project off in wrong-minded directions.

Apart from these cases the best way to write any program at all is to adopt, adapt and improve as much existing technology as you can! This can range from making the very second program that you ever write a variation on that initial “Hello World” example you were given through to exploiting existing large software libraries. The material that can be reclaimed may be as minor as a bunch of initial comments saying who you (the author) are and including space to describe what the program does. It might just be some stylised “import” statements needed at the head of almost any program you write. If you need a tree structure in today’s program do you have one written last week which gives you the data type definition and some of the basic operations on trees? Have you been provided with a collection of nice neat sample programs (or do you have a book or CD ROM with some) that can help? Many programming languages are packaged with a fairly extensive collection of chunks of sample code.

Most programming languages come with standardised libraries that (almost always) mean there is no need for you to write your own sorting procedure or code to convert floating point values into or out of textual form. In many important areas

---

<sup>4</sup>Remember that if the restriction is in the form of a patent then no amount of re-implementation frees you from obligations to the patent-owner, and in other cases you may need to be able to give a very clear demonstration that your new version really has been created completely independently.

there will be separate libraries that contain much much more extensive collections of facilities. For instance numerical libraries (eg the one from NAG) are where you should look for code to solve sets of simultaneous equations or to maximise a messy function of several variables. When you need to implement a windowed interface with pull-down menus and all that sort of stuff again look to existing library code to cope with much of the low-level detail for you. Similarly for data compression, arbitrary precision integer arithmetic, image manipulation. . .

Observing that there is a lot of existing support around does not make the problem of program construction go away: knowing what existing code is available is not always easy, and understanding both how to use it and what constraints must be accepted if it is used can be quite a challenge. For instance with the NAG (numerical) library it may take beginners quite a while before they discover that `E04ACF` (say, and not one of the other half-dozen closely related routines) is the name of the function they need to call and before they understand exactly what arguments to pass to it.

As well as existing pre-written code (either in source or library form) that can help along with a new project there are also packages that write significant bodies of code for you, basing what they do one on either a compact descriptive input file or interaction with the user through some clever interface. The well-established examples of this are the tools `yacc` and `lex` that provide a convenient and reliable way of creating parsers. Current users of Microsoft's Visual C++ system will be aware of the so-called "Wizards" that it provides that help create code to implement the user interface you want, and there are other commercial program generators in this and a variety of business application areas. To use one of these you first have to know of its availability, and then learn how to drive it: both of these may involve an investment of time, but with luck that will be re-paid with generous interest even on your first real use. In some cases the correct use of a program generation tool is to accept its output uncritically, while on other occasions the proper view is to collect what it creates, study it and eventually adjust the generated code until you can take direct responsibility for subsequent support. Before deciding which to do you need to come to a judgement about the stability and reliability of the program generator and how often you will need to adjust your code by feeding fresh input in to the very start.

Another way in which existing code can be exploited is when new code is written so that it converts whatever input it accepts into the input format for some existing package, one that solves a sufficiently related problem that this makes some sense. For instance it is quite common to make an early implementation of a new programming language work by translating the new language into some existing one and then feeding the translated version into an existing compiler. For early versions of ML the existing language was Lisp, while for Modula 3 some compilers work by converting the Modula 3 source into C. Doing this may result



in a complete compiler that is slower and bulkier than might otherwise be the case, but it can greatly reduce the effort in building it.

## 5.5 What skills and knowledge are available?



Figure 5.4: Nancy Silverton’s bakery is in La Brea near the tar pits, and her book (*Bread from the La Brea Bakery*) is unutterably wonderful. I like her chocolate-cherry loaf. This photo is of the racks in her shop. Not much about Java I agree but baking good bread is at least as important to know about as computing.

A balance needs to be drawn between working through a programming project using only the techniques and tools that you already know and pushing it forward using valuable but unfamiliar new methods. Doing something new may slow you down substantially, but an unwillingness to accept that toll may lead to a very pedestrian style of code development using only a limited range of idioms. There is a real possibility that short-term expediency can be in conflict with longer term productivity. Examples where this may feel a strain include use of formal methods, new programming languages and program generation tools. The main point to be got across here is that almost everything to do with computers changes every five years or so, and so all in the field need to invest some of their effort in continual personal re-education so that their work does not look too much as if it has been chipped out using stone axes. The good news is that although detailed technology changes the skills associated with working through a significant project should grow with experience, and the amount of existing code that an old hand will have to pillage may be quite large, and so there is a reasonable prospect for a long term future for those with skills in software design and construction. Remember that all the books on Software Engineering tell us that the competence of

the people working on a project can make more difference to its success than any other single factor.

It is useful to have a conscious policy of collecting knowledge about what can be done and where to find the fine grubby details. For example the standard textbook[9] contains detailed recipes for solving all sorts of basic tasks. Only rarely will any one of these be the whole of a program you need to write, but quite often a larger task will be able to exploit one or more of them. These and many of the other topics covered in the CST are there because there is at least a chance that they may occasionally be useful! It is much more important to know what can be done than how to do it, because the *how* can always be looked up when you need it.

## 5.6 Design of methods to achieve a goal

Perhaps the biggest single decision to be made when starting the detailed design of a program is where to begin. The concrete suggestions that I include here are to some extent caricatures; in reality few real projects will follow any of them totally but all should be recognisable as strategies. The crucial issue is that it will not be possible to design or write the whole of a program at once so it is necessary to split the work into phases or chunks.

### 5.6.1 Top-Down Design

In Top Down Design work on a problem starts by writing a “program” that is just one line long. Its text is:

```
{ solveMyProblem(); }
```

where of course the detailed punctuation may be selected to match the programming language being used. At this stage it is quite reasonable to be very informal about syntax. A next step will be to find some way of partitioning the whole task into components. Just how these components will be brought into existence is at present left in the air, however if we split things up in too unreasonable a way we will run into trouble later on. For many simple programs the second stage could look rather like:

```
/* My name, today's date, purpose of program */  
import Standard-libraries;  
{  
    /* declare variables here */  
    data = readInData();  
    results = calculate(data);  
}
```

```
    displayResults(results);  
}
```

The ideal is that the whole development of the program should take place in baby-sized steps like this. At almost every stage there will be a whole collection of worrying-looking procedures that remain undefined and not yet thought about, such as `Calculate` above. It is critical not to worry too much about these, because each time a refinement is made although the number of these unresolved problems may multiply the expected difficulty of each will reduce. Well it had better, since all the ones that you introduce should be necessary steps towards the solution of your whole original task, and it makes sense to expect parts to be simpler than the whole.

After a rather few steps in the top-down development process one should expect to have a fully syntactically correct main program that will not need any alterations later as the low level details of the procedures that it calls get sorted out. And each of the components that remain to be implemented should have a clearly understood purpose (for choice that should be written down) and each such component should be clearly separated from all the others. That is not to say that the component procedures might not call each other or rely on what they each can do, but the internal details of any one component should not matter to any other. This last point helps focus attention on interfaces. In my tiny example above the serious interfaces are represented by the variables `data` and `results` which pass information from one part of the design to the next. Working out exactly what must be captured in these interfaces would be generally need to be done fairly early on. After enough stages of elaboration the bits left over from top-down design are liable to end up small enough that you just code them up without need to worry: anything that is trivial you code up, anything that still looks murky you just apply one more expansion step to. With luck eventually the process ends.

There are two significant worries about top-down design. These are “How do I know how to split the main task up?” and “But I can’t test me code until everything is finished!”. Both of these are proper concerns.

Splitting a big problem up involves finding a strategy for solving it. Even though this can be quite hard, it is almost always easier to invent a high-level idea for how to solve a problem than it is to work through all the details, and this is what top-down programming is all about. In many cases sketching on a piece of paper what you would do if you had to solve the problem by hand (rather than by computer) can help. Quite often the partition of a problem you make may end up leading your design into some uncomfortable dead end. In that case you need to look back and see which steps in your problem refinement represented places where you had real choice and which ones were pretty much inevitable. It is then necessary to go back to one of the stages where a choice was possible and to re-think things in the light of your new understanding. To make this process sensible

you should refuse to give up fleshing out one particular version of a top-down design until you are in a position to give a really clear explanation of why the route you have taken represents failure, because without this understanding you will not know how far back you need to go in the re-planning. As an example of what might go wrong, the code I sketched earlier here would end up being wrongly structured if user interaction was needed, and that interaction might be based on evaluation of partial results. To make that sort of interface possible it might be necessary to re-work the design as (say)

```
/* My name, today's date, purpose of program */
import Standard-libraries;
{
  /* declare variables here */
  /* set empty data and results */
  while not finished do
  {
    extra = readInMoreData();
    if (endOfUserInput(extra)) finished = true;
    else
    {
      data = combine(data, extra);
      results = upDateResults(results, data);
      displaySomething(results);
    }
  }
  displayFinalResults(results);
}
```

which is clearly getting messier! And furthermore my earlier and shorter version looked generally valid for lots of tasks, while this one would need careful extra review depending on the exact for of user interaction required.

There is a huge amount to be said in favour of being able to test a program as it is built. Anybody who waits right to the end will have a dreadful mix of errors at all possible levels of abstraction to try to disentangle. At first sight it seems that top-down design precludes any early testing. This pessimism is not well founded. The main way out is to write *stubs* of code that fill in for all the parts of your program that have not yet been written. A stub is a short and simple piece of code that takes the place of something that will later on be much more messy. It does whatever is necessary to simulate some minimal behaviour that will make it possible to test the code around it. Sometimes a stub will just print a warning message and stop when it gets called! On other occasions one might make a stub print out its parameters and wait for human intervention: it then reads something back in, packages it up a bit and returns it as a result. The human assistant actually

did all the clever work.

There are two other attitudes to take to top-down design. One of these is to limit it to **design** rather than implementation. Just use it to define a skeletal shape for your code, and then make the coding and testing a separate activity. Obviously this only makes sense when you have enough confidence that you can be sure that the chunks left to be coded will in fact work out well. The final view is to think of top-down design as an ideal to be retrofitted to any project once it is complete. Even if the real work on a project went in fits and starts with lots of false trails and confusion, there is a very real chance that it can be rationalised afterwards and explained top-down. If that is done then it is almost certain that a clear framework has been built for anybody who needs to make future changes to the program.

### 5.6.2 Bottom-Up Implementation

Perhaps you are uncertain about exactly what your program is going to do or how it will solve its central problems. Perhaps you want to make sure that every line of code you ever write is documented, tested and validated to death before you move on from it and certainly before you start relying on it. Well these concerns lead you towards a bottom-up development strategy. The idea here is to identify a collection of smallish bits of functionality that will (almost) certainly be needed as part of your complete program, and to start by implementing these. This avoids having to think about the hard stuff for a while. For instance a compiler-writer might start by writing code to read in lines of program and discard comments, or to build up a list of all the variable names seen. Somebody starting to write a word processor might begin with pattern-matching code ready for use in search-and-replace operations. In almost all large projects there are going to be quite a few fundamental units of code that are obviously going to be useful regardless of the high level structure you end up with.

The worry with bottom-up construction is that it does not correspond to having any overall vision of the final result. That makes it all too easy to end up with a collection of ill-co-ordinated components that do not quite fit together and that do not really combine to solve the original problem. At the very least I would suggest a serious bout of top-down design effort be done before any bottom-up work to try to put an overall framework into place. There is also a clear prospect that some of the units created during bottom-up work may end up not being necessary after all so the time spent on them was wasted.

An alternative way of thinking about bottom-up programming can soften the impact of these worries. It starts by viewing a programming language not just as a collection of fragments of syntax, but as a range of ways of structuring data and of performing operations upon it. The fact that some of these operations happen to be hard-wired into the language (as integer arithmetic usually is) while others exist as

collections of subroutines (floating point arithmetic on 3000-digit numbers would normally be done that way) is of secondary importance. Considered this way each time you define a new data type or write a fresh procedure you have extended and customised your programming language by giving it support for something new. Bottom-up programming can then be seen as gradually building layer upon layer of extra support into your language until it is rich in the operations most important in your problem area. Eventually one then hopes that the task that at first had seemed daunting becomes just half a dozen lines in the extended language. If some of the procedures built along the way do not happen to be used this time, they may well come in handy the next time you have to write a program in the same application area, so the work they consumed has not really been wasted after all. The language Lisp is notable for having sustained a culture based on this idea of language extension.

### 5.6.3 Data Centred Programming

Both top-down and bottom-up programming tend to focus on what your program looks like and the way in which it is structured into procedures. An alternative is to concentrate not on the actions performed by the code but on the way in which data is represented and the history of transformations that any bit of data will be subject to. These days this idea is often considered almost synonymous with an Object Oriented approach where the overall design of the class structure for a program is the most fundamental feature that it will have. Earlier (and pre-dating the widespread use of Object Oriented languages) convincing arguments for design based on the entities that a program must manipulate or model come from Jackson Structured Programming and Design[8]. More recently SSADM[3] has probably become one of the more widespread design and specification methodologies for commercial projects.

