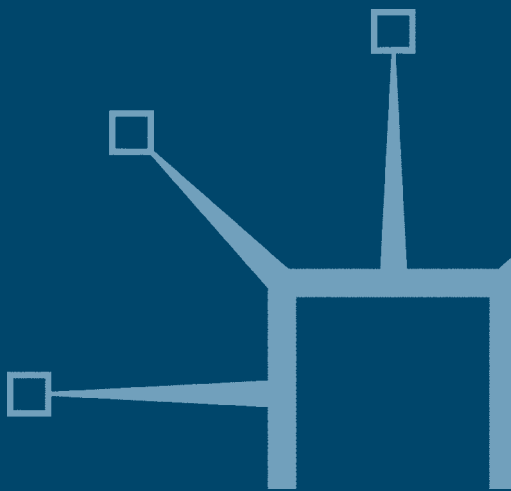
palgrave
macmillan

How to Program Using Java

Tony Jenkins and Graham Hardman

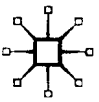


How to Program Using Java

Tony Jenkins
Graham Hardman

Illustrations by Christine Jopling

palgrave
macmillan



First published 2004 by
PALGRAVE MACMILLAN
Houndmills, Basingstoke, Hampshire RG21 6XS and
175 Fifth Avenue, New York, N.Y. 10010
Companies and representatives throughout the world

PALGRAVE MACMILLAN is the global academic imprint of the Palgrave Macmillan division of St. Martin's Press, LLC and of Palgrave Macmillan Ltd. Macmillan® is a registered trademark in the United States, United Kingdom and other countries. Palgrave is a registered trademark in the European Union and other countries.

ISBN 1–4039–1223–8

This book is printed on paper suitable for recycling and made from fully managed and sustained forest sources.

A catalogue record for this book is available from the British Library.

10 9 8 7 6 5 4 3 2 1
13 12 11 10 09 08 07 06 05 04

Printed and bound in China

This book is respectfully dedicated to all SysAdmins.
Now can we have some more disk quota, please?

Using this book

Deciding what to read

This book is meant to be read by someone who is learning to program. It is not meant to be a reference. The first chapter explains what's in each of the following chapters in some detail, but briefly:

If you're approaching programming as a complete novice you should aim to work through these chapters in this order. Don't be tempted to skip straight to the chapters with Java in! It's extremely important that you understand what you're trying to achieve and the best ways of achieving that before you go anywhere near any programs. This might seem odd, but please bear with us!

If you've already programmed in some other language (particularly something like Pascal or C, or definitely if you've met C++) and just want a flavour of Java you're probably safe to skip on to Chapter 6 or 7. It would still be a good plan to skim through the earlier chapters, though.

At the end of the book you'll find a quick Java Reference, a Glossary and an Index. This is where to look when you realise that you need that little bit of information but can't remember where it was.

Understanding what you read

There are some conventions that you need to keep in mind as you read.

In the text, anything in `this font` is a Java statement or name. Anything in this font is not.

All programs and classes also appear in `this font`. Like this:

```
/*
  Duck.java
  A simple Duck class.

  AMJ
  22nd January 2003
*/

public class Duck
{
  private String name;
```

```
    public Duck ()
    {
    }
}
```

Anything not in `that font` is not a program. Fragments of programs also appear in `this font`:

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

Anything in `this font` is correct Java.

Where a user is entering a value, the user's typing is shown in **`bold`**:

```
Enter your name: Tony
```

Sometimes it is necessary to show just the general format of the Java statement. This appears like this:

format of a Java statement

For example:

<type> <identifier>;

Anything that appears between < and > in these examples is a description of what is required in the program. Examples presented like this are not valid Java!

Finally, sometimes there are things that need to be laid out in a way that looks like a program, but isn't a program. This appears in the same font as a program, but in *`italics`*, like this:

```
IF the value of "sold to" is blank THEN
  Set the value of "sold to" to the Reserve object
OTHERWISE
  Display an error message – the Duck is already sold
END IF
```

This is not valid Java.

There are many definitions in the book. Words that are defined in the Glossary at the end appear *like this* in the text.

A note on programming style

You will learn that programming style is an important element of writing a program. Style refers to the way in which a programmer lays out a program, what names are chosen for various things in the program, and what else might be added. Programming style is a very individual thing, and develops in any programmer over many years; it's very much like handwriting.

The programs and examples in this book were written by two people. We each have rather different programming styles, so we've negotiated and agreed to adopt a "house style". You cannot imagine the bloodshed. We think that the style we have finally agreed to adopt is a reasonable compromise and that it should be reasonably clear.

Develop your own style as you learn to program. Copy ours for the moment if you want to, but if you find things that you don't like don't follow them. Just be consistent.

A note on persons

You will discover soon that the chapters in this book make use of the first person, and the first person singular in particular. This might seem odd in a book that has been written by two people. The thing to remember is that each chapter was, in fact, written by just one person. You can amuse yourself, if you like, by trying to work out who wrote which one … There is the occasional clue.

To the Student

Hi.

Welcome to the book. We hope you like it.

If you're a student just starting out on your first programming course, this book is for you. This book contains what we think you'll need to know as you go through your course. We very much hope you'll enjoy reading it and come to enjoy programming.

Because we want to get something absolutely clear before we go any further; programming *is* enjoyable. It's a creative pastime, and has been called by some a craft. Writing a program is the process of creating something from nothing – the process of creating something that solves a real problem and hopefully makes the world a better place. A complete and correct program can be a source of great satisfaction to a programmer. Even the appearance of the lines of a program laid out on a sheet of paper can be a thing of beauty, almost like a poem.

But enough of these fine words. We would be lying to you if we didn't admit that many people do not especially enjoy learning to program. Many people do find it difficult, but just as many take to it quickly and easily; we suppose that you'll be finding out which one you are in the next few weeks. Whichever turns out to be you, just keep in mind that anyone can get there eventually; there's nothing special about people who can write computer programs.

