Programming in Java

A C Norman, Lent Term 2006

Part IA

Contents

Chapter 1

Preface

1.1 What is programming about?

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 the 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 comfortable 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 then 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 then 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[1] 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.

Figure 1.1: Silver Bullet Needed.

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.

---

[1] 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 this year will be based on use of the version of Java sometimes known as "Java 5.0" and sometimes as "Java 1.5". 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.

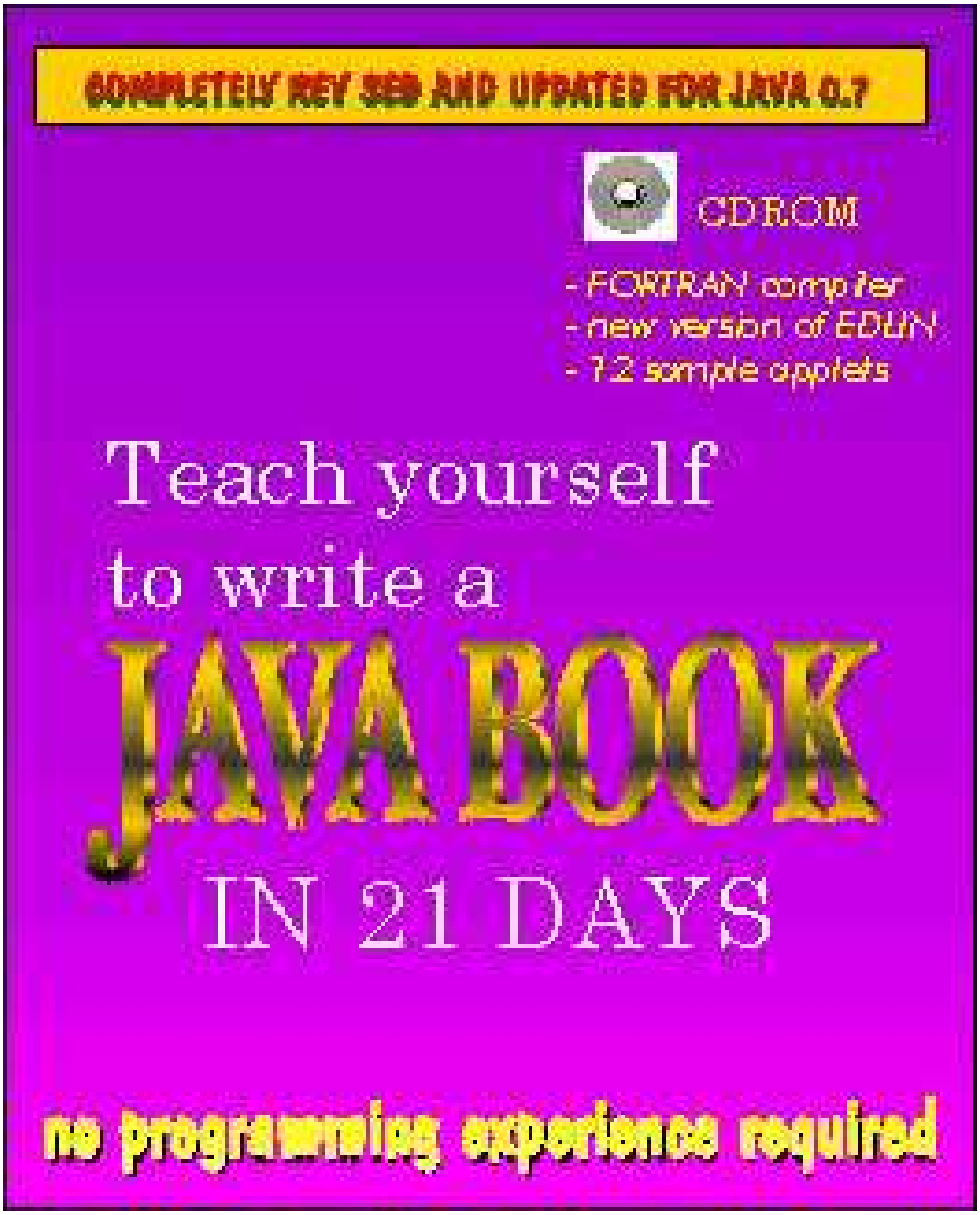


Figure 3.1: Reproduced courtesy Kevin McCurley.