### 5.6.4 Iterative Refinement

My final strategy for organising the design of a complete program does not even expect to complete the job in one session. It starts by asking how the initial problem can be restricted or simplified to make it easier to address. And perhaps it will spot how the most globally critical design decisions for the whole program could be made in two or three different ways, with it hard to tell in advance which would work out best in the end. The idea is then to start with code for a scruffy mock-up of a watered down version of the desired program using just one of these sets of design decisions. The time and effort needed to write a program grows much faster than linearly with the size of the program: the natural (but less obvious) consequence of this is that writing a small program can be **much** quicker and

easier than completing the full version. It may in some cases make sense even to write several competing first sketches of the code. When the first sketch version is working it is possible to step back and evaluate it, to see if its overall shape is sound. When it has been adjusted until it is structurally correct, effort can go into adding in missing features and generally upgrading it until it eventually gets transformed into the beautiful butterfly that was really wanted. Of all the methods that I have described this is the one that comes closest to allowing for “experimental” programming. The discipline to adhere to is that experiments are worthy of that tag if the results from them can be evaluated and if something can thus be learned from them.

### **5.6.5 Which of the above is best?**

The “best” technique for getting a program written will depend on its size as well as its nature. I think that puritanical adherence to any of the above would be unreasonable, and I also believe that inspiration and experience (and good taste) have important roles to play. However if pushed into an opinion I will vote for presenting a design or a program (whether already finished or still under construction) as if it were prepared top-down, with an emphasis on the early design of what information must be represented and where it must pass from one part of the code to another.

## **5.7 How do we know it will work?**

Nobody should ever write a program unless they have good reason to believe that it ought to work. It is of course proper to recognise that it will not work, because typographic errors and all sorts of oversights will ensure that. But the code should have been written so that in slightly idealised world where these accidental imperfections do not exist it would work perfectly. Blind and enthusiastic hope is not sufficient to make programs behave well, and so any proper design needs to have lurking behind it the seeds of a correctness proof. In easy-going times this can remain untended as little comments that can just remind you of your thinking. When a program starts to get troublesome it can be worth growing these comments into short essays that explain what identities are being preserved intact across regions of code, why your loops are guaranteed to terminate and what assumptions about data are important, and why. In yet more demanding circumstances it can become necessary to conduct formal validation procedures for code.

The easiest advice to give here is that before you write even half a dozen lines of code you should write a short paragraph of comment that explains what the code is intended to achieve and why your method will work. The comment

should usually not explain **how** it works (the code itself is all about “how”), but **why**. To try to show that I (at least sometimes!) follow this advice here is a short extract from one of my own<sup>5</sup> programs...

```

/*
 * Here is a short essay on the interaction between flags
 * and properties. It is written because the issue appears
 * to be delicate, especially in the face of a scheme that
 * I use to speed things up.
 * (a) If you use FLAG, REMFLAG and FLAGP with some
 *     indicator then that indicator is known as a flag.
 * (b) If you use PUT, REMPROP and GET with an indicator
 *     then what you have is a property.
 * (c) Providing the names of flags and properties are
 *     disjoint no difficulty whatever should arise.
 * (d) If you use PLIST to gain direct access to property
 *     lists then flags are visible as pairs (tag . t) and
 *     properties as (tag . value).
 * (e) Using RPLACx operations on the result of PLIST may
 *     cause system damage. It is considered illegal.
 *     Also changes made that way may not be matched in
 *     any accelerating caches that I keep.
 * (f) After (FLAG '(id) 'tag) [when id did not previously
 *     have any flags or properties] a call (GET 'id 'tag)
 *     will return t.
 * (g) After (PUT 'id 'tag 'thing) a call (FLAGP 'id
 *     'tag) will return t whatever the value of "thing".
 *     A call (GET 'id 'tag) will return the saved value
 *     (which might be nil). Thus FLAGP can be thought of
 *     as a function that tests if a given property is
 *     attached to a symbol.
 * (h) As a consequence of (g) REMPROP and REMFLAG are
 *     really the same operation.
 */

```

```

Lisp_Object get(Lisp_Object a, Lisp_Object b)
{
    Lisp_Object pl, prev, w, nil = C_nil;
    int n;
}

```

---

<sup>5</sup>This happens to be written in C rather than Java, but since most of it is comment maybe that does not matter too much.



```

* In CSL mode plists are structured like association
* lists, and NOT as lists with alternate tags and values.
* There is also a bitmap that can provide a fast test for
* the presence of a property...
*/
    if (!symbolp(a))
    {
#ifdef RECORD_GET
        record_get(b, NO);
        errexit();
#endif
        return onevalue(nil);
    }
    ... etc etc

```

The exact details of what I am trying to do are not important here, but the evidence of mind-clearing so that there is a chance to get the code correct first time is. Note how little the comment before the procedure has to say about low-level implementation details, but how much about specifications, assumptions and limitations.

I would note here that typing a program in is generally one of the least time-consuming parts of the whole programming process, and these days disc storage is pretty cheap, and thus various reasons which in earlier days may have discouraged layout and explanation in code no longer apply.

Before trying code and as a further check that it ought to work it can be useful to “walk through” the code. In other words to pretend to be a computer executing it and see if you follow the paths and achieve the results that you were supposed to. While doing this it can be valuable to think about which paths through the code are common and which are not, since when you get to testing it may be that the uncommon paths do not get exercised very much unless you take special steps to cause them to be activated.

The “correctness” that you will be looking for can be at several different levels. A *partially correct* program is one that can never give an incorrect answer. This sounds pretty good until you recognise that there is a chance that it may just get stuck in a loop and thereby never give any answer at all! It is amazingly often much easier to justify that a program is partially correct than to go the whole hog and show it is correct, ie that not only is it partially correct but that it will always terminate. Beyond even the requirements of correctness will be performance demands: in some cases a program will need not only to deliver the right answers but to meet some sort of resource budget. Especially if the performance target is specified as being for performance that is good “on the average” it can be dreadfully hard to prove, and usually the only proper way to start is by designing and justifying algorithms way before any mention of actual programming arises.

A final thing to check for is the possibility that your code can be derailed by unhelpful erroneous input. For instance significant security holes in operating systems have in the past been consequences of trusted modules of code being too trusting of their input, and them getting caught out by (eg) input lines so long that internal buffers overflowed thereby corrupting adjacent data.

The proper mind-set to settle in to while designing and starting to implement code is pretty paranoid: you want the code to deliver either a correct result or a comprehensible diagnostic whenever anything imaginable goes wrong in either the data presented to it or its own internal workings. This last statement leads to a concrete suggestion: make sure that the code can test itself for sanity and correctness every so often and insert code that does just that. The assertions that you insert will form part of your argument for why the program is supposed to work, and can help you (later on) debug when it does not.

## 5.8 While you are writing the program

Please remember to get up and walk around, to stretch, drink plenty of water, sit up straight and all the other things mentioned at the Learning Day as relevant occupational health issues. My experience is that it is quite hard to do effective programming in 5 minute snippets, but that after a few hours constant work productivity decreases. A pernicious fact is that you may not notice this decrease at the time, in that the main way in which a programmer can become unproductive is by putting more bugs into a program. It is possible to keep churning out lines of code all through the night, but there is a real chance that the time you will spend afterwards trying to mend the last few of them will mean that the long session did not really justify itself.

In contrast to programming where long sessions can do real damage (because of the bugs that can be added by a tired programmer) I have sometimes found that long sessions have been the only way I can isolate bugs. Provided I can discipline myself not to try to correct anything but the very simplest bug while I am tired a long stretch can let me chase bugs in a painstakingly logical way, and this is sometimes necessary when intuitive bug-spotting fails.

Thus my general advice about the concrete programming task would be to schedule your time so you can work in bursts of around an hour per session, and that you should plan your work so that as much as possible of everything you do can be tested fairly enthusiastically while it is fresh in your mind. A natural corollary of this advice is that projects should always be started in plenty of time, and pushed forward consistently so that no last-minute panic can arise and force sub-optimal work habits.

## 5.9 Documenting a program or project

Student assessed exercises are expected to be handed in complete with a brief report describing what has been done. Larger undergraduate projects culminate in the submission of a dissertation, as do PhD studies. All commercial programming activities are liable to need two distinct layers of documentation: one for the user and one for the people who will support and modify the product in the future. All these facts serve to remind us that documentation is an intrinsic part of any program.

Two overall rules can guide the writing of good documentation. The first is to consider the intended audience, and think about what they need to know and how your document can be structured to help them find it. The second is to keep a degree of consistency and order to everything: documents with a coherent overall structure are both easier to update and to browse than sets of idiosyncratic jottings.

To help with the first of these, here are some potential styles of write-up that might be needed:

1. Comments within the code to remind yourself or somebody who is already familiar with the program exactly what is going on at each point in it;
2. An overview of the internal structure and organisation of the whole program so that somebody who does not already know it can start to find their way around;
3. Documentation intended to show how reliable a program is, concentrating on discussions of ways in which the code has been built to be resilient in the face of unusual combinations of circumstance;
4. A technical presentation of a program in a form suitable for publication in a journal or at a conference, where the audience will consist of people expert in the general field but not aware of exactly what your contribution is;
5. An introductory user manual, intended to make the program usable even by the very very nervous;
6. A user reference manual, documenting clearly and precisely all of the options and facilities that are available;
7. On-line help for browsing by the user while they are trying to use the program;
8. A description of the program suitable for presentation to the venture capitalists who are considering investing in the next stage of its development.

It seems inevitable that the above list is not exhaustive, but my guess is that most programs could be presented in any one of the given ways, and the resulting document would be quite different in each case. It is not that one or the other of these styles is inherently better or more important than another, more that if you write the wrong version you will either not serve your reader well or you will find that you have had to put much more effort into the documentation than was really justified.

A special problem about documentation is that of when it should be written. For small projects at least it will almost always be produced only after the program has been (at least nearly) finished. This can be rationalised by claiming “how can I possibly document it before it exists?”

I will argue here for two ideals. The first is that documentation ought to follow on from design and specification work, but precede detailed programming. The second is that the text of the documentation should live closely linked to the developing source code. The reasoning behind the first of these is that writing the text can really help to ensure that the specification of the code has been fully thought through, and once it is done it provides an invaluable stable reference to keep the detailed programming on track. The second point recognises some sort of realism, and that all sorts of details of just what a program does will not be resolved until quite late in the implementation process. For instance the exact wording of messages that are printed will often not be decided until then, and it will certainly be hard to prepare sample transcripts from the use of the program ahead of its completion<sup>6</sup>. Thus when the documentation has been written early it will need completing when some of these final details get settled and correcting when the code is corrected or extended. The most plausible way of making it feasible to keep code and description in step is to keep them together. The concept of Literate Programming[17] pursues this goal. A program is represented as a composite file that can be processed in (at least) two different ways. One way “compiles” it to create typeset-quality human readable documentation, while the other leaves just statements in some quite ordinary programming language ready to be fed into a compiler. This goes beyond just having copious comments in the code in two ways. Firstly it expects that the generated documentation should be able to exploit the full range of modern typography and that it can include pictures or diagrams where relevant. It is supposed to end up as cleanly presented and readable as any fully free-standing document could ever be. Secondly Literate Programming recognises that the ordering and layout of the program that has to be compiled may not be the same as that in the ideal manual, and so the disentangling tool needs to be able to rearrange bits of text in a fairly flexible way so that description can simultaneously be thought of as close to the code it relates to and

---

<sup>6</sup>Even though these samples can be planned and sketched early.

to the section in the document where it belongs. This idea was initially developed as part of the project to implement the  $\text{\TeX}$  typesetting program that is being used to prepare these lecture notes.

## 5.10 How do we know it does work?



Figure 5.5: Many people think that their work is over well before it actually is.

A conceptual difficulty that many people suffer from is a confusion between whether a program should work and whether it does. A program should work if it has been designed so that there are clear and easily explained reasons why it can achieve what it should. Sometimes the term “easily explained” may conceal the mathematical proof of the correctness of an algorithm, but at least in theory it

would be possible to talk anybody through the justification. As to programs that actually do work, well the reality seems to be that the only ones of these that you will ever see will be no more than around 100 lines long: empirically any program much longer than that will remain flawed even after extensive checking. Proper Oriental rugs will always have been woven with a deliberate mistake in them, in recognition of the fact that only Allah is perfect. Experience has shown very clearly indeed that in the case of writing programs we all have enough failings that there is no great need to insert extra errors — there will be plenty inserted however hard we try to avoid them. Thus (at least at the present state of the art) there is no such thing as a (non-trivial) program that works.

If, however, a program *should* work (in the above sense) then the residual errors in it will be ones that can be corrected without disturbing the concepts behind it or its overall structure. I would like to think of such problems as “little bugs”. The fact that they are little does not mean that they might not be important, in that missing commas or references to the wrong variable can cause aeroplanes to crash just as convincingly as can errors at a more conceptual level. But the big effort must have been to get to a first testable version of your code with only little bugs left in it. What is then needed is a testing strategy to help locate as many of these as possible. Note of course that testing can only ever generate evidence for the presence of a bug: in general it can not prove absence. But careful and systematic testing is something we still need whenever there has been human involvement in the program construction process<sup>7</sup>.