This book is not like many of the other books on programming that you can see on shelves in the bookshops or libraries. For a start there's no chapter on the history of computers, gloriously illustrated with highly amusing photos and hairstyles from the 1950s. No. We'll be assuming that if you want to look at that sort of stuff you know where to find it, and you'll go and seek it out. No. This is a book about *programming*.

You notice that we say *programming* and not *Java*? That's important. What we are about here is learning to program, and the programming language that we have chosen to use is something called Java. The skills and techniques that you'll find in this book, and which you'll learn, can be applied to many, if not most, other programming languages. A mechanic does not learn to repair just one kind of car, and a chef does not learn to cook just one kind of pie. So a programmer does not learn to program in only one language.

Just reading this book won't turn you overnight into a programmer. Reading this book will help turn you into a programmer, but you're going to have to do other things. You're going to have to write your own programs. You're going to have to practise. There are plenty of our programs in this book, and the first stage for you is to look at these and understand them. Then you're going to have to take the leap of starting to write your own programs. This transition – from understanding a program that someone else has written, to writing your own programs – is difficult, and let no one tell you otherwise. When you've managed it you'll have achieved something that should make you very proud.

The chapters that you're going to read are quite short. This is quite deliberate. With any luck you'll be able to find time to read them between classes or on the bus home, or something. As you read, doodle on the book, or make notes. Read *actively* and think about what you're reading. Learning is about thinking and reflecting, not just about reading.

Right. Sermon over. There's one last practical thing. You'll find out soon that we're going to assume that you've been given a copy of something that we're going to call your *Local Guide*. This should explain how the Java system that you're going to use works, and should fill you in on any other little local details. If you've got that (and there's no need to look inside it just yet), all that remains to be said is …

… let's go and do some programming!

To the Teacher

Hello. Welcome to a book about learning to program.

Before we go any further, you need to be absolutely clear about what it is that you are holding in your hand. This is possibly a book with the name of a programming language in the title that is unlike any book about with the name of a programming language in the title that you have encountered before. And you have probably encountered many. Too many.

A big claim, that. But this is not a book about Java. This is not a book that seeks to explain all the minute details of the Java language.[1] This book contains no UML and none of whatever the flavour of the month is at the moment in systems development. This is not a book that an experienced programmer, working in industry, would use as a reference while working on some commercial project. There are lots of books like that, and lots of books written for experienced programmers, and this is not one of them.

This book came about like this. We'll let Tony explain.

I've been to a few conferences on teaching computing, and I've given a few presentations and so on describing some of my ideas on what's wrong and right with the way we teach programming. I think I've come to the conclusion that there's rather more that's wrong than right. A publisher's rep came up to me at one of these happy events and started to pester me to write my own Java book. I declined, since there were already far too many Java books about and I saw no need to add to this needlessly large pile of paper. More to the point, I didn't know Java, even if it was flavour of the month at the time.

The problem that then emerged was that this was a persistent publisher's rep. I kept finding that she kept popping up in my office. I will admit to having been bought a beer, but despite advice from other authors, I always seemed to miss the free lunch. Eventually, during ITiCSE 2001 at Canterbury, I cracked and agreed to write something. But only on my terms. I was not going to write another totally unnecessary book about Java. I was, in fact, going to write a book about programming and C++. I knew C++, you see. It was sort of last month's flavour that was still quite good.

The C++ book has been and gone. You might have seen it; it's the one with all the sheep. Now we come to the Java version. First, let's be very clear that this is not just the C++ book rewritten in Java. Some of the chapters are similar, yes. The style is not completely different, even if the sheep have mysteriously become ducks. But the whole basic approach and structure have been revisited. The main change is that Java demands a much earlier and deeper discussion of objects (which is present and correct), and that objects need to be used more and throughout. The rest of the material has all been revisited too, and changed where needed. Underneath, though, the approach is the same – the underlying belief is that students need to learn to program, they do not need to learn Java or C++ or some other language. And they need to understand that.

That is why this is a book about learning to program. Specifically this is a book that is intended to support a student following an introductory programming course in further or higher education. There is sufficient Java in this book to be included in such a course; there are also some pointers in the final chapter that would be of interest in the

1 You can probably tell that from the size!

more ambitious courses.[2] *My hope is that after reading this book, and after following your course, a student would be able to write some reasonably complex Java programs and make sensible use of one of the many other Java books that are available.*

Now let me explain why this book is like this. I have taught programming for many years in what is probably one of the most respected university computing departments in the UK. Every year I have some successes, and every year there are failures. I see students struggle with this topic; they are struggling with something that lies at the very heart of our discipline. I often see students suffer as they attempt to come to terms with programming; often I have seen them drop out of their degree simply to avoid more programming. I have certainly seen them carefully choosing course options in future years to avoid anything that resembles programming.[3] *Your students might be different, but somehow I doubt it (and if you think they are I respectfully recommend a second, closer, look). This sad state of affairs just cannot be right.*

One aspect of this problem (or at least one issue that contributes to the problem) is the nature of the programming textbooks available. These are often weighty tomes indeed, and many run to well over 1000 pages. You know the ones I mean. Most contain far more than could ever be learned effectively in a single course that is, after all, only one part of what a student is expected to study during the year. These books (there are a few honourable exceptions, of course) simply do not meet the needs of our students.

There was one last thing that I had to do before starting on the Java book. I had to learn some Java. Oddly enough, I did not seek out a Java programming course. I did not sit in a room with 200 or more other people trying to learn Java. No. I found someone who knew some Java (so a welcome to Graham!), I got him to tell me the basics, and then I had some fun writing some programs. Isn't it odd that we still expect students to learn to program from attending our lectures?