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
Note that the fifth edition is due to be published as your course begins!

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` 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*

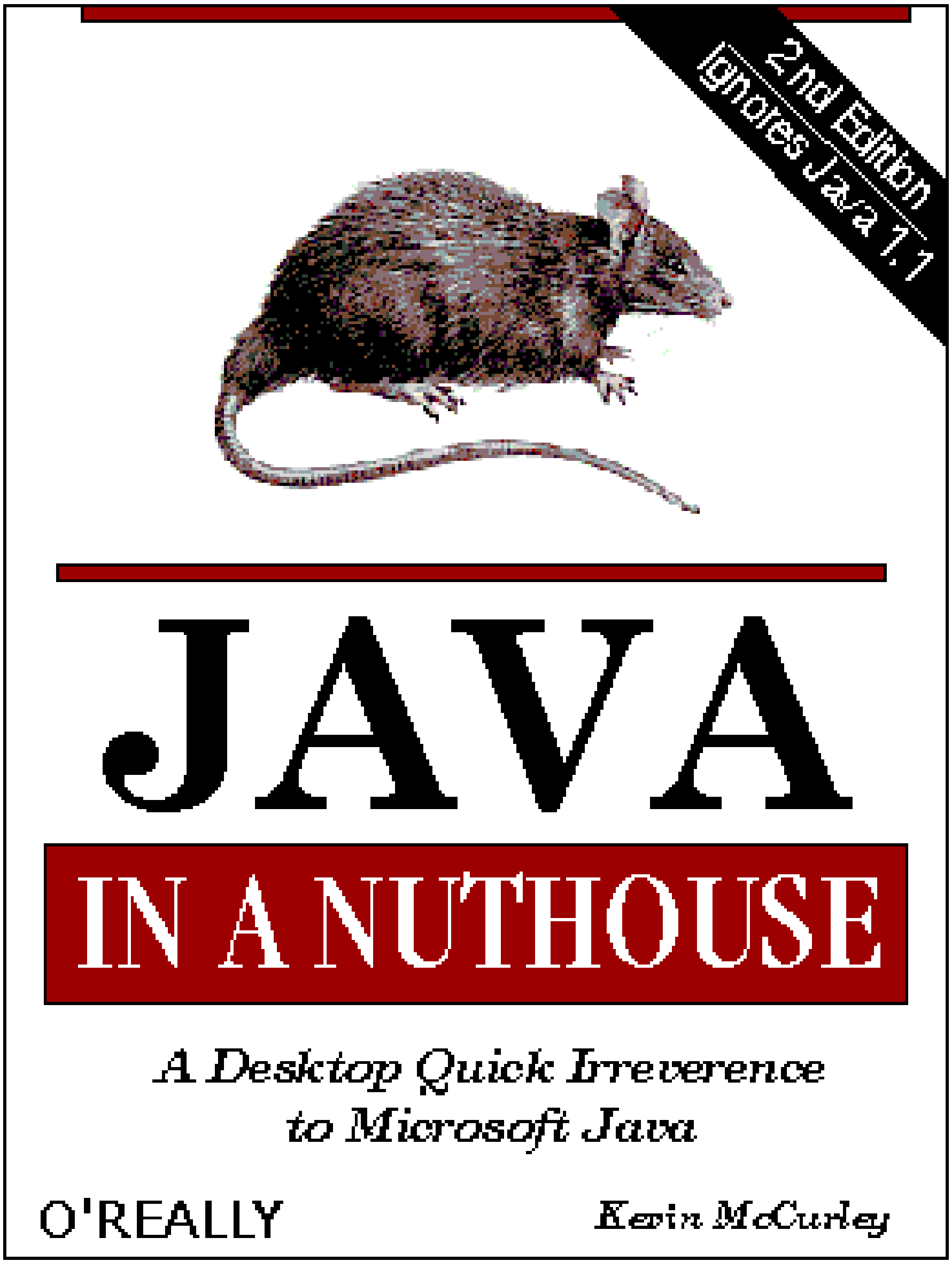


Figure 3.2: Not (quite) the main course book.

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 if 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[1]. 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

---

[1] 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.

PWF by using `ssh` and an “X-windows server” to access of of the lab’s PWF linux systems that are set up for remote use, eg `linux2.pwf.cl.cam.ac.uk` or `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[2], 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 nay 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.