The following thoughts may help in planning a test regime:

1. Even obvious errors in output can be hard to notice. Perhaps human society has been built up around a culture of interpreting slightly ambiguous input in the “sensible” way, and certainly we are all very used to seeing what we expect to see even when presented with something rather different. By the time you see this document I will have put some effort into checking its spelling, punctuation, grammar and general coherence, and I hope that you will not notice or be upset by the residual mistakes. But anybody who has tried serious proof-reading will be aware that blatant mistakes can emerge even when a document has been checked carefully several times;
2. If you are checking your own code and especially if you know you can stop work once it is finished then you have a clear incentive **not** to notice mistakes. Even if a mistake you find is not going to cause you to have to spend time fixing it it does represent you having found yet another instance of your own lack of attention, and so it may not be good for your ego;

---

<sup>7</sup>Some see this observation as a foundation for hope for the future

3. It is very desirable to make a clear distinction between the job of testing a program to identify the presence of bugs and the separate activity of correcting things. It can be useful to take the time to try to spot as many mistakes as you can before changing anything at all;
4. A program can contain many more bugs and oddities than your worst nightmares would lead you to believe!
5. Testing strategies worked out as part of the initial design of a program are liable to be better than ones invented only once code has been completed;
6. It can be useful to organise explicit test cases for extreme conditions that your program may face (eg sorting data where all the numbers to be sorted have the same value), and to collect test cases that cause each path through your code to be exercised. It is easy to have quite a large barrage of test cases but still have some major body of code unvisited.
7. Regressions tests are a good thing. These are test cases that grow up during project development, and at each stage after any change is made all of them are re-run, and the output the produce is checked. When any error is detected a new item in the regression suite is prepared so that there can remain a definite verification that the error does not re-appear at some future stage. Automating the application of regression tests is a very good thing, since otherwise laziness can too easily cause one to skip running them;
8. When you find one bug you may find that its nature gives you ideas for other funny cases to check. You should try to record your thoughts so that you do not forget this insight;
9. Writing extra programs to help you test your main body of code is often a good investment in time. On especially interesting scheme is to generate pseudo-random test cases. I have done that while testing a polynomial factorising program and suffered randomly-generated tests of a C compiler I was involved with, and in each case the relentless random coverage of cases turned out to represent quite severe stress;
10. You do not know how many bugs your code has in it, so do not know when to stop looking. One theoretical way to attack this worry would be to get some fresh known bugs injected into your code before testing, and then see what proportion of the bugs found were the seeded-in ones and which had been original. That may allow you to predict the total bug level remaining.

Having detected some bugs there are several possible things to do. One is to sit tight and hope that nobody else notices! Another is to document the deficiencies

at the end of your manual. The last is to try to correct some of them. The first two of these routes are more reasonable than might at first seem proper given that correcting bugs so very often introduces new ones.

In extreme cases it may be that the level of correctness that can be achieved by bug-hunting will be inadequate. Sometimes it may then be possible to attempt a formal proof of the correctness of your code. In all realistic circumstances this will involve using a large and complicated proof assistant program to help with all the very laborious details involved. Current belief is that it will be very unusual for bugs in the implementation of this tool to allow you to end up with a program that purports to be proved but which in fact still contains mistakes!

## 5.11 Is it efficient?

I have made this a separate section from the one on detecting the presence of errors because performance effects are only rarely the result of simple oversights. Let me start by stressing the distinction between a program that is expensive to run (eg the one that computes  $\pi$  to 20,000,000,000 decimal places) and ones that are inefficient (eg one that takes over half a second to compute  $\pi$  correct to four places). The point being made is that unless you have a realistic idea of how long a task ought to take it is hard to know if your program is taking a reasonable amount of time. And similarly for memory requirements, disc I/O or any other important resource. Thus as always we are thrown back to design and specification time predictions as our only guideline, and sometimes even these will be based on little more than crude intuition.

If a program runs fast enough for reasonable purposes then there may be no benefit in making it more efficient however much scope for improvement there is. In such cases avoid temptation. It is also almost always by far best to concentrate on getting code correct first and only worry about performance afterwards, taking the view that a wrong result computed faster is still wrong, and correct results may be worth waiting for.

When collecting test cases for performance measurements it may be useful to think about whether speed is needed in every single case or just in most cases when the program is run. It can also be helpful to look at how costs are expected to (and do) grow as larger and larger test cases are attempted. For most programming tasks it will be possible to make a trade between the amount of time a program takes to run and the amount of memory it uses. Frequently this shows up in a decision as to whether some value should be stored away in case it is needed later or whether any later user should re-calculate it. Recognising this potential trade-off is part of performance engineering.

For probably the majority of expensive tasks there will be one single part of the



entire program that is responsible for by far the largest amount of time spent. One would have expected that it would always be easy to predict ahead of time where that would be, but it is not! For instance when an early TITAN Fortran compiler was measured in an attempt to discover how it could be speeded up it was found that over half of its entire time was spent in a very short loop of instructions that were to do with discarding trailing blanks from the end of input lines. Once the programmers knew that it was easy to do something about it, but one suspects they were expecting to find a hot-spot in some more arcane part of the code. It is thus useful to see if the languages and system you use provide instrumentation that makes it easy to collect information to reveal which parts of your code are most critical. If there are no system tools to help you you may be able to add in time-recording statements to your code so it can collect its own break-down to show what is going on. Cunning optimisation of bits of code that hardly ever get used is probably a waste of effort.

Usually the best ways to gain speed involve re-thinking data structures to provide cheap and direct support for the most common operations. This can sometimes mean replacing a very simple structure by one that has huge amounts of algorithmic complexity (there are examples of such cases in the Part IB Complexity course and the Part II one on Advanced Algorithms). In almost all circumstances a structural improvement that gives a better big-O growth rate for some critical cost is what you should seek.

In a few cases the remaining constant factor improvement in speed may still be vital. In such cases it may be necessary to re-write fragments of your code in less portable ways (including the possibility of use of machine code) or do other things that tend to risk the reliability of your package. The total effort needed to complete a program can increase dramatically as the last few percent in absolute performance gets squeezed out.

## 5.12 Identifying errors

Section 5.7 was concerned with spotting the presence of errors. Here I want to talk about working out which part of your code was responsible for them. The sections are kept separate to help you to recognise this, and hence to allow you to separate noticing incorrect behaviour from spotting mistakes in your code. Of course if, while browsing code, you find a mistake you can work on from it to see if it can ever cause the program to yield wrong results, and this study of code is one valid error-hunting activity. But even in quite proper programs it is possible to have errors that never cause the program to misbehave in any way that can be noticed. For instance the mistake might just have a small effect on the performance of some not too important subroutine, or it may be an illogicality that could only

be triggered into causing real trouble by cases that some earlier line of code had filtered out.

You should also recognise that some visible bugs are not so much due to any single clear-cut error in a program but to an interaction between several parts of your code each of which is individually reasonable but which in combination fail. Most truly serious disasters caused by software failure arise because of complicated interactions between multiple “improbable” circumstances.

The first thing to try to locate the cause of an error is to start from the original test case that revealed it and to try to refine that down to give a minimal clear-cut demonstration of the bad behaviour. If this ends up small enough it may then be easy to trace through and work out what happened.

Pure thought and contemplation of your source code is then needed. Decide what Sherlock Holmes would have made of it! Run your compilers in whatever mode causes them to give as many warning messages as they are capable of, and see if any of those give valuable clues. Check out the `assert` facility and place copious assertions in your program that verify that all the high level expectations you have are satisfied.

If this fails the next thought is to arrange to get a view on the execution of your code as it makes its mistake. Even when clever language-specific debuggers are available it will often be either necessary or easiest to do this by extra print statements into your code so it can display a trace of its actions. There is a great delicacy here. The trace needs to be detailed enough to allow you to spot the first line in it where trouble has arisen, but concise enough to be manageable. My belief is that one should try to judge things so that the trace output from a failing test run is about two pages long.

There are those who believe that programs will end up with the best reliability if they start off written in as fragile way as possible. Code should always make as precise a test as possible, and should frequently include extra cross checks which, if failed, cause it to give up. The argument is that this way a larger number of latent faults will emerge in early testing, and the embedded assertions can point the programmer directly to the place where an expectation failed to be satisfied, which is at least a place to start working backwards from in a hunt for the actual bug.

When debugging, When you have eliminated the impossible, whatever remains, however improbable, must be the truth.

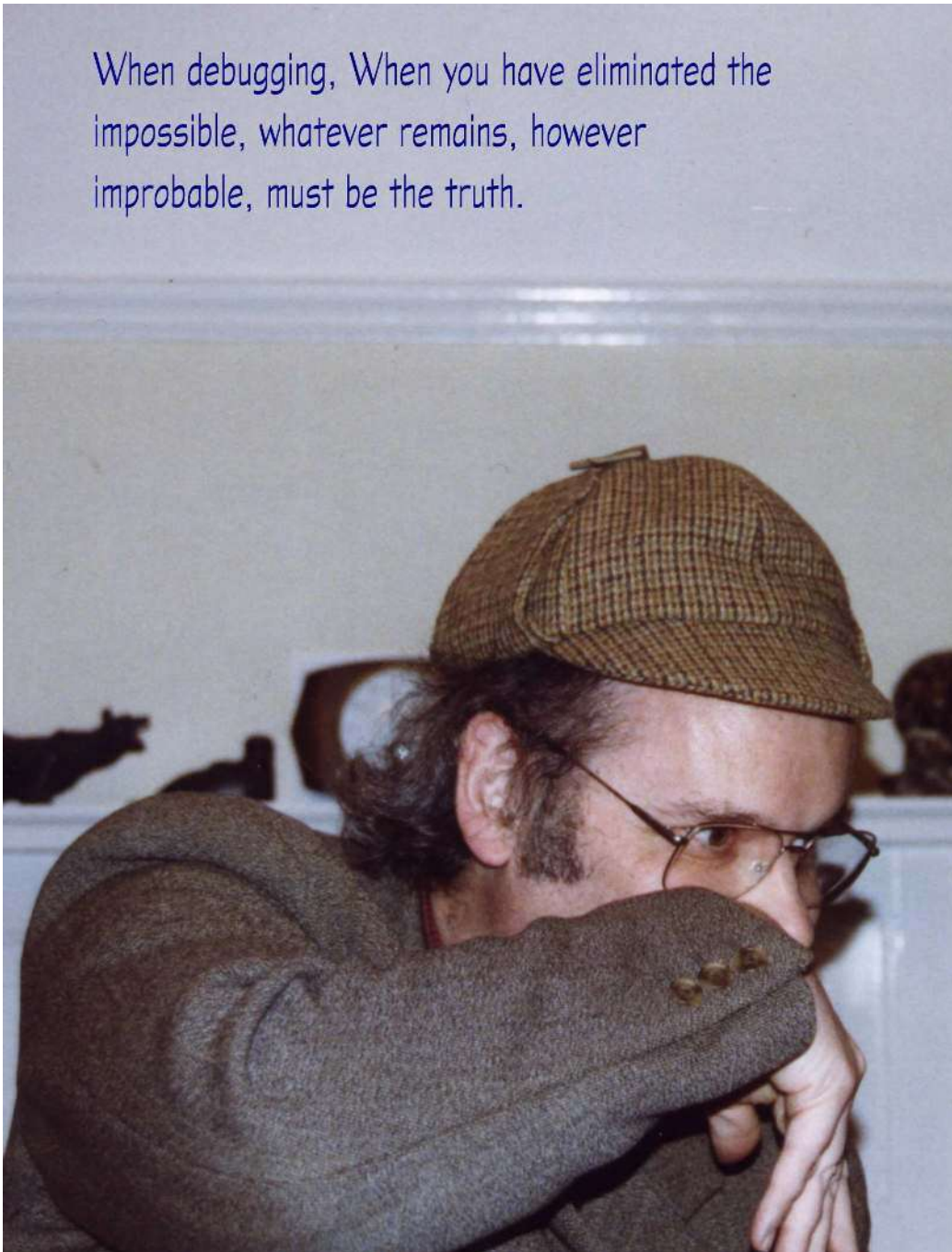


Figure 5.6: Effective debugging calls for great skill.

With many sorts of bugs it can be possible to home in on the difficulty by some sort of bisection search. Each test run should be designed to halve the range of code within which the error has been isolated.

Some horrible problems seem to vanish as soon as you enable any debugging features in your code or as soon as you insert extra print statements into it. These can be amazingly frustrating! They may represent your use of an unsafe language and code that writes beyond the limit of an array, or they could involve reliance on the unpredictable value of an un-initialised variable. Sometimes such problems turn out to be bugs in the compiler you are using, not in your own code. I believe that I have encountered trouble of some sort (often fairly minor, but trouble nevertheless) with every C compiler I have ever used, and I have absolute confidence that no other language has attained perfection in this regard. So sometimes trying your code on a different computer or with a different compiler will either give you a new diagnostic that provides the vital clue, or will behave differently thereby giving scope for debugging-by-comparison.

Getting into a panic and trying random changes to your code has no proper part to play either in locating or identifying bugs.

### 5.13 Corrections and other changes

With a number of bugs spotted and isolated the time comes to extirpate them. The ideal should be that when a bug is removed it should be removed totally and it should never ever be able to come back. Furthermore its friends and offspring should be given the same treatment at the same time, and of course no new mistakes should be allowed to creep in while the changes are being made. This last is often taken for granted, but when concentrating on one particular bug it is all too easy to lose sight of the overall pattern of code and even introduce more new bugs than were being fixed in the first case. Regression testing is at least one line of defence that one should have against this, but just taking the correction slowly and thinking through all its consequences what is mostly wanted. Small bugs (in the sense discussed earlier) that are purely local in scope give fewest problems. However sometimes testing reveals a chain of difficulties that must eventually be recognised as a sign that the initial broad design of the program had been incorrect, and that the proper correction strategy does not involve fixing the problems one at a time but calls for an almost fresh start on the whole project. I think that would be the proper policy for the program in section 5.18, and that is part of why the exercise there asks you to identify bugs but not to correct them.