Now, this book works like this. It not our intention, or our place, to try and replace your lectures. The place of this book is to support your lectures by providing something that your students will actually read, hopefully before your lecture. Our job is to explain to them what's coming up, why it's important, and why it's useful. Each chapter should occupy about a week in a 20-week course; for a student this week should probably include a couple of lectures, some supervised practical time, and opportunities for plenty of practice. There are some exercises at the end of each chapter; please add in some of your own to fit your own local system or needs.

You might think that sometimes our explanations are a little simplistic. Sometimes we admit that we are, in Civil Servant terms, "economical with the truth". This approach is essential with a language as complex as the language that Java has now become. There are so many little details that can tend to get in the way of the real business of the day, which is to learn to program. Sometimes we've added a more complete explanation as a footnote; you might well want to go into more detail, particularly with your more experienced or advanced students. It's up to you.

As for the technical details, all the programs in this book have been written and tested using, at the earliest, version 1.4.0 of the JDK running on a Linux platform. We believe that all the programs work (except where stated otherwise),

---

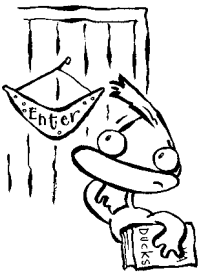
2 But those teaching more ambitious courses would do well to ponder whether it is better for a student to understand a little Java or to be totally baffled by a lot of Java.

3 At Leeds there is a second year course on *Linear Programming*. They avoid that too. Just in case.

and should work unchanged with other comparable Java systems. One issue might be that we've chosen to use the `ArrayList` structure, which only appeared in JDK version 1.4.

There's one last thing we need you to do before we start. We don't know what Java system you're planning to use. We know nothing about your editor-of-choice, and we don't even know what operating system you're using. To be honest, we don't much care. We've ignored all these issues, since we want this book to be useful to everyone. Obviously, though, there are some things your students need to know. As you read through the book you'll see that we've told them about something called the *Local Guide*.[4] We need you to put this together for us. Here you can describe how your system works, you can set down coding or layout standards if you want to, and you can pass on any other local wisdom. You could probably combine it with some existing set of notes. We hope that's not too much work for you. Thanks. The exercises at the end of Chapter 2 include most of the things that we need you to make sure that your students know.

Finally, we hope you agree with our reasons for writing this book. If you do, we're ready to go and teach some programming!



4 If you were to suspect that we've "borrowed" this idea from Leslie Lamport's LaTeX book, you'd be quite correct.

The web site of the book

This book is accompanied by a web site. The address is:

http://www.comp.leeds.ac.uk/tony/book/java/

The site is mirrored here:

http://www.palgrave.com/htpuj/

On the web site you can find:

- All the code that you'll need to complete the exercises.
- An interactive Java reference using the examples from the book.
- All the solutions to the exercises, with some extra details.
- Additional exercises for every chapter.
- All the programs from the book, so that you can download them, try them out, and adapt them.
- Links to free Java compilers, editors, and other development tools.

There are also forms to submit comments and useful links, and much more.

About the authors

Tony Jenkins is a Senior Teaching Fellow in the School of Computing at the University of Leeds. His hobby is teaching introductory programming. He is lucky that his hobby is also his job. He has given many presentations and written many papers about the ways in which programming is taught.

Tony gained his BSc from the University of Leeds in Data Processing (back when computers were *real* computers) in 1988. Five years spent writing programs in what many call the "real world" convinced him that fun was more important than money, and he returned to the University of Leeds in 1993. He has been teaching since then, and teaching introductory programming since 1995. In 2002 the University of Kent at Canterbury saw fit to award Tony an MSc for research into the experience and motivation of students learning to program.

When not at work, Tony can generally be found with Dave and Wallace in the Grove or the Eldon, probably after two hours on the terraces at Headingley, observing the antics of the Tykes or the Rhinos. He has been known to drink beer.

Tony approves of cats, but is allergic to them. He thinks that ducks are alright, but he has never been closely acquainted with one.

Anyone wanting to contact Tony about this book (or just to have a chat) can send email to `tony@tony-jenkins.co.uk`. There are rumours of some sort of web page at `http://www.comp.leeds.ac.uk/tony/`, but `http://www.leeds-camra.com/` is probably a better bet.

Graham Hardman works as a computer support officer in the School of Computing at the University of Leeds, and has done so since graduating from there with a BSc in Computer Science in 2001. During his time as an undergraduate, he acquired the *nom de plume* Mr Gumboot, for reasons lost in the mists of time, and probably best left there.

In the course of his job, Graham writes programs in many languages, including C, Java, Perl and Python. He can usually be found helping to maintain the large number of Linux workstations and servers in the School, but has been known to touch Windows machines on occasion.

Outside the hallowed corridors of Yorkshire academia, Graham maintains an avid interest in an organisation known as Everton, apparently a popular gentlemens' sporting establishment in his home town. He can sometimes be found in the Spellow House on Dane Street enjoying 5.68cl of draught Irish stout with various gentlefolk in royal blue attire. He also chases inflated spherical objects on artificial grass surfaces, is trying (and struggling) to teach himself Greek, and possesses a drumkit and several guitars.

Graham currently lives in Armley, Leeds with his wife Tanya, two guinea pigs named Arthur and Geraldine, a fish named Colin, and several dozen unnamed feral pigeons. He would rather like a dog, but feels that one would get decidedly bored in their 6 m^2 back yard.

Graham can be contacted at `gph@mrgumboot.co.uk`, whether to discuss Java, the 2008 European Capital of Culture, or the Modern Greek verb system.

Acknowledgements

There are, as always, many people that we need to thank for their help in preparing this book. This has indeed been something of an experience.

Thanks are due to all at Palgrave for their support and advice. Tracey Alcock started it going all those years ago (because she *really* wanted a Java book and not a C++ one), and Becky Mashayekh and Anna Faherty have kept us going more recently.

Christine Jopling once again drew the pictures. We only wish there was space for all the ones she drew before we changed the chapter titles. As always, Chris never seemed to mind if we needed just one more picture, or some really subtle and annoying change.