---

[2] Great thanks are due to the Computing Service for ensuring that this is the case.

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

```
mkdir Tick1B
cd Tick1B
```

Issue the following commands that copy two files form 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.

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/
```

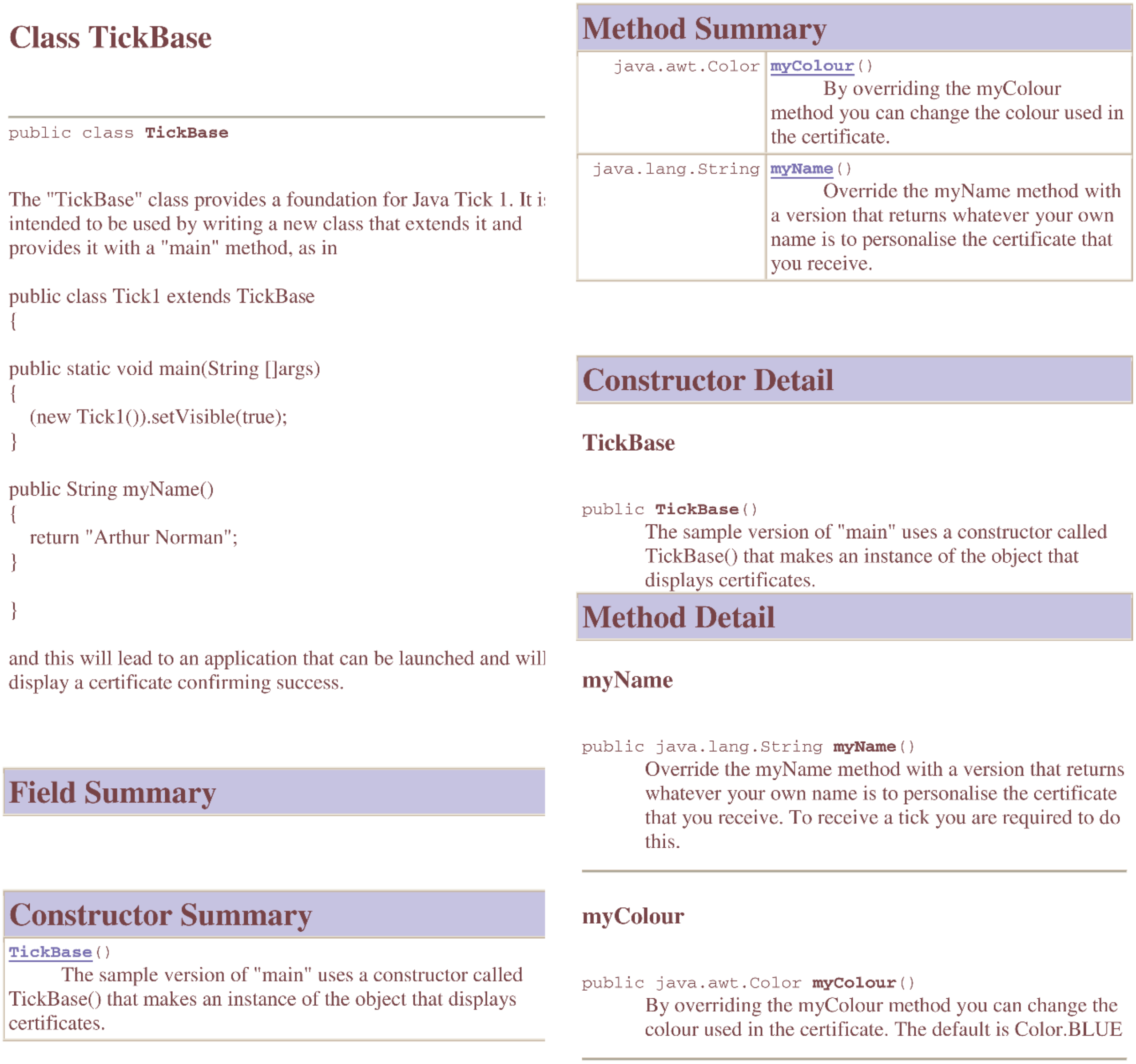
Now prepare a file that called `Tick1.java` containing the text

```
// 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 i 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`	**`myColour`**`()` By overriding the myColour method you can change the colour used in the certificate.
`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.