Upgrading a program to add new features is at least as dangerous as correcting bugs, but in general any program that lasts for more than a year or so will end up with a whole raft of alterations having been made to it. These can very easily

damage its structure and overall integrity, and the effect can be thought of as a form of *software rot* that causes old code to decay. Of course software rot would not arise if a program never needed correcting and never needed upgrading, but in that case the program was almost certainly not being used and was fossilised rather than rotting. Note that for elderly programs the person who makes corrections is never the original program author (even if they have the same name and birthday, the passage of time has rendered them different). This greatly increases the prospect of a would-be correction causing damage.

All but the most frivolous code should be kept under the control of some source management tool (perhaps `rcs`) that can provide an audit trail so that changes can be tracked. In some cases a discussion of a bug that has now been removed might properly remain as a comment in the main source code, but much more often a description of what was found to be wrong and what was changed to mend it belongs in a separate project log. After all if the bug really has been removed who has any interest in being reminded of the mistake that it represented?

Whenever a change is made to a program, be it a bug-fix or an upgrade, there is a chance that some re-work will be needed in documentation, help files, sample logs and of course the comments. Once again the idea of literate programming comes to the fore in suggestion that all these can be kept together.

## 5.14 Portability of software

Most high level languages make enthusiastic claims that programs written in them will be portable from one brand of computer to another, just as most make claims that their compilers are “highly optimising”. Java makes especially strong claims on this front, and its owners try rather hard to prevent anybody from diverging from a rigid standard. However even in this case there are differences between Java 1.0 and 1.1 (and no doubt 1.2) that may cause trouble to the unwary.

In reality achieving portability for even medium sized programs is not as easy as all that. To give a gross example of a problem not addressed at all by programming language or standard library design, a Macintosh comes as standard with a mouse with a single button, while most Unix X-windows systems have three-button mice. In one sense the difference is a frivolity, but at another it invites a quite substantial re-think of user interface design. At the user interface level a design that makes good use of a screen with 640 by 480 pixels and 16 or 256 colours (as may be the best available on many slightly elderly computers) may look silly on a system with very much higher resolution and more colours.

For most programming languages you will find that implementations provided by different vendors do not quite match. Even with the most standardised languages hardly any compiler supplier will manage to hold back from providing

some private extra goodies that help distinguish them from their competitors. Such extras will often be things that it is very tempting to make use of. Around 1997-8 a good example of such a feature is “Active-X” which Microsoft is promoting. To use such a feature tends to lock you to one vendor or platform, while to ignore it means that you can not benefit from the advantages that it brings. By now you will know what my suggested response to conflicts like this will be. Yes, it is to make your decisions explicitly and consciously rather than by default, to make them in view of stated ideas about what the users of your code will need, and to include all the arguments you use to support your decision in your design portfolio.

There are frequently clever but non-portable tricks that can lead to big performance gains in code but at cost in portability. Sometimes the proper response to these is to have two versions of the program, one slow but very portable and the other that takes full advantage of every trick available on some platform that is especially important to you.

## 5.15 Team-work

Almost all of this course is about programming in the small, with a concentration on the challenges facing a lone programmer. It is still useful to think for a while how to handle the transition from this state into a large-team corporate mentality. One of the big emotional challenges in joining a team relates to the extent to which you end up “owning” the code you work on. It is very easy to get into a state where you believe (perhaps realistically) that you are the only person who can properly do anything to the code you write. It is also easy to become rather defensive about your own work. A useful bit of jargon that refers to breaking out of these thought patterns is *ego-free programming*. In this ideal you step back and consider the whole project as the thing you are contributing to, not just the part that you are visibly involved in implementing. It may also be useful to recognise that code will end up with higher quality if understanding of it is shared between several people, and that bugs can be viewed as things to be found and overcome and never as personal flaws in the individual who happened to write that fragment of code.

When trying to design code or find a difficult bug it can be very valuable to explain your thoughts to somebody else. It may be that they need not say much more than er and um, and maybe they hardly need to listen (but you probably need to believe that they are). By agreeing that you will listen to their problems at a later stage this may be a habit you can start right now with one or a group of your contemporaries.

Reading other people’s code (with their permission, of course) and letting them read yours can also help you settle on a style or idiom that works well for

you. It can also help get across the merits of code that is well laid out and where the comments are actually helpful to the reader.

If you get into a real group programming context, it may make sense to consider partitioning the work in terms of function, for instance system architect, programmer, test case collector, documentation expert, . . . rather than trying to distribute the management effort and split the programming into lots of little modules, but before you do anything too rash read some more books on software engineering so that once again you can make decisions in an informed and considered way.

## 5.16 Lessons learned

One of the oft-repeated observations about the demons of large-scale software construction is that *there is no silver bullet*. In other words we can not expect to find a single simple method that, as if by magic, washes away all our difficulties. This situation also applies for tasks that are to be carried out by an individual programmer or a very small team. No single method gives a key that makes it possible to sit down and write perfect programs without effort. The closest I can come to an idea for something that is generally valuable is experience – experience on a wide range of programming projects in several different languages and with various different styles of project. This can allow you to spot features of a new task that have some commonalty with one seen before. This is, however, obviously no quick fix. The suggestions I have been putting forward here are to try to make your analysis of what you are trying to achieve as explicit in your mind as possible. The various sections in these notes provide headings that may help you organise your thoughts, and in general I have tried to cover topics in an order that might make sense in real applications. Of course all the details and conclusions will be specific to your problem, and nothing I can possibly say here can show you how to track down your own very particular bug or confusion! I have to fall back on generalities. Keep thinking rather than trying random changes to your code. Try to work one step at a time. Accept that errors are a part of the human condition, and however careful you are your code will end up with them.

But always remember the two main slogans:

**Programming is easy**

and

**Programming is fun.**

## 5.17 Final Words

Do I follow my own advice? Hmm I might have known you would ask that! Well most of what I have written about here is what I try to do, but I am not especially formal about any of it. I only really go overboard about design and making documentation precede implementation when starting some code that I expect to give me special difficulty. I have never got into the swing of literate programming, and suspect that I like the idea more than the reality. And I sometimes spend many more hours on a stretch at a keyboard than I maybe ought to. If this course and these notes help you think about the process of programming and allow you to make more conscious decisions about the style you will adopt then I guess I should be content. And if there is one very short way I would like to encapsulate the entire course, it would be the recommendation that you make all the decisions and thoughts you have about programming as open and explicit as possible.

Good luck!

## 5.18 Challenging exercises

Some of you may already consider yourselves to be seasoned programmers able to cope with even quite large and complicated tasks. In which case I do not you to feel this course is irrelevant, and so I provide here at the end of the notes some programming problems which I believe are hard enough to represent real challenges, even though the code that eventually has to be written will not be especially long. There is absolutely no expectation that anybody will actually complete any of these tasks, or even find good starting points. However these examples may help give you concrete cases to try out the analysis and design ideas I have discussed: identifying the key difficulties and working out how to partition the problems into manageable chunks. In some cases the hardest part of a proper plan would be the design of a good enough testing strategy. The tasks described here are all both reasonably compact and fairly precisely specified. I have fought most of these myself and found that producing solutions that were neat and convincing as well as correct involved thought as well as more coding skill. There are no prizes and no ticks, marks or other bean-counter's credit associated with attempting these tasks, but I would be jolly interested to see what any of you can come up with, provided it can be kept down to no more than around 4 sides of paper.



## MULDIV

The requirement here is to produce a piece of code that accepts four integers and computes  $(a * b + c) / d$  and also the remainder from the division. It should be assumed that the computer on which this code is to be run has 32-bit integers, and that integer arithmetic including shift and bitwise mask operations are available, but the difficulty in this exercise arises because  $a * b$  will be up to 64-bits long and so it can not be computed directly. “Solutions” that use (eg) the direct 64-bit integer capabilities of a DEC Alpha workstation are not of interest!

It should be fairly simple to implement `muldiv` if efficiency were not an issue. To be specific this would amount to writing parts of a package that did double-length integer arithmetic. Here the additional expectation is that speed does matter, and so the best solution here will be one that makes the most effective possible use of the 32-bit arithmetic that is available. Note also that code of this sort can unpleasantly easily harbour bugs, for instance due to some integer overflow of an intermediate result, that only show up in very rare circumstances, and that the pressure to achieve the best possible performance pushes towards code that comes very close to the limits of the underlying 32-bit arithmetic. Thought will be needed when some or all of the input values are negative. The desired behaviour is one where the calculated quotient was rounded towards zero, whatever its sign.

## Overlapping Triangles

A point in the  $X$ - $Y$  plane can be specified by giving its co-ordinates  $(x,y)$ . A triangle can then be defined by giving three points. Given two triangles a number of possibilities arise: they may not overlap at all or they may meet in a point or a line segment, or they may overlap so that the area where they overlap forms a triangle, a quadrilateral, a pentagon or a hexagon. Write code that discovers which of these cases arises, returning co-ordinates that describe the overlap (if any).

A point to note here is that any naive attempt to calculate the point where two lines intersect can lead to attempts to divide by zero if the lines are parallel. Near-parallel lines can lead to division by very small numbers, possibly leading to subsequent numeric overflow. Such arithmetic oddities must not be allowed to arise in the calculations performed.

## Matrix transposition

One way of representing an  $m$  by  $n$  matrix in a computer is to have a single vector of length  $mn$  and place the array element  $a_{i,j}$  at offset  $mi + j$  in the vector. Another would be to store the same element at offset  $i + nj$ . One of these representation

means that items in the same row of the matrix live close together, the other that items in the same column are adjacent.

In some calculations it can make a significant difference to speed which of these layouts is used. This is especially true for computers with virtual memory. Sometimes one part of a calculation would call for one layout, and a later part would prefer the other.

The task here is therefore to take integers  $m$  and  $n$  and a vector of length  $mn$ , and rearrange the values stored in the vector so that if they start off in one of as one representation of a matrix they end up as the other. Because the matrix should be assumed to be quite large you are not allowed to use any significant amount of temporary workspace (you can not just allocate a fresh vector of length  $mn$  and copy the data into it in the new order — you may assume you may use extra space of around  $m + n$  if that helps, but not dramatically more than that).

If the above explanation of the problem<sup>8</sup> feels out of touch with today's computer uses, note how the task relates to taking an  $m$  by  $n$  matrix representing a picture and shuffling the entries to get the effect of rotating the image by 90 degrees. Just that in the image processing case you may be working with data arranged in sub-word-sized bite-fields, say at 4 bits per pixel.

## Sprouts

The following is a description of a game<sup>9</sup> to be played by two players using a piece of paper. The job of the project here is to read in a description of a position in the game and make a list of all the moves available to the next player. This would clearly be needed as part of any program that played the game against human opposition, but the work needed here does not have to consider any issues concerning the evaluation of positions or the identification of good moves.

The game starts with some number of marks made on a piece of paper, each mark in the form of a capital 'Y'. Observe that each junction has exactly three little edges jutting from it. A move is made by a player identifying two free edges and drawing a line between them. The line can snake around things that have been drawn before as much as the player making the move likes, but it must not cross any line that was drawn earlier. The player finishes the move by drawing a dot somewhere along the new line and putting the stub of a new edge jutting out from it in one of the two possible directions. Or put a different but equivalent way, the player draws a new 'Y' and joins two of its legs up to existing stubs with lines that do not cross any existing lines. The players make moves alternately and the first player unable to make a further legal move will be the loser.

---

<sup>8</sup>This is an almost standard classical problem and if you dig far enough back in the literature you will find explanations of a solution. If you thought to do that for yourself, well done!

<sup>9</sup>Due to John Conway

A variation on the game has the initial state of the game just dots (not ‘Y’ shapes) and has each player draw a new dot on each edge they create, but still demands that no more than three edges radiate from each dot. The difference is that in one case a player can decide which side of a new line any future line must emerge from. I would be equally happy whichever version of the game was addressed by a program, provided the accompanying documentation makes it clear which has been implemented!

The challenge here clearly largely revolves around finding a way to describe the figures that get drawn. If you want to try sprouts out as a game between people before automating it, I suggest you start with five or six starting points.

### **ML development environment**

The task here is not to write a program, but just to sketch out the specification of one. Note clearly that an implementation of the task asked about here would be quite a lot of work and I do not want to provide any encouragement to you to attempt all that!

In the Michaelmas Term you were introduced to the language ML, and invited to prepare and test various pieces of test code using a version running under Microsoft Windows. You probably used the regular Windows “notepad” as a little editor so you could change your code and then paste back corrected versions of it into the ML window. Recall that once you have defined a function or value in ML that definition remains fixed for ever, and so if it is incorrect you probably need to re-type not only it but everything you entered after it. All in all the ML environment you used was pretty crude (although I am proud of the greek letters in the output it generates), and it would become intolerable for use in medium or large-scale projects. Design a better environment, and append to your description of it a commentary about which aspects of it represent just a generically nice programmer’s work-bench and which are motivated by the special properties of ML.

### **An example from the literature**

The following specification is given as a paragraph of reasonably readable English text, and there is then an associated program written in Java. This quite small chunk of code can give you experience of bug-hunting: please do not look up the original article in CACM<sup>10</sup> until you have spent some while working through the code checking how it works and finding some of the mistakes. In previous years when I have presented this material to our students they did almost as well as the professional programmers used in the original IBM study, but they still found only

---

<sup>10</sup>Communications of the ACM, vol 21, no 9, 1978.

a pathetically small proportion of the total number of known bugs! I am aware that the Java transcription of this program has changed its behaviour from the original PL/I version and the intermediate C one. I do not believe that the changes are a case of bugs evaporating in the face of Java, but some may be transcription errors I have made. But since the object of this exercise is that you locate bugs, whatever their source, this does not worry me much, and I present the example as it now stands, oddities and all.