We are very grateful to the anonymous reviewers of the first version of this text. We know that one of you was David Barnes of the University of Kent because of some of the things you (quite correctly) said. The second reviewer remains anonymous, even if we do have our suspicions. And once again our colleague Nick Efford volunteered to go through a more complete version making many useful suggestions, most of which have hopefully found their way into the final version.

Respect and thanks to Mukesh and the team in Chennai for turning the Word files into a book.

Elvis the Duck is a bit of a mystery. Flossy the Sheep (as seen in the C++ book (and indeed as seen on Elvis's wall)) could be explained, but Elvis is a mystery. The best theory is that Elvis is Christine's idea, probably because she can draw ducks. Chris also produced Zoot the Coot. Tony would like to claim some of the credit for Don the Swan and Bruce the Goose.[1] Graham produced Mr Martinmere, because Graham is from that part of the world.

We are also grateful to Walrus Gumboot.

1 Yes. This is a very bad joke to do with Goslings.

This page intentionally left blank


Chapter 0 What this book is
grogalong...didyab
BRUCE'S HUT
JAVA for

Good Evening. Glad you could make it. Pull up a chair. Just let me put some more logs on the fire. Move the papers off the stool. Help yourself to a drink from the cabinet; there's some orange juice at the back, and maybe something stronger somewhere. I'm afraid I've run out of toast and pretzels. Would you like a chocolate digestive?

Ah, you've got a copy of the new book, "How to Program Using Java". A wise choice if I may say so. While you have your drink and digestive just let me explain what all this is about. There are some things that you're going to need to know before we start.

This is Chapter 0 and that is your first lesson. Most books have Chapter 1 as the first but this one has Chapter 0. I'm afraid that computers are like that. You're going to have to get out of the habit of starting to count from 1. Computers have a nasty habit of starting to count from 0 and if we're going to write programs to make a computer do useful things for us we're going to have to get into that habit too.

Before we go any further allow me to explain what this book *is* and more importantly what this book is *not*.

What this book is not

This is not a book about Java. There are plenty of books about Java available and it would have been very foolish of Graham and me to spend our precious leisure time trying to add to their number. This is not a Java reference book, although there is a handy Java reference in the back. It is not a book to read if you are already confident about programming and can program well in some other language. Oh no. This book is for people who can't program. It is for people who don't have much idea of what the whole thing is about. It is perhaps even a book for people who are starting a course in programming and are just a little bit worried about it.

What this book is

This is a book about learning to program. There are not many books about learning to program even if some books about programming languages claim to be about learning to program. These claims are normally wrong. This is a book about learning to program, and the language we will use to write programs as we learn is Java. This means of course that you will also learn some Java, but don't confuse the two things. If you can write programs in Java you can quickly learn to write programs in lots of other similar languages.

Computer programs are written in computer programming languages. Deep down most programming languages are basically the same. They use the same concepts and ideas and much the same constructions. Sometimes the way of achieving something is exactly the same in several languages. If you can learn to program in one language you can normally pick up another language without too much bother. I originally learned to program in a language called BASIC. Then I learned another called Pascal and then something else called C. Eventually I arrived at a development of C called C++, and finally I came to Java, the language in this book. It's learning the first language that's the difficult bit; it's much, much easier after that!

After they've managed the first language a lot of people find that they enjoy learning new ones. Graham and I have a colleague who claims to make a point

of learning a new language every year. There are many programming languages. Some have been designed for a particular purpose and are very good in that special area. Others, like Java, have been designed so that they can be used for almost anything. As you learn more and more programming, and more and more languages, you'll come to be able to pick the right language for the job. But you have to start somewhere, and you're starting with Java.

This is a book about learning to program. It is not a book about a particular programming language. This means that this book is rather unusual. There is not surprisingly a lot in this book about learning to program but not very much just about Java. When you've read this book and when you've done all the exercises you'll be able to get a different book about Java and learn about all the extra fiddly little details that we've missed off as and when you need them. Java certainly has a lot of these, and the last chapter will point you in the right direction. Our job in this book is to put you in a position where you can use one of the other books if you want to learn more Java or another language.

You are learning to program. You are not learning Java. Promise me that you'll remember that.

Who this book is for

This book is mainly for students following a first programming course in Java as part of a further or higher education course. This will probably be in the first year of the course. It is especially for those students who have never done any programming before. It doesn't matter what your main course is, whether it's computing or something totally different. If you've not done any programming before this is the book for you.

It really doesn't matter at all if you've done no programming before. In fact, about half of the students I meet every year before their programming course starts have never done any. If you've done a bit before you might want to let your attention wander in a few places. I'll trust you to spot when these are.

Of course other readers are welcome to join us. If you just want to know something about programming or object-oriented programming in particular you're in the right place. The more the merrier, that's what I say.

Why this book is like this

I've been teaching people to program for a very long time. Every year I teach many students and every year almost all of them succeed. When they find things difficult they often go away to read their book. Sadly their book has almost always been written for people who can already program or for people who find the whole thing very easy or by people who are very good at programming and don't understand what it's like to learn to program. Most of my students do not find the whole thing very easy, and I hope I understand why. This book is for them.

Often my students tell me that they don't understand something or that they're stuck with some programming topic. They tell me that they've been away and read their book but that it didn't seem to help. This doesn't surprise me because I've seen their books. This is the book I wish they'd had.

Without further ado let me explain what is in this book.

What is in this book

After this introduction every chapter[1] in this book has five sections. The sections work like this:

In brief This section introduces you briefly to the topic of the chapter. It tells you what the chapter is all about and tells you what you should understand after reading it. Think of it as an enticing appetiser. Our friends who have done some programming before might want to read this to help them to decide whether they need to read more.

The idea This is the main part of the chapter. It explains the new idea that is being introduced in some detail and illustrates it with sample programs. This is the section you should pay the most attention to. It is the hearty main course.