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 or 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`[3] 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[4], 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[5] 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.

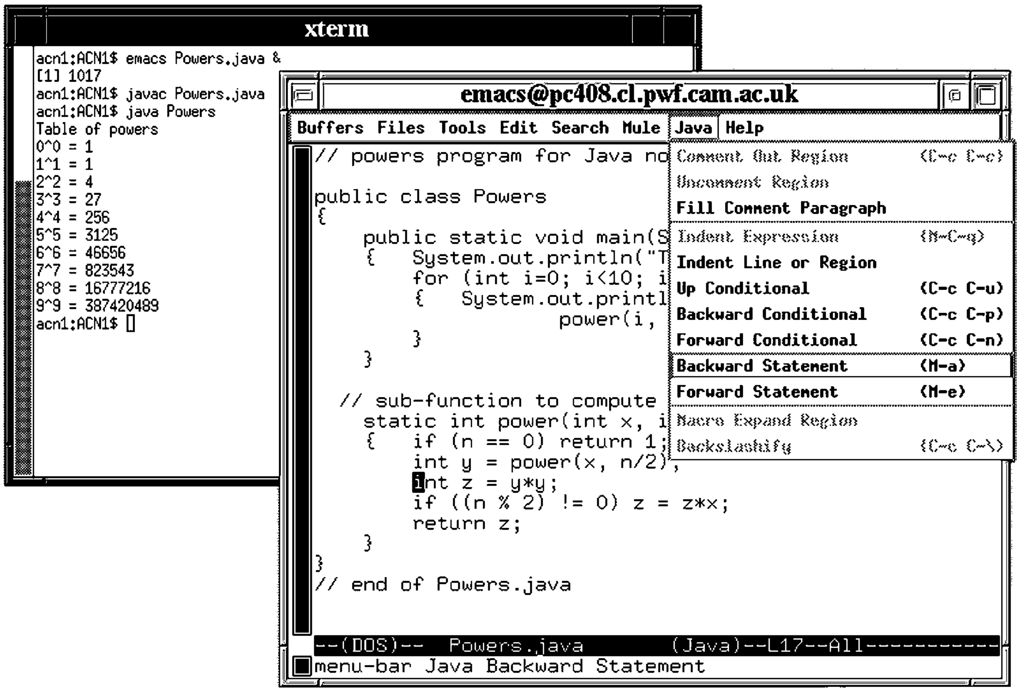
---

[3] Many Java texts use a function `println` here rather than `printf`.

[4] 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!

[5] 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[6]. 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!



```
acn1:ACN1$ emacs Powers.java &
[1] 1017
acn1:ACN1$ javac Powers.java
acn1:ACN1$ java Powers
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
acn1:ACN1$
```

```
// powers program for Java no

public class Powers
{
    public static void main(S
    {   System.out.println("T
        for (int i=0; i<10; i
        {   System.out.printl
                    power(i,
        }
    }

  // sub-function to compute
    static int power(int x, i
    {   if (n == 0) return 1;
        int y = power(x, n/2);
        int z = y*y;
        if ((n % 2) != 0) z = z*x;
        return z;
    }
}
// end of Powers.java

--(DOS)--  Powers.java       (Java)--L17--All-----------
menu-bar Java Backward Statement
```