Figure 5.7: Picture courtesy Shamila Corless.

Formatting program for text input. Converted from the original PL/I version which is in a paper by Glen Myers, CACM vol 21 no 9, 1978

- (a) This program compiles correctly: it is believed not to contain either syntax errors or abuses of the Java library.
- (b) A specification is given below. You are to imagine that the code appended was produced by somebody who had been provided with the specification and asked to produce an implementation of the utility as described. But they are not a very good programmer!
- (c) Your task is one of quality control - it is to check that the code as given is in agreement with the specification.  
If any bugs or mis-features are discovered they should be documented but it will be up to the original programmer to correct them. If there are bugs it is desirable that they all be found.
- (d) For the purposes of this study, a bug or a mis-feature is some bad aspect of the code that could be visible to users of the binary version of the code. Ugly or inefficient code is deemed not to matter, but even small deviations from the letter of the specification and the things sensibly implicit in it do need detecting.
- (e) Let me repeat point (a) again just to stress it - the code here has had its syntax carefully checked and uses the Java language and library in a legal straightforward way, so searching for bugs by checking fine details of the Java language specification is not expected to be productive. I have put in comments to gloss use of library functions to help those who do not have them all at their finger-tips. The code may be clumsy in places but I do not mind that!  
I have tried to keep layout of the code neat and consistent. There are few comments "because the original programmer who wrote the code delivered it in that state".

```

/*****
 * Specification
 * =====
 *
 * Given an input text consisting of words separated by
 * blanks or new-line characters, the program formats it
 * into a line-by-line form such that (1) each output
 * line has a maximum of 30 characters, (2) a word in
 * the input text is placed on a single output line, and
 * (3) each output line is filled with as many words as
 * possible.
 *
 * The input text is a stream of characters, where the
 * characters are categorised as break or nonbreak
 * characters. A break character is a blank, a new-line
 * character (&), or an end of text character (/).
 * New-line characters have no special significance;
 * they are treated as blanks by the program. & and /
 * should not appear in the output.
 *
 * A word is defined as a nonempty sequence of non-break
 * characters. A break is a sequence of one or more
 * break characters. A break in the input is reduced to
 * a single blank or start of new line in the output.
 *
 * The input text is a single line entered from a
 * simple terminal with an fixed 80 character screen
 * width. When the program is invoked it waits for the
 * user to provide input. The user types the input line,
 * followed by a / (end of text) and a carriage return.
 * The program then formats the text and displays it on
 * the terminal.
 *
 * If the input text contains a word that is too long to
 * fit on a single output line, an error message is
 * typed and the program terminates. If the end-of-text
 * character is missing, an error message is issued and
 * the user is given a chance to type in a corrected
 * version of the input line.
 *
 * (end of specification)
 *****/

```

```
import java.io.*;

public class Buggy
{
    final static int LINESIZE = 31;

    public static void main(String [] args)
    {
        int k,
            bufpos,
            fill,
            maxpos = LINESIZE;
        char cw,
            blank = ' ',
            linefeed = '$',
            eotext = '/';
        boolean moreinput = true;
        char [] buffer = new char [LINESIZE];
        bufpos = 0;
        fill = 0;
        while (moreinput)
        {    cw = gchar();
            if (cw == blank || cw == linefeed || cw == eotext)
            {
                if (cw == eotext) moreinput = false;
                if ((fill + 1 + bufpos) <= maxpos)
                {    pchar(blank);
                    fill = fill + 1;
                }
                else
                {    pchar(linefeed);
                    fill = 0;
                }
                for (k = 0; k < bufpos; k++) pchar(buffer[k]);
                fill = fill + bufpos;
                bufpos = 0;
            }
            else if (bufpos == maxpos)
            {    moreinput = false;
                System.out.println("Word to long");
            }
        }
    }
}
```

```

        else
        {   bufpos = bufpos + 1;
            buffer[bufpos-1] = cw;
        }
    }
    pchar(linefeed);
    return;
}

// I use B as a shorthand for the character ' '.
final static char B = ' ';

final static int ILENGTH = 80;

// Make suitable array with initial contents Z
// then a load of blanks.
static char [] buffer = {
    'Z',B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B};

// bcount is defined here so that it keeps its values
// across several calls to gchar().
static int bcount = 1;

static char gchar()
{
    char [] inbuf = new char [ILENGTH];
    char eotext = '/';
    char c;
    if (buffer[0] == 'Z')
    {   getrecord(inbuf);

```



```

// indexOf returns the index of a position where the given
// character is present in a string, or -1 if it is not
// found.
    if (new String(inbuf).indexOf((int) eotext) == -1)
    {   System.out.println("No end of text mark");
        buffer[1] = eotext;
    }
    else for (int j=0; j<ILENGTH; j++)
        buffer[j] = inbuf[j];
}
c = buffer[bcount-1];
bcount = bcount + 1;
return c;
}

// a static output buffer, again blank-filled.
static char [] outline =
{ B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
  B,B,B,B,B,B,B,B,B,B,B,B
};

// i indicates which place in outline pchar should
// put the next character at.
static int i = 1;

static void pchar(char c)
{
    int linefeed = '$';
    if (c == linefeed)
    {   System.out.println(outline);
        for (int j=0; j<LINESIZE; j++)
            outline[j] = B;
        i = 1;
    }
    else
    {   outline[i-1] = c;
        i = i + 1;
    }
}
}

```

```
// Get access to keyboard input. No tricks here!
static BufferedReader in =
    new BufferedReader(
        new InputStreamReader(System.in),
        1);

static void getrecord(char [] b)
{
    String s;
    try
    {    s = in.readLine();
    }
    catch (IOException e)
    {    s = " ";
    }
    for (int i = 0; i < ILENGTH; i++)
    {    if (i < s.length()) b[i] = s.charAt(i);
        else b[i] = ' ';
    }
}

}

/* End of file */
```

# Chapter 6

## A representative application

The purpose of this chapter is not to form part of the official examinable course, but to provide you with some extended samples of Java so that you can see the various facilities working together and so you can consider how the code ought to have been written to make it as clear and robust as possible. Note that I do not guarantee that my code is a paragon of clarity, and although the example here is much larger than the ones that have gone before it has still been trimmed fairly close to the bone to make it as small and perhaps comprehensible as I could manage.

### 6.1 A Lisp interpreter

This final example for these notes is as large and complicated as any of the others here — but most of the Java features it uses are ones that have been seen before. It is an implementation of a very much cut-down version of the programming language Lisp. In this language, which is one of the oldest programming languages that is still in use today, and a version of which is used to customise the `emacs` editor, all syntax is indicated with explicit parentheses. The programs that one writes are very much like ML ones, except that Lisp does not have either the syntax and operators of ML nor the type-checking. While I do not want to divert this Java course into one on Lisp I will give a brief example of the sort of program that can be used to test my Minilisp. It defines a function to reverse lists and demonstrates its use.

```
(defun reverse (x)
  (rev1 x nil))

(defun rev1 (a b)
  (cond
```

```
((eq a nil) b)
(t (rev1 (cdr a) (cons (car a) b))))

(reverse '(a b c d))
```

To find out about Lisp in its modern and very large form you could check out *Common Lisp — the Language*[21] however the Minilisp here perhaps makes better sense with reference to the *Lisp 1.5 manual*[13], which is amazingly ancient now but which has the huge advantage of brevity. Of course the Common Lisp manual was written by Guy Steele who participated in writing the Java language reference — so as well as Lisp being a very direct ancestor of ML it can also be seen as having had noticable input into Java<sup>1</sup>. Another reason for including an implementation of it here.

I have in fact extended the tiny Lisp shown here into a full-scale one that is capable of running programs that are many tens of thousands of lines long. The write-up of that work is in “Further evaluation of Java for Symbolic Calculation”, Proc. ISSAC 00, St Andrews, Scotland, August 2000.

---

<sup>1</sup>The “Flavours” package developed for Lisp at MIT was one of the earlier programming systems that encouraged object oriented design, supported inheritance and worked to make large-scale programming practical. CLOS (Common Lisp Object System) is the major internationally standardised model for dynamic object-oriented programming.

Now for the implementation. Making sense of it is liable to be a substantial struggle for most of you, but I hope that the fact that I can fit an implementation of a programming language into these notes is at least interesting!

```
// Minilisp
//
// Basic and utterly tiny Lisp system coded
// in Java by Arthur Norman, 1998.
//
// In spirit much like an older BCPL then C
// version of the same thing!
//
// Supports
//   quote, cond, defun
//   atom, eq, car, cdr, cons
// has fragments of code waiting to be extended
// to do rather more.

import java.io.*;

// Lisp has a single inclusive data-type, which I call
// LispObject here. It has sub-types that are symbols,
// numbers, strings and lists. Here I give just two
// methods (print and eval) that may be used on anything.

abstract class LispObject
{
    public abstract void print();
    public abstract LispObject eval(Environment env);
}

// A "cons" is an ordered pair. In ML terms it would be
// a bit like ('a * 'b)

class Cons extends LispObject
{
    // The left and right parts of a pair are called
    //           CAR and CDR
    public LispObject car, cdr;
    Cons(LispObject car, LispObject cdr)
    { this.car = car; this.cdr = cdr; }
}
```

```

// Function calls are written as lists (fn a1 a2 ...)
public LispObject eval(Environment env)
{   int n = 0;
    for (LispObject a=cdr;
        a instanceof Cons;
        a = ((Cons)a).cdr) n++;
    LispObject [] args = new LispObject [n];
    n = 0;
    for (LispObject a=cdr;
        a instanceof Cons;
        a = ((Cons)a).cdr) args[n++] = ((Cons)a).car;
// Now I have unpicked the actual arguments into a vector
    if (car instanceof Symbol)
        {   Symbol f = (Symbol)car;
// "special" functions are for QUOTE, CONS and DEFUN. They
// do not evaluate their arguments
            if (f.special != null)
                return f.special.op(args, env);
// All other functions have their arguments evaluated.
            for (int i=0; i<n; i++)
                args[i] = args[i].eval(env);
// Call the function!
            return f.fn.op(args);
        }
// return NIL if I do not otherwise know what to do
    else return Minilisp.nil;
}

// Lists print as (a b c ...)
// and if a list ends in NIL then it is displayed with
// just a ")" at the end, otherwise the final atom is
// shown after a "."
public void print()
{   LispObject x = this;
    String delim = "(";
    while (x instanceof Cons)
    {   System.out.print(delim);
        delim = " ";
        ((Cons)x).car.print();
        x = ((Cons)x).cdr;
    }
}

```

```

        if (x != Minilisp.nil)
        {   System.out.print(" . ");
            x.print();
        }
        System.out.print(")");
    }
}

// I do not do a lot with strings here.
class LispString extends LispObject
{
    public String string;
    LispString(String s)
    { this.string = s; }
    public LispObject eval(Environment env)
    { return this; }
    public void print()
    {   System.out.print "\"" + string + "\"");
    }
}

class Symbol extends LispObject
{
    public String pname;      // print name
    public LispObject plist; // property list (unused)
    Symbol obListNext;       // chaining of symbols
    public LispFunction fn;  // function (if any)
    public SpecialFunction special; // special fn (if any)
// intern() looks up a Java String and find the Lisp
// symbol with that name. It creates it if needbe.
    public static Symbol intern(String name,
        LispFunction fn, SpecialFunction special)
    {   Symbol p;
        for (p=Minilisp.obList; p!=null; p=p.obListNext)
        {   if (p.pname.equals(name)) return p;
        }
    }
}

```

```

// not found on "object-list" (oblist), so create it
    p = new Symbol();
    p.pname = name;
    p.plist = Minilisp.nil;
    p.obListNext = Minilisp.obList;
    Minilisp.obList = p;
    p.fn = fn != null ? fn : new Undefined(name);
    p.special = special;
    return p;
}

// The symbols NIL and T are special - they evaluate
// to themselves. All others get looked up in an
// environment that stores current values of local vars.
    public LispObject eval(Environment env)
    {   if (this == Minilisp.nil ||
        this == Minilisp.lispTrue) return this;
        return env.eval(this);
    }
    public void print()
    {   System.out.print(pname);
    }
}

// An environment is a chain of Bindings terminated with
// a NullEnvironment. Each binding holds information of
// the form
//     variable = value

abstract class Environment
{
    public abstract LispObject eval(Symbol name);
}

class NullEnvironment extends Environment
{
    public LispObject eval(Symbol name)
    {   System.out.println("Undefined variable: " +
        name.pname);
        System.exit(1);
        return null;
    }
}