Examples Most chapters also have some examples. These use the new ideas from the chapter and apply them to a new problem. I'll show you the correct solutions and working programs and explain why they are correct and why they work. This is the dessert and is something to look forward to.

Exercises There are always exercises or tasks for you to try yourself. They are not optional! You should work through these on your own, and don't start looking at the answers until you have. You should linger a while over this. Think of it as the cheese board.

Summary Finally you'll find a brief summary of the chapter. This explains what you should have learned and what you should now be able to do or what you should understand. This is what you should be able to do before going on to the next chapter. The coffee and after-dinner mints.

You should try to read each chapter in one sitting; they're not very long. Hopefully you'll be able to read one on the bus on the way home or something. Then try the exercises later on. Don't be tempted to read the book while sitting at a computer. Read the chapter. Then think about it. And then go to a computer and try the exercises.

This book uses an approach to object-oriented programming called "objects first". We'll start by looking at what "objects" are and how we might recognise them in the real world. Then we'll move on to see how we can implement programs using objects in Java. Some Java books don't deal with objects first; many years of experience have convinced me (and others) that they're wrong.

The chapters

The chapters are short and so there are quite a few of them. Here's what's in them. Read this now and you'll see where we're going.

Chapter 0 What this book is You are here! I want you to understand why this book is like this. You need to know what's going to happen in the rest of the book. Most importantly you need to realise that learning to program and learning Java are not the same thing.

---

1 Alright, I admit it. I mean "almost every". Sometimes in this book I will stray slightly from the exact truth and tell you what you need to know at the moment rather than what's strictly true. I'll talk in footnotes when that happens.

Chapter 1
Programming

The book starts with a bit of background and a quick history lesson. Before you start to learn to program it's important that you understand what a computer program really is. You'll learn that programs are all around us in all sorts of surprising places. Programming has developed over many years. Over this time there have been many different approaches and many different languages. Java is the programming language you'll use. In order that you appreciate why it is as it is we'll also have a brief history of programming and of Java itself.

Chapter 2
The mechanics

You are going to learn to write programs. You are also going to have to learn how to make your computer execute your programs to make them actually do something. You are going to have to know how to find the results that your programs produce.

What precisely you have to do will depend a lot on the computer system that you're going to use; you might be using Microsoft Windows, Unix or even something else. I don't really mind.

This chapter gives you the general idea of how to create and execute a program and will tell you what you need to find out and where to find it. It explains what goes on behind the scenes when you prepare to run one of your programs.

Chapter 3
Before you start

There are some important things that you will need to get hold of before starting to learn to program. This chapter explains what these are and points you in the right direction to find them.

There is also a very quick introduction to a Java programming environment that many new programmers have found meets their needs – *BlueJ*.

As well as physical things (like a computer!) this chapter also goes through some of the problems that some people have when they learn to program and explains what you need to do to avoid them.

Chapter 4
Objects. The building block

Java is an example of something called an *object-oriented* (and not object-*orientated* – let's get that right from the start!) programming language. In fact everything in Java (even the program itself, in a way) is an object. This chapter explains what this means by explaining precisely what an object is and how you might identify one.

Objects have special properties called *attributes* and *methods*. This chapter also explains what these are and how you can choose the correct set for the types of object in your programs.

Chapter 5
A word on analysis and design

Before anyone can write a program someone must decide precisely what the program should do. This involves *analysis* of the problem that the program is going to address and *design* of the solution.

This is really only a book about programming but you will also need to be able to do some basic analysis and design. You will need, for example, to analyse a problem you have been given to solve and then design a solution. This chapter explains what you need to know to do just that.

Chapter 6
A first look

Without further ado, we arrive at some computer programs and some Java. This chapter gives you a first look at some programs in Java, and shows you how the example from the previous chapter might look in a program.

This points you in the right direction for the rest of the book.

Chapter 7
Programming (don't panic!)

Programming is a tricky business. The title of this chapter gives the best advice of all for new programmers – Don't Panic! Programming is a structured activity that requires and even demands a structured, methodical approach.

Many new programmers do indeed panic when faced with a new problem. They make mistakes that an experienced programmer would never make. They make mistakes and work themselves into a hopeless state from which they can never recover. This is why many books on programming don't really help people as they learn.

This chapter describes the process of writing a program and some of the common pitfalls and mistakes that new programmers make. Hopefully after reading it you won't make them! Or at least you'll realise when you do make them so you'll only make them once.

Chapter 8
The basics

This is over a third of the way through the book and this is the first chapter that includes a detailed explanation of any Java! Don't be tempted to skip straight to it though; the chapters before are there for a purpose and contain essential background that you will need to have read and understood.

This chapter introduces the ideas of a *variable*, the most basic component of a program, and of course of the program itself. It shows how variables are used to store and manipulate values held in a program and explains how to display the values held in the variables. After reading this chapter you should be able to write your first Java program.

Chapter 9
Input

To write properly useful Java programs you need to be able to get values from a user. You need a user to *input* these values, usually from the computer's keyboard.

There are a couple of ways to do this and this chapter explains the Java you need to do it. After reading it you'll be able to write some useful Java programs.

Chapter 10
A word on testing

For a program to be truly useful we need to have confidence in the results that it produces. We need to believe that they're accurate and correct. This means that we have to *test* all our programs very thoroughly. Testing is another structured process that requires a plan.

This chapter explains how to build up a series of *test cases* into a *test plan*. This plan will allow you to test your programs so that you can be confident that they work. If you are confident in the results produced by your program your users should be too.

After reading this chapter you should also understand why it is never possible to be completely sure that any program works in every possible case!

Chapter 11
A first class

Objects are implemented in Java as something called *classes*. An object in a Java program is in fact an instance of a Java class.

This chapter introduces a very simple class to show the basic ideas of how this works. After reading this chapter you should be able to write programs that make use of other simple classes, and you should have a basic idea of how to write such classes yourself.