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

---

[6] 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[7] 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");
    }
}
```

---

[7] 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.

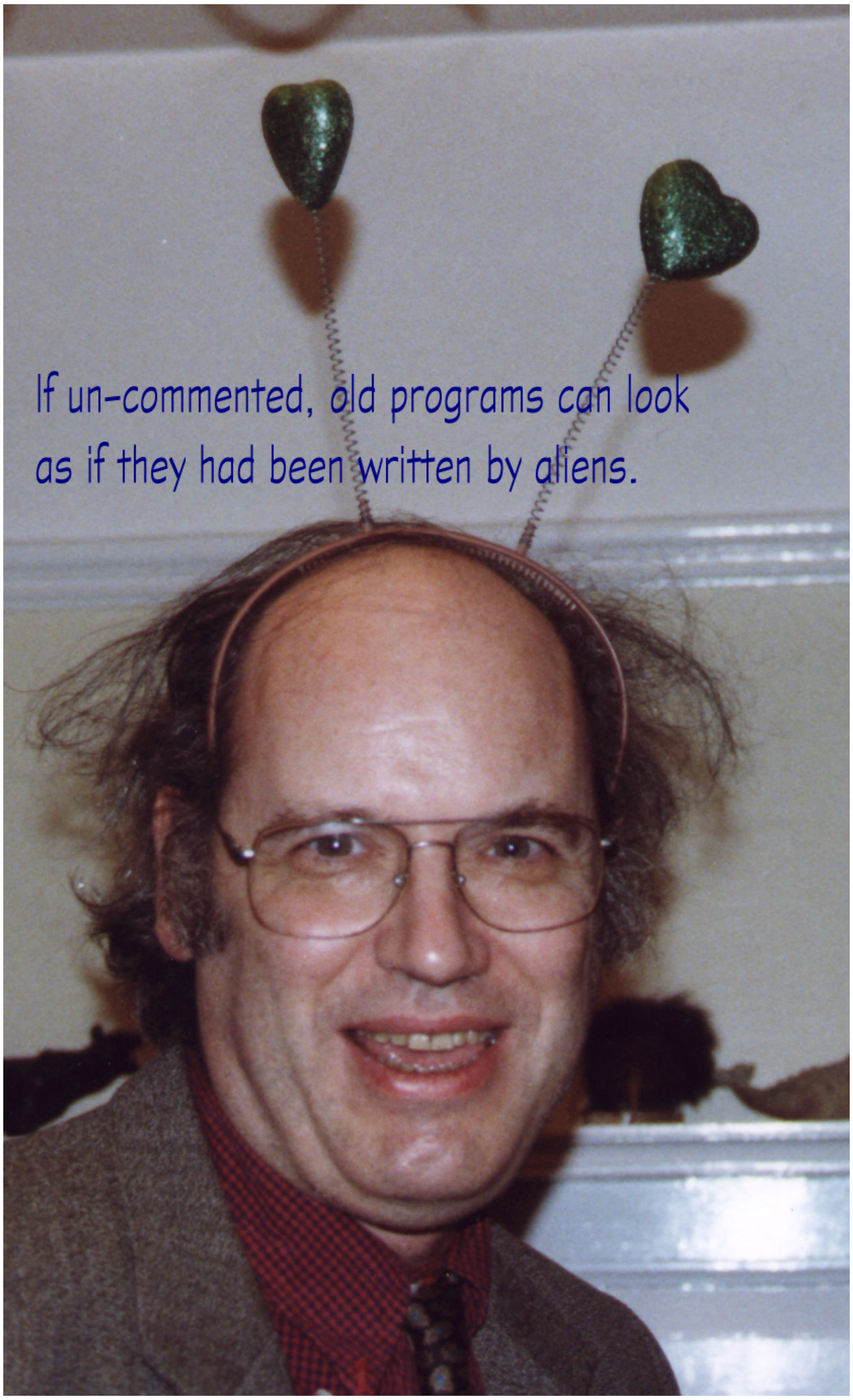


Figure 3.6: Style does matter.

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 "`/**`"[8] 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-

---

[8] Ordinarily as well as "`//`" comments that just run to the end of the line you can write long comments starting with "`/*`".

readable[9] source file such as `Hello.java` into a more compact[10] 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"[11] 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:

---

[9] Well, at least it is readable if you include enough comments!

[10] 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.

[11] 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.

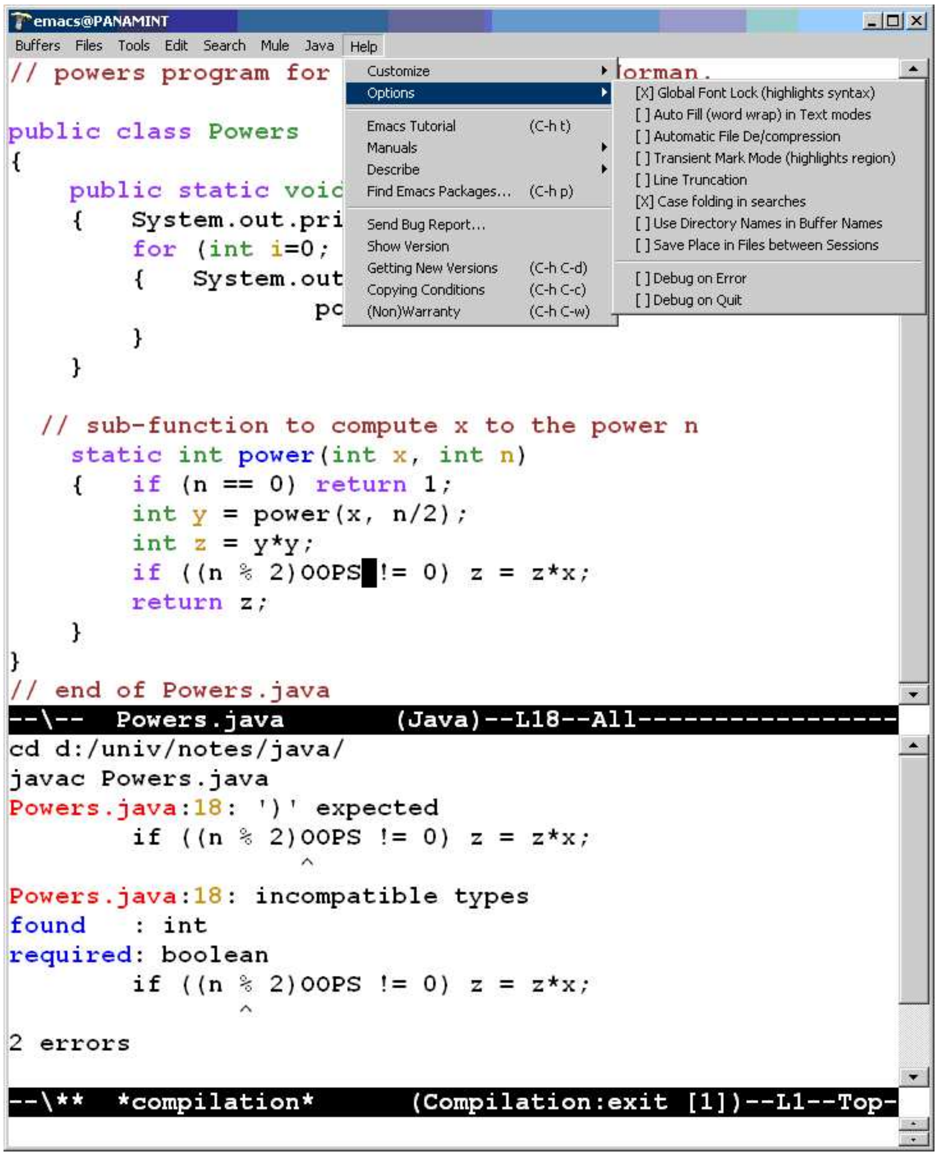
You need to use the word `return` explicitly to indicate what value your procedure should hand back.

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 refereed 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



[X] Global Font Lock (highlights syntax)
[ ] Auto Fill (word wrap) in Text modes
[ ] Automatic File De/compression
[ ] Transient Mark Mode (highlights region)
[ ] Line Truncation
[X] Case folding in searches
[ ] Use Directory Names in Buffer Names
[ ] Save Place in Files between Sessions
[ ] Debug on Error
[ ] Debug on Quit