```



```
class Binding extends Environment
{
    public Symbol name;
    public LispObject value;
    public Environment next;
    Binding(Symbol name, LispObject val, Environment next)
    {    this.name = name;
        this.value = val;
        this.next = next;
    }
    public LispObject eval(Symbol x)
    {    if (x == name) return value;
        else return next.eval(x);
    }
}

// I do not do a lot with numbers here!
class LispNumber extends LispObject
{
    public int value;
    LispNumber(int value)
    {    this.value = value; }
    public LispObject eval(Environment env)
    {    return this; }
    public void print()
    {    System.out.print(value);
    }
}

// Each built-in function is created wrapped in a class
// that is derived from LispFunction.

abstract class LispFunction
{
    public abstract LispObject op(LispObject [] args);
}
```

```

class Undefined extends LispFunction
{
    String name;
    Undefined(String name)
    { this.name = name; }
    public LispObject op(LispObject [] args)
    {   System.out.println("Undefined function " + name);
        System.exit(1); // throw?
        return null;
    }
}

// If a symbol has an interpreted definition its
// associated function is this job, which knows how to
// extract the saved definition and activate it.

class Interpreted extends LispFunction
{
    LispObject a, b;
    Environment env;
    Interpreted(LispObject a,    // formal args
                LispObject b,    // body
                Environment env) // environment
    {   this.a = a;
        this.b = b;
        this.env = env;
    }
    public LispObject op(LispObject [] args)
    {   LispObject a1 = a;
        int i = 0;
        Environment e = env;
        while (a1 instanceof Cons)
        {   e = new Binding(
            (Symbol)((Cons)a1).car, args[i++], e);
            a1 = ((Cons)a1).cdr;
        }
        return b.eval(e);
    }
}

```

```
// Similar stuff, but for "special functions"

abstract class SpecialFunction
{
    public abstract LispObject op(LispObject [] args,
                                  Environment env);
}

// (quote xx) evaluates to just xx

class QuoteSpecial extends SpecialFunction
{
    public LispObject op(LispObject [] args,
                        Environment env)
    {
        return args[0];
    }
}

// (cond (p1 e1)          if p1 then e1
//       (p2 e2)          else if p2 then e2
//       (p3 e3) )        else if p3 then e3
//                       else nil

class CondSpecial extends SpecialFunction
{
    public LispObject op(LispObject [] args,
                        Environment env)
    {
        for (int i=0; i<args.length; i++)
        {
            Cons x = (Cons)args[i];
            LispObject predicate = x.car;
            LispObject consequent = ((Cons)x.cdr).car;
            if (predicate.eval(env) != Minilisp.nil)
                return consequent.eval(env);
        }
        return Minilisp.nil;
    }
}
```

```
// (defun name (a1 a2 a3) body-of-function)

class DefunSpecial extends SpecialFunction
{
    public LispObject op(LispObject [] args,
                        Environment env)
    {
        Symbol name = (Symbol)args[0];
        LispObject vars = args[1];
        LispObject body = args[2];
        name.fn = new Interpreted(vars, body, env);
        return name;
    }
}

// like ML "fun car (a :: b) = a;"

class CarFn extends LispFunction
{
    public LispObject op(LispObject [] args)
    {
        return ((Cons)(args[0])).car;
    }
}

// like ML "fun cdr (a :: b) = b;"

class CdrFn extends LispFunction
{
    public LispObject op(LispObject [] args)
    {
        return ((Cons)(args[0])).cdr;
    }
}

// like ML "fun atom (a :: b) = false | atom x = true;"

class AtomFn extends LispFunction
{
    public LispObject op(LispObject [] args)
    {
        return args[0] instanceof Cons ? Minilisp.nil :
            Minilisp.lispTrue;;
    }
}
```

```
// (eq a b) is true if a and b are the same thing

class EqFn extends LispFunction
{
    public LispObject op(LispObject [] args)
    {    return args[0]==args[1] ? Minilisp.lispTrue :
        Minilisp.nil;
    }
}

// like ML    "fun cons a b = a :: b;"

class ConsFn extends LispFunction
{
    public LispObject op(LispObject [] args)
    {    return new Cons(args[0], args[1]);
    }
}

// (stop) exist from this Lisp.

class StopFn extends LispFunction
{
    public LispObject op(LispObject [] args)
    {    System.exit(0);
        return null;
    }
}

// The top-level class has a bunch of input
// and management code.

public class Minilisp
{

    public static Symbol nil, lispTrue,
        obList, lambda, cond, quote, defun;

    static StreamTokenizer input;
    static int inputType;
    static boolean inputValid;
```

```

static void initInput()
{
    input = // Get stream & establish syntax
           new StreamTokenizer(
               new BufferedReader(
                   new InputStreamReader(System.in),
                   1));
    input.eolIsSignificant(false);
    input.ordinaryChar('/');
    input.commentChar(';');
    input.ordinaryChar('\\');
    input.quoteChar('\\"');
    input.ordinaryChar('.'); // disable floating point
    input.lowerCaseMode(true);
    inputValid = false;
}

// read a single parenthesised expression.
// Supports 'xx as a short-hand for (quote xx)
// which is what most Lisps do.

// Formal syntax:
//   read => SYMBOL | NUMBER | STRING
//         => ' read
//         => ( tail
//   tail => )
//         => . read )
//         => read readtail

static LispObject read() throws IOException
{
    LispObject r;
    if (!inputValid)
    {
        inputType = input.nextToken();
        inputValid = true;
    }
    switch (inputType)
    {
case StreamTokenizer.TT_EOF:
        throw new IOException("End of file");
case StreamTokenizer.TT_WORD:
        r = Symbol.intern(input.sval, null, null);
        inputValid = false;
        return r;

```

```

case StreamTokenizer.TT_NUMBER:
    r = new LispNumber((int)input.nval);
    inputValid = false;
    return r;
case '\"': // String
    r = new LispString(input.sval);
    inputValid = false;
    return r;
case '\\':
    inputValid = false;
    r = read();
    return new Cons(quote, new Cons(r, nil));
case '(':
    inputValid = false;
    return readTail();
case ')':
case '.':
    inputValid = false;
    return nil;
default:
    r = Symbol.intern(
        String.valueOf((char)inputType), null, null);
    inputValid = false;
    return r;
}
}

static LispObject readTail() throws IOException
{
    LispObject r;
    if (!inputValid)
    {
        inputType = input.nextToken();
        inputValid = true;
    }
    switch (inputType)
    {
case '.':
        inputValid = false;
        r = read();
        if (!inputValid)
        {
            inputType = input.nextToken();
            inputValid = true;
        }
    }
}

```

```

        if (inputType == ')') inputValid = false;
        return r;
case StreamTokenizer.TT_EOF:
    throw new IOException("End of file");
case ')':
    inputValid = false;
    return nil;
default:r = read();
    return new Cons(r, readTail());
    }
}

// set up fixed definitions

static void initSymbols()
{
    obList = null;
    nil = Symbol.intern("nil", null, null);
    nil.plist = nil;
    Symbol.intern("car", new CarFn(), null);
    Symbol.intern("cdr", new CdrFn(), null);
    Symbol.intern("cons", new ConsFn(), null);
    Symbol.intern("atom", new AtomFn(), null);
    Symbol.intern("eq", new EqFn(), null);
    Symbol.intern("stop", new StopFn(), null);
    lispTrue = Symbol.intern("t", null, null);
// lambda is ready for extension of this code
    lambda = Symbol.intern("lambda", null, null);
    cond = Symbol.intern("cond", null,
        new CondSpecial());
    quote = Symbol.intern("quote", null,
        new QuoteSpecial());
    defun = Symbol.intern("defun", null,
        new DefunSpecial());
}

public static void main(String [] args)
{
    initInput();
    initSymbols();
    System.out.println("Arthur's Minilisp...");
}

```



```

    try
    {
// this is s READ-EVAL-PRINT loop
    for (int i=1;;i++)
        {   System.out.print(i + ": ");
// Ensure that the prompt gets displayed.
        System.out.flush();
        LispObject r = read();
        LispObject v = r.eval(new NullEnvironment());
        System.out.print("Value: ");
        v.print();
        System.out.println("");
        }
    }
    catch (IOException e)
    {   System.out.println("IO exception");
    }

    System.out.println("End of Lisp run. Thank you");

}

}

// End of Minilisp.java

```

### 6.1.1 Exercises

#### ML to Lisp

Find a Lisp 1.5 manual and/or study the Minilisp code, and then see how many ML list-processing functions you can convert into Lisp and run on the Java implementation. You have already seen `reverse`, so the next thing to try is `append`. It would probably be possible to coax the ML exercise on transitive closures through Minilisp!

#### Add a few more functions

Show that you have understood what is going on in the Minilisp code by adding in support for arithmetic, specifically functions to add, subtract, multiply and divide numbers, and to compare them for inequality and “less-than”.

## Emacs Lisp

Now you have at least minimal exposure to Lisp, investigate the way it is used in the `emacs` editor to allow users to create new language-specific editing modes, indentation and colour conventions.

## WeirdX

Visit the web-site <http://www.jcraft.com/weirdx/> and fetch yourself a copy of the source of the WeirdX X-windows server. It is around 25000 lines of Java! Do not fetch or examine in any way the related but commercial product called WiredX. Inspect the associated GNU public license carefully: it gives you permission to work on the source but imposes an obligation to make the source version of any adjustments available to the world at no cost. If you are happy with the GNU rules<sup>2</sup> investigate the behaviour of WeirdX and look for ways to

1. Identify and remove bugs;
2. Make it implement the latest X-windows specification more fully;
3. Enhance its performance;
4. Put a proper and copious number of comments into the code (!);
5. Ensure that it implements all possible facilities to reduce the security vulnerabilities that X-windows often opens up;
6. Make a careful comparison between the behaviour and capabilities of the improved WeirdX and other commercial and free X servers. Create a nicely structured wish-list for future enhancements.

As you make and test changes arrange to make your improvements available to the the world: see <http://sourceforge.net/projects/weirdx/>. In case you had not spotted, this is not a small exercise for an individual to attack lightly: it is a substantial challenge that calls for a lot of study way beyond core Java and if you decide to try it I suggest you form a small group to work in. Please let me know of any progress.

---

<sup>2</sup>For an extended discussion of the GNU public license and “free” software in general, see “The Cathedral and the Bazaar” by Eric Raymond[20]. Some of the explanation there makes very good sense, some seems to me to be silly!

# Chapter 7

## What you do NOT know yet

These days anybody arranging a lecture course is expected to think in terms of aims, objectives and learning outcomes. In particular it is deemed important to consider carefully what will be known by students who have taken the course. I want to take a contrary view: that what is most important to understand is what you will **not** know just because you have attended this course and done all the exercises. I view a recognition of ones limitations as vital both for any individual's personal integrity and as something that is essential if they are to be productive in any work environment. So let me start with:

### **After one course you are not a Java expert...**

To a large extent you only become an expert in any sort of programming after you have built up a substantial body of experience working on both individual and group projects. Most people can only gain the paranoia about bugs and documentation that is really needed via personal involvement in projects that fail horribly! Most people only gain a proper paranoid attitude to overall system design via personal involvement in projects that collapse through being inadequately specified, ill-planned or where project management is not strong enough. The Computer Science Tripos provides courses on Software Engineering that document examples of software disasters and explain the state of the art in avoiding trouble. Despite this most people view the horror stories as things that happen to "other people" until it is too late.

Java is an object oriented language. Proper use of it involves obtaining full leverage from the package, class and inheritance features it provides. This course has taken the attitude that Object Oriented Design is something that can not be fully appreciated until you are able to write a range of small programs comfortably. So once you have mastered this material it will be proper to re-start Java from the beginning concentrating on starting the design by planning a class hierarchy. This has to include careful thought about `private`, `protected` etc class

members, and it can involve formal schemes to document structures.

I have explained what packages are, but not discussed the practicalities of using lots of different packages for your own code. For small programs this is OK, but larger scale work will put more pressure on this side of things.

The Java libraries that support windows have been introduced in a very sketchy way here. That is because there is a huge amount to understand, and it would not even start to fit in the time available within the Part IA course. All the issues of getting pages laid out, controlling multiple windows, organising cut-and-paste operations and so on take a lot of learning. The most that can be hoped is that this course has given you a starting point to work from.

When a Java applet is run using a web browser it is subject to a variety of security limitations. This is so that when some remote user loads a web page that runs your Java code you can not then steal or destroy information on their computer. There is an elaborate scheme involving signatures and permissions that makes it possible for applets to be granted additional privilege, without giving them total access. Such issues will be important for large-scale Java projects.

Network access and concurrency are both areas where a programmer needs to absorb quite a lot of additional understanding before they can use Java (or any other language) in a fully satisfactory manner. It is easy to end up with systems that can either gum up in deadlock or have different threads create inconsistent results, and proper recovery after one thread fails or one network link times out is not at all easy to arrange. The operating systems threads later in the CST have much to say about the issues and pitfalls involved. The way in which Java can interface to databases may appear straightforward, but competent design of the database itself needs specialist understanding.

While Java is a pretty respectable general purpose language there are plenty of areas where simple use of pure Java will prove inadequate. Sometimes however it will still make sense to implement either the bulk of a program or perhaps just the user-interface in Java, with some other components in another language. This raises issues of inter-language working, and serious design challenges in the area of the inter-language interface.

During this course you have been encouraged to use `javac` as your Java compiler. For real projects it is almost certain that much more elaborate tools would be used. These would include ones to manage a chain of versions of code, tools to generate stylised code for some bits of a user interface, and a variety of debugging and performance analysis packages.

Actually hardly any programmer, however experienced, will be a full expert on all of the above matters. . .

**... but at least you have taken the first step!**

# Chapter 8

## Model Examination Questions

Some of these are ones I have invented, some have been submitted by members of the class, while yet more are adjustments of previous examination questions modified to fit into a Java context. They may not all be normalised for precision of wording or difficulty, but should give you something to try your skills on. I neither confirm nor deny the possibility that variants on some of these may appear in this or a future year's real examination paper. . .

Note also that for the Computer Science Part IA papers there can be full-sized (ie around 30 mins) questions, half size (around 15 mins) and tiny (about 1 minute) questions, and some of these fit or could readily be modified to fit several of these categories.