Chapter 12
Classes and objects

This chapter looks in more detail at how classes and objects are defined in Java. There are quite a few fiddly little details that you need to understand!

At the end of this chapter you should be able to write simple Java classes and programs that make use of them. You should be able to identify the classes in a problem area and design and write a program using them to solve the problem.

Chapter 13
Get your hands off my data!

Java classes have a public face and a private face. The public part, called the *interface*, is available to all programs that use the class. The private part, on the other hand, is available only to the class itself. Classes are not unlike people.

This is a very powerful mechanism that allows for *data hiding*. It is one of the key ideas that make Java programs portable between different computer systems and one of the properties of Java that makes it one of today's most popular programming languages.

A well-written Java program written on one computer should be able to run unaltered on another computer system. It should be obvious why this is a good thing.

After reading this chapter you should understand the difference between the public and private parts of a Java class. You should be able to design classes with suitable public and private parts and you should understand why this means that Java programs can be portable and reusable.

Chapter 14
Making things happen. Sometimes

This chapter moves on to deal with the rest of the basics of Java. It describes the two *conditional control statements*. Sometimes there are parts of a program that we want to execute only if some condition is true; a word processor program should print a document only if the print button is pressed, for example.

This chapter explains what a *condition* is and how it can either be *true* or *false*. It explains how to combine single conditions into more complex expressions using Boolean logic. Finally it explains how you can use such expressions to control how your programs behave in certain situations.

After reading this chapter you should be able to write more complex Java programs that can carry out different tasks depending on the user's input and on other values. Your programs will be able to deal sensibly with unexpected input values and you will be able to implement simple menu-based systems.

Chapter 15 Making things happen. Again and again

Sometimes parts of a program must be executed many times. This might be for some number of times that is always known (*determinate*) or for an unknown number of times until some event happens (*indeterminate*).

This is achieved in Java with *program loops*. This chapter describes the different kinds of loops available in Java and explains how to use them.

At this point you should be able to write many useful Java programs. You will have learned most of the basic Java you will need and will hopefully have had a good amount of practice. You will be a Java programmer!

Chapter 16 More methods

The new Java in the previous two chapters will enable us to write some more complex methods in our classes.

This chapter provides some examples of doing that and also fills in some of the final details in how methods are written and used.

Chapter 17 Collections

Often programs must process collections of data. The final part of Java that you need is the ability to store such collections. You might want to write a program to store the details of all the books on Java in a library, for example. This program might need to search for a particular book or display a list of all the books.

This chapter explains two ways to do this in Java using *array lists* and *arrays*. These two ideas will allow you to design and write complex and useful Java programs. You will indeed be a Java programmer!

Chapter 18 A case study

As you get near to the end of the book you have learned many things. This chapter ties them all together with one final big example.

This chapter shows you a problem and explains how to analyse it, design a solution, and then how to write a Java program to solve the problem. This is exactly what Java programmers do and now you should be able to do it too.

Chapter 19 More on testing

With bigger programs you need bigger and better testing if you are to be sure that the programs work. This chapter explains how to test more complicated programs and how to test Java classes so that they can be used elsewhere in other programs and by other programmers.

Chapter 20 Onward!

And this is the end of the book! You won't have seen all the Java that there is (very few people have) but you'll have seen, used, and practised a fair bit. This chapter introduces you to some of the more important and useful aspects of Java that you have not seen and explains how to find out more about them.

Java reference and examples

As you write your programs you will often want quick access to a short summary and some simple examples of how the basic Java commands and ideas work. You will want to check the details of the syntax of some command or you will need to be reminded of some detail.

This is what you need!

Further reading

After reading this book you will probably want to move on to more programming. This section points the way to where you might start and gives you some references both on the World-Wide Web and in books to follow up if something has particularly interested you.

Some languages that you might want to move on to sooner rather than later are C++, Python, and C#. If this is what you want to do, you'll find some handy pointers at the end of this chapter.

The book is rounded off with a handy glossary and index.

The characters

All good books on programming have a cast of lovable characters; it has to be said that in some books they're the best bit. Not wanting this book to be an exception to this fine tradition[2] I too have assembled a suitable cast of suitably lovable characters to guide you on your way through the book.

Here they are.

Elvis

The star of the show. Elvis lives on a wild-fowl reserve with his friends near a nice big pond. He used to be just a typical duck whose day generally consisted of looking out for the odd person who throws bread in the pond and going "quack" a lot.

This was not enough for Elvis, and he failed to find fulfilment in this role. So he has decided to branch out. Computers seem to be quite big at the moment, so Elvis has decided to see what they can offer duck-kind.

Elvis is a duck that is going to learn Java.

Buddy

Buddy is Elvis's best friend. Buddy is always keen to try new things, so he has decided to join Elvis.

Cilla

Cilla makes up the happy band of duck programmers.

Cilla is an especially good friend of Buddy.

2 There are many fine traditions that should be observed by all books on programming. I hope that I haven't missed any; I'll be pointing them out as we go along.

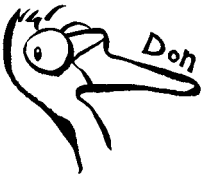
Bruce

Bruce is just a typical goose. He recently arrived at the reserve from Australia, a country where all the geese are skilled Java programmers.

Bruce has agreed to teach Elvis and his friends all he knows.

Zoot

Zoot is a coot. He is not entirely convinced that computer programming has much to offer coot-kind, but he has decided to join in. Just in case.



Don

Don is the only swan at the reserve. He finds that the size of his wings makes typing difficult and has little time for computers.

He sometimes gets in the way.

Mr Martinmere

Mr Martinmere runs the nature reserve where Elvis and the others live.

He knows little about Java, but is not afraid of using the skills of his birds to make the management of his reserve easier.