```
// powers program for                              orman.

public class Powers
{
    public static voic
    {   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
--\--  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 image 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`[12]. 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!

---

[12] 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 lastY = 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[13] 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

---

[13] 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 $2\ldots\sqrt{n}$.

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 $\mod 30$. Ie 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 computer the same answers! The binomial coefficients[14] may be defined by the rules

$$\begin{aligned} {}^{n}C_{r} &= {}^{n-1}C_{r} + {}^{n-1}C_{r-1} \\ {}^{n}C_{0} &= 1 \\ {}^{n}C_{n} &= 1 \end{aligned}$$

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 ${}^{2n}C_{n}$ 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)

---

[14] There is some question as to whether I should use the notation ${}^{n}C_{r}$ here or $\begin{pmatrix} n \\ r \end{pmatrix}$. 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

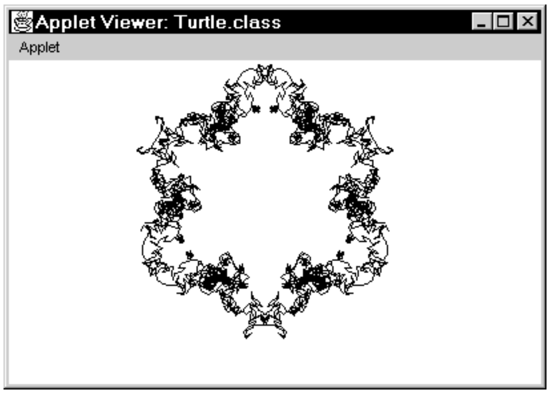
   ```
   g.setColor(Color.blue); // or red, black etc
   ```