### 8.1 Java vs ML

Compare and contrast Java with ML under the following headings

1. Primitive data types;
2. Support for arrays;
3. Support for lists and trees;
4. Functions applicable to several different data-types;
5. Complexity of syntax.

[4 marks per section]

## 8.2 Matrix Class

Design a `Matrix` class for doing operations on  $n \times n$  matrices.

The class should include functions for performing matrix multiplication, addition, and multiplication by scalars (and all matrices may be supposed to have elements that are type `double`).

To what extent should other classes have rights to modify individual matrix elements? Justify your answer.

## 8.3 Hash Tables

Java allows you to declare arrays with any sort of content — for instance `int`, `double`, `String` or `Object`, but the value used to index the array is restricted to being an `int`. In some applications programmers want structures that behave like arrays whose subscripts are of type `String`. One way of doing this is to use a structure known as a *hash table*: Given an index value of type `String` an integer is computed using a *hash function*. The Java method `hashCode` in class `String` computes a suitable value. The `int` value computed depends only on the contents of the `String`, but different strings can be expected to lead to integers that are uniformly distributed across the whole range that integers cover. This hash value is reduced modulo the size of a fixed ordinary array, this (almost) lets the original `String` act as an index. The problem that arises is that it could be that two different `String` values hash onto the same location in the array.

This concern can be overcome by making the entries in the array linked lists of (*key*, *value*) pairs, so that the original index `String` will match one of the *key* strings and then the associated *value* is what is stored with it. The method `equals(String s)` in class `String` returns a `boolean` telling if one string is equal to another. Storing into a hash table will involve adding a new (*key*, *value*) pair to one of the lists.

Design appropriate data structures and classes for such a table in the case where the values to be stored are themselves of type `String`. Use the following method signatures in a class called `HashTable`:

```
void put(String key, String value) throws Duplicate;  
String get(String key) throws Missing;
```

where you may assume that the exceptions have already been defined elsewhere.

**Note:** The Java class-libraries provide a class called `HashMap` that does all this for you! The exercise here in coding it for yourself not something the ordinary Java programmer ever needs to do!

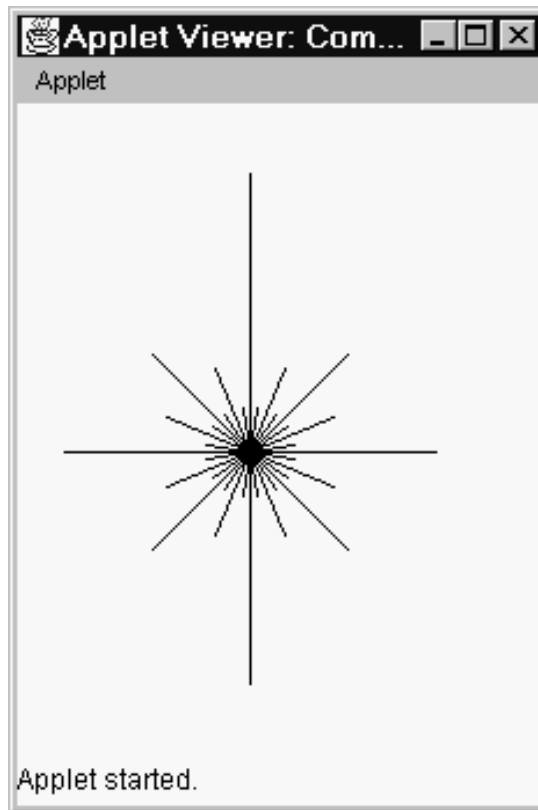
## 8.4 Compass Rose

A Java program to draw compass roses is needed. In the example shown the line pointing North is 6 units long, that pointing South is 5, East and West 4, NE, SE, SW and NW 3 and NNE, ENE and so on 2 and the rest 1 unit. In general if the line pointing North is  $n$  units long there will be  $2^{n-1}$  radial lines in all in the complete rose. You may assume that in your Java applet the method `paint` will be called when an image is to be displayed, and it gets passed an argument of type `Graphics`. The class `Graphics` has a method called `drawLine` that draws a line from one point to another given four integer arguments  $x_1, y_1, x_2$  and  $y_2$ .

Write two versions of the compass rose applet. The first should use recursion on the length  $n$  of a line, while the second should be iterative. In the second case it may be useful

to write an auxiliary method that calculates the highest power of two that divides exactly into a given number.

The length of the North line should be specified as a `final` variable in the class.



## 8.5 Language Words

In the context of Java, and given that this whole question is supposed to last 30 minutes, explain the following:

- package
- class
- import
- public

- `protected`
- `interface`
- `static`

## 8.6 Exception abuse

Show how Java exceptions can be used in conjunction with a `for` loop

1. to simulate the effect of a `break; statement;`
2. to simulate the effect of a `continue; statement.`

## 8.7 Queues

A priority queue maintains a list of pairs, each consisting of a priority (an integer) and a name (a `String`), sorted into increasing order of priority. Three operations are required — one that just creates an empty queue, and then one called `insert` that takes an integer and a string as its arguments and places them in the correct position in the list. Finally a method `next` takes no arguments. It removes the first pair from the list and returns its text string. It should throw an exception if the queue is empty when it is called.

Give the definition of a Java class that implements the queue object.

## 8.8 Loops

Describe the features of Java for controlling the repeated execution of a block of code.

Show how general uses of `for`, `while` and `do` could all be emulated using only loops that start of `while (true).`

## 8.9 Snap

Two identical packs of ordinary playing cards (52 different cards per pack) are shuffled and placed face downwards on a table. Two players then play a game of *Snap*. Each is allocated one pack, and in each turn in the game one card is turned up. The two upturned cards (one in front of each player) are compared. If the cards match a *snap-turn* is declared. A game ends when all 52 cards have been compared. Being computer scientists rather than five year olds these players



just record snap-turns and do not pick up or otherwise disturb the cards when one occurs!

Write a Java program which will simulate the game for the purposes of determining the probability of there being at least one snap-turn in a game. You may assume the existence of a random number generator but must state the properties of it that you rely on.

## 8.10 Partitions

Write a program in Java which, given two integer inputs  $j$  and  $k$  will output the combinations of  $k$  things partitions into  $k$  groups. For instance if  $j = 5$  and  $k = 3$  the output would be

```
(5, 0, 0)
(4, 1, 0)
(3, 2, 0)
(3, 1, 1)
(2, 2, 1)
```

## 8.11 Laziness

A Java class as follows has been defined

```
abstract class NextFunction
{
    public int next(int n);
}

class Lazy
{
    int head;
    Lazy tailOrNull;
    NextFunction next;
    public Lazy(int head, NextFunction next)
    {
        this.head = head;
        this.tailOrNull = null;
        this.next = next;
    }
    public int first() { return head; }
    public Lazy tail()
    {
        if (tailOrNull != null) return tailOrNull;
        tailOrNull = new Lazy(next(head), next);
    }
}
```

```

        return tailOrNull;
    }
}

```

The idea is to use this to represent lazy lists. In fact the small trick in the `tail` function that checks if the successor to a node has already been computed makes this a good representation. Derive a sub-class from `NextFunction` that overrides the `next` method with one that allows you to create a lazy list of integers  $(1, 2, 3, \dots)$ . Write code that, when given a lazy list, will print the first  $n$  integers in it.

Adapt the code so as to create a function that can accept a lazy list and generate a new lazy list from it that holds values which are the squares of the ones in the original list. Thus if passed my first lazy list as an argument this one would generate  $(1, 4, 9, \dots)$ .

## 8.12 Cryptarithmic

Write a Java program which can solve cryptarithmic puzzles in the format of the sum of two words. For example given the input

```

    SEND
  +MORE
  -----
  MONEY

```

the program would output

```

    9567
  +1085
  -----
  10652

```

NB Each letter has to represent a different digit.

## 8.13 Bandits

In Snoresville in small-town America, there are  $N$  bandits, and only one sheriff, Sheriff Dozy, who likes to do the minimum amount of work possible. He knows quite a lot about each of them, to the extent that he knows precisely which bandits each bandit knows. Soon, \$200,000 worth of gold will be deposited in Snoresville's main bank; a very tempting target for the local thieves. Dozy knows that the defences and procedures are more than good enough to resist attack by

one bandit, and that they can even resist attack by  $N/3$  bandits. He wonders if some sub-set of the collection of local bandits will manage to gang together to grab the gold.

The friendship data for the bandits can be represented in the following way: A '1' indicates a friendship with another bandit. If one bandit is known then he also knows of the bandit who knows him, i.e. friendship is mutual. Each line ends with a '1' because each bandit knows himself, obviously. Explain why the triangular table given is all that's needed to detail all the friendships, and show how it can be expanded into a square to give the friendship data with a row for each bandit. [2 marks]

```

1
01   N = 5
111
0101
10011

```

Given a global data structure defined as an  $N$  by  $N$  matrix (`int [][]gang`), filled initially with zeros, and where  $N$  is the largest number of bandits to consider, give two Java procedures:

1. `generate()`: This should input from the keyboard an integer  $n$ , and a floating-point  $p$ . Check that  $n$  is in the range  $0 < n \leq N$ , and that  $p$  is a valid probability between 0 and 1. The procedure should then randomly generate a triangular table as above, such that the probability of any of the points being '1' is  $p$ , except for each bandit with himself which should always be '1'.
2. `square()`: This should take the triangular table now in `gang` from the generate function and transform it into a square table as discussed earlier. It should then display the block in a tabular form. [6 marks]
3. A group of bandits is only possible when each bandit knows every other bandit in the group. In the example given above, what is the group with largest cardinality? [1 mark]
4. Examine the following pieces of code; the procedure `compare` and a fragment of code to show how it is called.

Calling Fragment:

```

for (i=0; i<n; i++)
{
    compare(i, gang[i]);
    for (j=0; j<n; j++)
    {
        gang[i][j]=0;
        gang[j][i]=0;
    }
}

```

```

    }

Main example

int maxgroup[N];
int max=0;
/*
 * NOTE: current[N] is one array of the array of
 * arrays gang[N][N]
 */
void compare(int i, int current[N])
{   for (int j=0; j<n; j++) current[j] &= gang[i][j];
    for (int k=i+1; k<n; k++)
    {   if (current[k]==1)
        {   compare(k, current);
            current[k]=0;
        }
    }
    if (count(current)>max)
    {   max=0;
        for (int k=0; k<n; k++)
        {   if (current[k]==1) maxgroup[max++]=k;
        }
    }
}

```

5. Explain in detail what the function `compare` does, and how it works, paying special attention to the recursive nature of the procedure and the actions of the calling fragment, and giving a function `count` which returns the number of '1's in the array `current`. [8 marks]
6. Outline a function `main` to bring all this together and for each  $N$  in the range 2 to 20 calculate an average, over 10 tries, of the cardinality, given  $p = 0.5$ , and display the smallest  $N$  for which the cardinality is less than  $N/3$ . This is the critical number of bandits for the gold safety, and for sheriff Dozy not to lose his job. [3 marks]

## 8.14 Exception

Describe some circumstances where it is useful for functions to return errors as exception, and some where it is not. Give an example of an algorithm which is simplified by the use of exceptions.

## 8.15 Features

Write brief notes on *four* of the following aspects of Java. In some cases it may be appropriate to compare what Java does in the situation with other programming languages such as ML.

1. Using the same name for several different functions;
2. Data types where a single type has several variants;
3. Programs that live in several source files;
4. Inheritance and abstract classes;
5. The degree to which code will behave the same when run on different computers.

## 8.16 More features

For *five* of the following Java features write a very short code fragment (it does not have to be complete, and perhaps 2 or 3 lines will suffice in most of the cases) that illustrates the syntax involved. In each case explain briefly what your example achieves.

1. Declaration of constants;
2. Casts between class types;
3. The two styles of comment;
4. Catching an exception;
5. The `switch` statement, including a default label;
6. Summing all the `BigInteger` values that are in an array;
7. The class `Object`.

## 8.17 Debate

A grand debate is being planned by a society that has among its members a large number of computer professionals and working programmers. Arthur will propose a motion “That this house considers ML to be a much better programming language than Java”, while Larry will lead the opposition under the banner “Java is the language for any programmer who seeks employment and hence who can truly be referred to as *working*”. Organise the points in support of the two languages that each proponent will use to justify their position, and identify areas where they are liable to find common ground.

You are not expected to reach a definite conclusion about which way the vote will go at the end of the debate: your job is just to collect and organise the arguments so that comparing and contrasting the merits of the two language becomes easy.

## 8.18 Design

You have been invited to start a project to build a program that will check ML programs to see if they have missing punctuation marks or other mistakes, so that this check can be performed before the ML code is forwarded for full execution. So far all you know about what is wanted is the above. Without concerning yourself with fine detail of how the checking will be implemented, identify and discuss:

1. questions you might want to ask the client organisation about its needs before deciding on any more details of your design;
2. choices or options available to you when making a detailed project plan;
3. ways of partitioning the whole job into a handful of separate modules that could be implemented more or less independently;
4. plans for testing the code you write and determining whether, when notionally finished, it has met its objectives.

## 8.19 Filter (Coffee?)

The structure of a binary tree containing integers at just some of its leaves could have been given by the ML type  $T$  defined as follows

```
datatype T = X | N of int | D of T*T;
```

Define a Java class or set of classes that can be used to represent trees in the same general form. Then add a method `filter` which uses an integer  $k$  and a tree. Its job is to simplify trees by using the rule that and sub-node (type `N` in the ML declaration) where the integer stored is  $k$  gets turned into an `X` leaf. Then any node  $(X, t)$ , or  $(t, X)$  [ie nodes of the ML type `D` where one of the sub-trees is an `X`] get turned into just  $t$ .