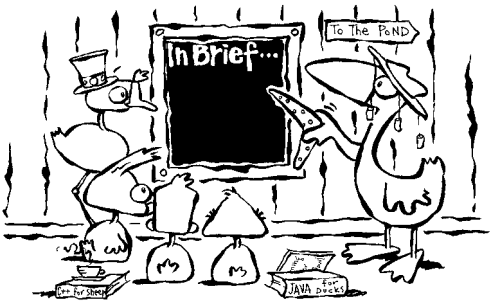
The end of the start

This is the end of the start of the book. Hopefully you now know what's coming up and why it's coming up in that way. It's time to read on.

Right. Put the dog back on the chair and I'll put the fire out. Leave the empty mug on the table. Mind the crumbs.

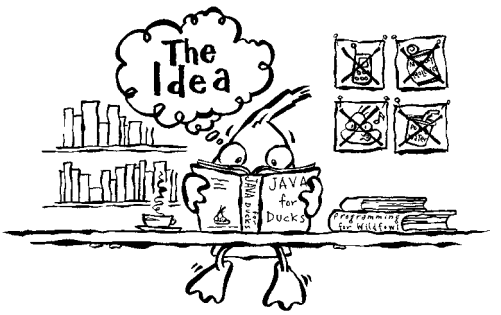
Now, it's on with the show. Remember that we're in this together and that I'm right behind you!

Chapter 1
Programming



In the modern world computer programs are all around us. They are sometimes hidden away in the most unexpected and unlikely places. All of these computer programs have at some time been written by a human programmer using the processes and skills that you are about to learn. This programmer has gone through the stages that you are going to learn about in this book and has probably used a programming language not unlike Java. At some point in the past this programmer learned to write programs. Before you start learning to write programs it is essential that you know what a program is.

After reading this chapter you should understand what a computer program is, and you should understand what programming is all about. You should be able to identify the various programs that run on the computers you use and you should be able to make a reasonable guess at how they were written. You should understand something about Java and why it is a popular programming language. You should know about some of the people who have contributed to the development of programming and to other currently popular programming languages and to Java in particular.



Computer *programs* are everywhere. I am using a computer program to write these words. These words will be processed by many more computer programs before you read them; an electronic mail program will be used to send them to an editor, a backup and compression program will be used to keep a safe copy, and the typesetter will use yet more programs. It is entirely likely that a computer program was involved when you bought this book; there would have been one in the till when you paid, you may have used a computer-based catalogue to find the book you wanted, and you may have paid with a credit card that was authorised by a computer. All these things rely on computer programs, and all these programs have been developed by human computer *programmers*.

While I type these words, my computer is running many other programs. The *operating system* itself (Microsoft Windows as it happens) is a program. I am using yet another program to choose which track to play on a CD; this program is using another program buried deep inside my CD drive to translate the data stored on the CD into sound. The CD itself was probably recorded on a machine that used many computer programs. Many of the instruments played to make the recording also contained computer programs.

Computer programs can be found in devices ranging from mobile telephones through dishwashers to sports cars and fighter aircraft. All of these programs have one thing in common; a human programmer wrote them and went through much the same processes to write each one.

Computers are pretty useless things without programs. On its own a computer is little more than a collection of wires, plastic, and tiny bits of silicon. On its own a computer would make a handy doorstop, and little more. It is only when a program is added to this collection of components that a computer becomes a useful tool. The development of computer programs is a fundamental part of the development of computing.

A program

Here is a simple program. It is not the sort of program that can be used on a computer, but it *is* a program. It is, in fact, a recipe for chocolate cake.[1]

```
Ingredients
  4 oz soft margarine
  4 oz caster sugar
  2 size 3 eggs
  3 oz self-raising flour
  1 oz cocoa powder

Method
  Pre-heat the oven to gas mark 4, 325°F, 160°C.
  Beat the margarine and sugar together until the mixture is
    almost white.
  Beat the eggs one at a time into the mixture.
  Sift the flour together with the cocoa powder and carefully fold
    it into the mixture.
  Divide the mixture into two 6 inch round tins.
  Bake on middle shelf for 20 minutes.
  If desired, cover with melted chocolate.
```

(Source: Home Recipes with Be-Ro Self Raising Flour, *37th Edition, Rank Hovis MacDougall, 1982)*

Although this may appear to have little to do with computer programming, this recipe shares many features with computer programs:

- It follows a format dictated by convention – all recipes look like this, with the ingredients listed first and then the method described step-by-step.
- It is written in a specialised form of language.
- This language has a vocabulary that someone reading it must understand – beat, sift, and fold are all terms that the person making the cake would need to understand.

---

1 You can try it, if you like. It's really rather good.

- After the ingredients have been listed the recipe consists of a sequence of steps that the user must follow in the correct order to achieve the desired result – it would be foolish to put the cake tins in the oven before making the mixture.
- Some steps require the user to make comparisons and take decisions based on what they find – in this case the user must repeat the action "beat" until the mixture is white.
- Some steps require the user to make choices – here the cake is covered in chocolate only if desired.

It would also be possible (but admittedly unlikely) for a cook to follow all the steps of this recipe without any knowledge of what the final result was going to be. That is exactly how a computer interacts with a program; it follows the instructions that it is given without any knowledge or understanding of the result that the programmer is trying to achieve.

A computer program is simply a "recipe", called an *algorithm*, presented to a computer in order to allow it to carry out some task. The recipe is written by a programmer and is expressed in a programming language. The programming language has a vocabulary from which the programmer constructs the instructions that the computer can understand and execute. These instructions form a computer program.

Programming languages

Early "computers" were designed to fulfil a single purpose. The devices generally considered the earliest computers, the Difference Engine and Analytical Engine designed and partially built by Charles Babbage, were designed to carry out the calculation of mathematical tables. This extremely tedious and repetitive task was an obvious candidate for automation, especially as the many errors in the human-generated tables could prove very costly.