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 redrawing, 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

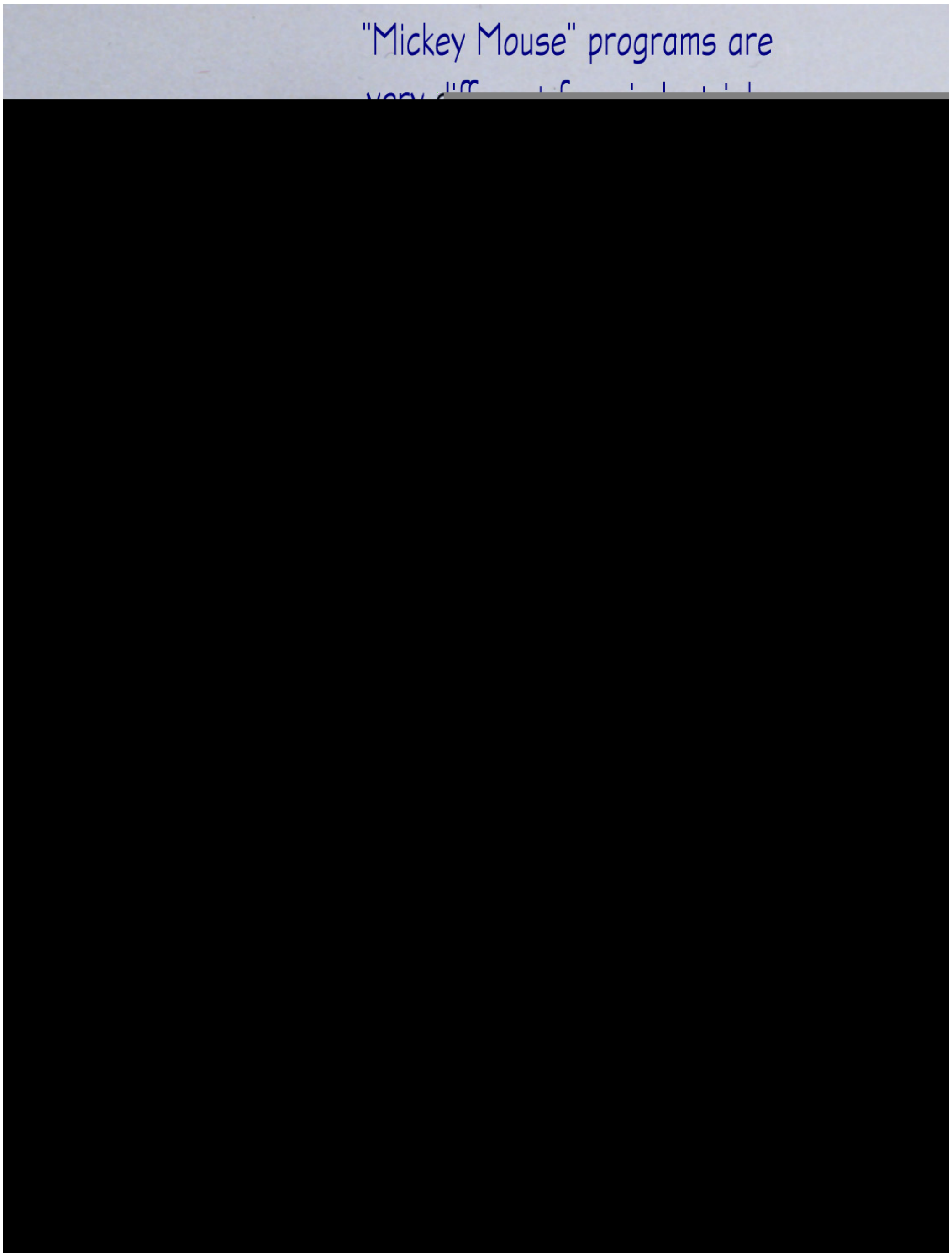
Basic use of Java

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



"Mickey Mouse" programs are

[redacted]