Thus for instance if  $k = 0$  the initial tree  $((0, 0), ((2, 0), 3))$  would simplify to just  $(2, 3)$

## 8.20 Parse trees

What does it mean for a Java class to be `abstract`?

A Java program includes the following class declarations:

```
abstract Class Node
{
    public int eval();
}
Class Num extends Node
{
    int value;
    public Num(int value) { this.value = value; }
    ...
}
Class Op extends Node
{
    String sym;
    Node left;
    Node right;
    public Op(String sym, Node left, Node right)
    { this.sym=sym; this.left=left; this.right=right; }
    ...
}
```

The objective of the programmer who wrote this was to be able to write assignments such as

```
test = new Op("*", new Num(4),
             new Op("-", new Num(7), new Num(2)));
```

The variable `test` is of type `Node` which can cover either of the two concrete cases of `Op` (representing a dyadic operator together with its two operands) or `Num`

(a number). Thus the above assignment sets up a representation of the expression  $4 * (7 - 2)$ .

The abstract class `Node` declares a method `eval()`. Fill in the dots in the other two classes with code that overrides this so that calling the `eval` method on a `Node` returns the value of the arithmetic expression it represents, supposing that the only operators that will be used are plus, minus and times.

## 8.21 Big Addition

Java comes with a class `BigInteger` that represents potentially huge numbers. Suppose it did not, or for some reason you were prohibited from using it but still needed to work with large positive integers. To fit your needs you will define a new class called `Big` that stores integers as arrays of `byte` values, where each byte holds a single decimal digit from the number being used, with the least significant digit held at position 0 in the array.

Write a definition of such a class including in it methods to create a big integer from an `int` (provided that `int` is positive), to add two `Big` values together and to convert from a `big` to a `String` ready for printing. You need not implement any other methods unless they are needed by the ones mentioned here.

## 8.22 Lists in Java

A list in Java can be represented as a sequence of links. Each link is an object containing one value in the list and a reference to the rest of the list following the link. A `null` reference indicates the end of the list.

Write a Java class that can represent such lists, where the items stored in lists are of type `Object`. Provide your implementation with two static public methods that append lists. The first of these should be called `append` and should take two arguments, its result should be the concatenation of the two lists and neither input should be disturbed. The second should be called `conc` and should have the same interface, but it should work by altering the final reference in the first list to point it towards the second, and it should thus not need to use `new` at all.

## 8.23 Pound, Shillings and Ounces

The Imperial system for Sterling currency was based on the *pound*, *shilling* and *penny* (plural *pence*). There were 12 pence in a shilling and 20 shillings in a pound. A Java class that could store amounts in this format might be



```

class LSD
{
    boolean negative;
    int pounds;
    int shillings;
    int pence;
    LSD(boolean m, int l, int s, int d)
    { ... }
    ...
}

```

Adjust or finish off the constructor so that it raises an exception of class `BadInput` (which may be supposed to have been defined already) if the input is invalid, ie unless the number of pence is from 0 to 11 and the number of shillings from 0 to 19, and the specified number of pounds is positive.

Now you need to provide a `toString` method in the class that converts currency into textual form. The following table shows the desired effect when a number of pence is first converted to `LSD` and then to a string:

0	⇒	zero
1	⇒	1 penny
10	⇒	10 pence
60	⇒	5 shillings
80	⇒	6 shillings and 8 pence
252	⇒	1 pound and 1 shilling
479	⇒	1 pound, 19 shillings and 11 pence
1201	⇒	5 pounds and 1 penny
2400	⇒	10 pounds
-252	⇒	minus 1 pounds and 1 shilling

Credit will be given for a clearly explained, concise and tidily presented solution. Minor syntax or punctuation errors in the Java code will not count heavily against you.

## 8.24 Details

Give a brief explanation of each of the following aspects of Java

1. The difference between `>>` and `>>>`;
2. The possibility that in some program the test `(a == a)` might return the value `false` for some variable `a`;
3. The keywords `final` and `finally`;

4. The expression "three" + 3 and other expressions of a generally similar nature;
5. The meaning of or errors in (whichever case is relevant!)

```
...
int [10] a;
for (int i=1; i<=10; ++i)
    a[i] = 1-a[i];
...
```

## 8.25 Name visibility

A complete Java program may use the same name for several different methods or variables. Java has a number of features that allow the user to prevent such re-use of names from causing chaos. Describe these under the headings:

1. Scope rules within individual functions; [6]
2. Visibility of method names within classes, and the effects of inheritance; [8]
3. Avoiding ambiguity when referring to the names of classes. [6]

## 8.26 Several Small Tasks

Write fragments of Java definitions, declarations or code to achieve each of the following effects. You are not expected to show the whole test of a complete program — just the parts directly important for the task described, and you may describe in words rather than Java syntax any supporting definitions or context that you will want to rely on. Clarity of explanation will be viewed as at least as important as syntactic accuracy in the marking scheme. It is also understood that names of methods from the standard Java class libraries are things that programmers check in on-line documentation while writing code, so if you need to use any of these you do not need to get their names or exact argument-format correct provided that you (a) describe clearly what you are doing and (b) your use is correct at an overview level:

1. Take a `long` argument called `x` and compute the `long` value obtained by writing the 64 bits of `x` in the opposite order; [6]

2. Define a class that would be capable of representing simple linked lists, where each list-node contains a string. You should show how to traverse such lists, build them and how to reverse a list. In the case of the list reversing code please provide two versions, one of which creates the reversed list by changing pointers in the input list, and another which leaves the original list undamaged and allocates fresh space for the reversed version; [8]
3. Cause a line to appear in a the window of an applet running from the bottom left of the window towards the top right. Your line should remain visible if the user obscures and then re-displays the window, but you can assume that the size of the windows concerned will be fixed at 100 by 100 units. [6]

## 8.27 Some Tiny Questions

1. List the eight Java primitive data types.
2. What result will be printed if the following fragment of Java code is executed? Why?

```
double d = 6.6;
try
{   d = 1.0 / 0.0;
}
finally
{   System.out.println("d = " + d);
}
```



Figure 8.1: Remember: programming is fun!

# Chapter 9

## Java 1.5 or 5.0 versus previous versions

The Java course from 2005 onwards uses a version of Java (and its libraries) that support a range of things that earlier release did not. The commentary about these features here is not part of the examinable content of the course, but may help those who want their code to be backwards compatible, and may help supervisors understand how features new in 1.5 are relevant in an introductory Java course.

### 9.1 An enhanced `for` loop

New Java allows direct iteration over either arrays or collection types using syntax along the lines of

```
for (Type s : arrayOrCollection) use(s)
```

With previous versions you would need

```
for (int i=0; i<array.length; i++) use(array[i])
```

for arrays, or the yet more clumsy

```
for (Iterator i=collection.iterator(); i.hasNext();)
{
    Type s = (Type)i.next();
    use(s);
}
```

There are special delicacies to watch with the use of nested iterations if you use the old versions, and a strong interaction with the generics feature described next.

## 9.2 Generics

Most of the examples that use collections in these notes decorate the type declarations of the collections to show what they contain. For instance there could be use of `HashMap<String,String>` for a `HashMap` that will only be used with strings as both they keys and values it stores. With these decorations code is naturally type-safe. Older Java does not support this. The result is that instead all generic structures use the fall-back `Object` type. When data is retrieved from then the Java type-checker does not know what sort of `Object` is being used, and so casts, involving run-time checks, have to be used.

## 9.3 `assert`

The `assert` keyword came in with Java 1.4. For use with any earlier version of Java you must remove them all. Please ensure that your code is fully debugged first!

## 9.4 Static imports

If your new code includes a line such as `import java.Math.PI` and you then use the constant `PI` in your code you will need to remove the import statement and use the longer name `Math.PI` everywhere that you reference it.

## 9.5 Auto-boxing

With both collections and the `printf` facility listed later, you can use the built in types (eg integers and reals, characters and booleans) without having to worry too much about special consequences of them being primitive built-in types rather than part of the Java class system. In older versions of Java this is not the case, and you need to make much more explicit use of the wrapper classes `Integer`, `Double` and so on. The effect is much messier code in places! The term “auto-boxing” refers to the fact that these wrapper classes are still used, and use of a constructor `new Integer(1)` is referred to as “boxing” the integer up.

## 9.6 Enumerations

In Java 1.5 you may have used `enum` to introduce a collection of distinct names, and you may then have used these enumeration values in `switch` statements. Be-

fore Java 1.5 you could either use an explicit encoding as integers (which does not protect you from type errors) or some more elaborate scheme that uses the class system to protect details of the implementation. In any case the effect is significantly more clumsy than the new scheme.

## 9.7 `printf`

When you look at many existing Java books and sample codes you will see printing done using the idiom

```
System.out.println("The result is" + i);
```

where these notes have written something more like

```
System.out.printf(
    "The result is %d (or %<x in hex)%n", i);
```

These notes have preferred the second if only because the format string item `%d` makes explicit what type of item is being displayed, and so there is an extra check on internal consistency in your code. Format conversions provide amazing (and sometimes complicated) levels of refinement in controlling just how simple information such as numbers are to be laid out. Replicating that control using the facilities from previous Java releases is tedious and leads to fairly bulky and unreadable mess.

## 9.8 `Scanner`

If you needed to split an input file into words or wanted to read in a simple column of numbers you may have used `Scanner`. Beware because it is new and you would need to do things by hand to if backwards compatibility was essential. In big programs `Scanner` may not matter but I find it jolly handy in all sorts of small examples.

## 9.9 Variable numbers of arguments for methods

The `smallVarargs` facility is something you should probably not often use directly in your own code, but it is exploited by `printf` and friends. The old way of achieving a similar effect is to make your function accept an array of items, and construct a new array to pass arguments in each call you make to it.

## 9.10 Annotations

Java 1.5 provides a general framework for adding annotations to your code in a way that is expected to make it easy for program management tools to extract them. This is a relative of the notation `/**` that inserts “document comments” that `javadoc` can extract. The scheme is not mentioned in this course beyond the fact that the new-style annotations are introduced by a name preceded by an at-sign (`@`), and new styles of annotation can be declared using the keyword `@interface`. Any code that has this syntax will need it removed for use with earlier versions of Java. This will not change the behaviour of the code itself in any way, but would have an effect on how it could interact with annotation-processing tools (see the Java documentation for the command `apt`). For those who want to investigate further it might be useful to note that the Java reflection mechanisms can detect when a method defined in a class has been annotated. A lot of the time you will not use annotations, but when you do they may be a huge help and you would hate to go back to a system without them.

## 9.11 Enhanced concurrency control

For example `ConcurrentHashMap`, `EnumSet`.

Many of the above features interact together so that the savings of using them combined are even greater than using them one at a time. A consequence of that is that giving them up would be even more painful than you might at first expect. I fully expect to see many of them becoming the standard idiom for Java programmers in the very near-term future.



# Bibliography

- [1] Abramowitz and Stegun. *Handbook of Mathematical Functions (with formulas, graphs and mathematical tables)*. Dover, 1965.
- [2] David Barnes and Michael Kölling. *Objects First with Java: a Practical Introduction using BlueJ*. Prentice Hall/Pearson, 2 edition, 2005.
- [3] Colin Bentley. *Introducing SSADM 4+*. NCC Blackwell, and also see <http://www.blackwellpublishers.co.uk/ssadmfil.htm>, 1996.
- [4] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986.
- [5] Jon Bentley. *More Programming Pearls*. Addison-Wesley, 1988.
- [6] Judy Bishop. *Java Gently*. Addison Wesley, 3 edition, 2001.
- [7] Fred Brookes. *The Mythical Man Month*. Addison Wesley, 2 edition, 1996.
- [8] J. Cameron. *JSP and JSD: The Jackson Approach to Software Development*. IEEE Computer Society Press, 1989.
- [9] Leiserson Cormen and Rivest. *An Introduction to Algorithms*. MIT and McGraw-Hill, 1990.
- [10] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [11] Bruce Eckel. *Thinking in Java*. Prentice Hall, 1998.
- [12] Christopher Essex, Matt Davison, and Christian Schulsky. Numerical monsters. *SIGSAM Journal*, 34(4):16–32, December 2000.
- [13] John McCarthy et al. *The Lisp 1.5 Programmer's manual*. MIT Press, 1965.
- [14] M. H. Halstead. *Elements of Software Science*. Elsevier North Holland, 1977.
- [15] Henry S Warren Jr. *The Hacker's Delight*. Addison Wesley, 2003.

- [16] Brian W. Kernighan and Dennis M. Richie. *The C programming language*. Prentice-Hall, 1978.
- [17] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes and CUP, 1992.
- [18] Steve Maguire. *Writing Solid Code*. Microsoft Press, 1993.
- [19] C.A.R. Hoare O.-J. Dahl, E.W. Dijkstra. *Structured Programming*. Academic Press, 1972.
- [20] Eric S Raymond. *The Cathedral and the Bazaar*. O' Reilly, 1999.
- [21] Guy Steele. *Common Lisp the Language*. Digital Press, 1990.
- [22] X3J11. *ANSI X3.159, ISO/IEC 9899:1990*. American National Standards Institute, International Standards Organisation, 1990.