Babbage never completed his machines. He was misunderstood and ridiculed during his lifetime and died unknown. One of the few to recognise the potential and importance of his work was Lady Ada Lovelace, now generally recognised as the first computer programmer. Lady Lovelace developed a mathematical notation for representing different ways to "program" Babbage's Analytical Engine. Sadly, as the machine was never built, it was never programmed. Today there is a replica of Babbage's Difference Engine in the Science Museum in London; it is the first exhibit in the gallery tracing the history of computing. Any self-respecting programmer should take a moment when in London to examine the Difference Engine, especially now that admission is free.

The first electronic computers were programmed in what we would now consider very "low-level" ways. The programming involved changing physical components in the computers; levers were set, wires were moved, and switches were changed. This was a slow and error-prone job. This was also the time when the first computer program "bugs" were encountered; these were moths that crawled into the computers, promptly died, and stopped them working[2] properly.

---

2 The moth normally stopped working properly too. There is actually a school of thought that this story about the moths is an "Urban Legend". Why not spend a bit of time looking in to this on the web?

It became apparent that a different way of programming computers was needed. The existing way of working was too close to the machine's way of operating; the programmer was being forced to think at too low a level. This led to the development of the first programming languages expressed in a form that was more convenient to human programmers.

This was an improvement but it was not ideal. Every computer had its own language and programming any computer was still a complex and time-consuming task. A language was needed that would be easier to use and that could be used on any computer. A "high level" language was needed.

This idea had been seen before. At the end of the Second World War Konrad Zuse had developed a high level language that he called Plankalkul. This was never implemented, but it remains as the first high level programming language ever described.

The first language to be implemented and used was FORTRAN, first released in 1957. FORTRAN was designed mainly for mathematical applications and was highly *portable*; FORTRAN programs would run unaltered on many different types of computers. Languages based closely on the original version of FORTRAN are still popular for many mathematical applications today.

Many more languages followed. Some were designed to be truly general-purpose and others were designed for particular types of application. Notable languages include:

Lisp — Lisp was designed specifically for List Processing applications. It is still used today in some areas of artificial intelligence.

Algol — Algol offered programmers the ability to express algorithms in a neat and concise way. In many ways it is the earliest ancestor of Java.

COBOL — COBOL was designed for applications in business. It offered sophisticated features for processing large files of data to produce the sorts of reports that managers needed. Decisions made in the development of COBOL programs in the early years of computing led to most of the panic surrounding the "Y2K Crisis" at the start of this century.

BASIC — As programming languages developed there was a need to train new programmers. BASIC was a language designed for beginners; it included all the features of most other languages and was a fine learning tool. Most of the early home microcomputers in the 1980s provided a version of BASIC as their programming language.

Simula — Simula was a special-purpose language, designed to carry out simulations in applications such as queuing theory. In a way it is also an ancestor of Java as it introduced the concept of an object.

Java

We come now to Java. Java is the latest in a proud family of languages. Its origins can probably be traced back to a language called BCPL. BCPL was developed at Cambridge University in the late 1960s as an alternative to FORTRAN. BCPL was developed by Martin Richards while he was visiting the Massachusetts Institute of Technology (MIT); it became widely used by those working on the new Unix operating system there.

A more direct ancestor can be found in the B language also developed at MIT in the late 1960s and early 1970s. B was designed as a programming language for developing systems on Unix, which was yet another project being developed at MIT.[3] B was developed, extended, and refined and eventually evolved into a new language, imaginatively named C. C was first fully described in "The C Programming Language" by Brian Kernighan and Dennis Ritchie in 1978. C became one of the most popular programming languages, if not the most popular, in the 1980s and 1990s. It is still widely used today.

In the early 1980s an extension to C was proposed by Bjarne Stroustrup; this was C++ which is essentially a development of C with extensions to support something called *object-oriented programming*. Stroustrup was influenced by Simula and wanted to extend C to include object-oriented facilities. C++ offers significant advantages over C for many applications, although even Stroustrup himself has remarked that C is still to be preferred in some areas. The key is choice; C++ can be used as an object-oriented language or it can be used in a more traditional, C-like, way. This flexibility means that C++ became one of the most popular programming languages in a whole range of applications.

Java can in a way be seen as a further development of C++, although it was developed from a "clean slate". The history of C++ and its development means that it is effectively a hybrid language; in some ways it is an object-oriented language while in others it is more traditional. Java is a much more pure object-oriented language.

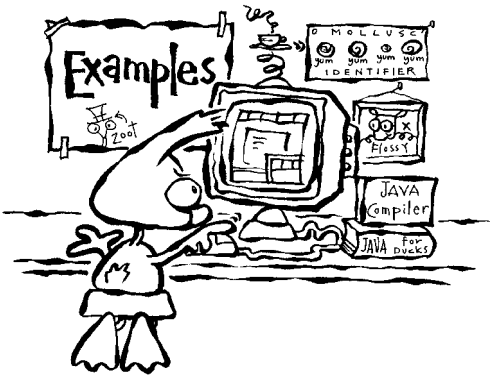
Java was developed by a group led by James Gosling at Sun Microsystems. It started out as part of something called the "Green Project", a research and development project started by Sun to try to look ahead to future trends in computing. The project developed various electronic gadgets aimed at the consumer market. These gadgets needed to be programmed and so a new language, originally called Oak, was developed. Oak is the language that eventually became Java; rumour has is it that the name was changed when the Green Project discovered that there was already a programming language called Oak. The name "Java" has a lot to do with coffee, a very popular drink among computer programmers; the Java logo of a steaming coffee cup can be see on Sun's Java web pages to this day (and versions of it are liberally spread around this book!).

Java is now one of the most popular programming languages around. Its heritage and design means that programs written in Java can run unaltered on all sorts of computers and devices (I have a mobile phone that can run small Java programs!). This design is based around something called the *Java Virtual Machine* (or JVM), which we will meet later on. The JVM is, incidentally, written in C.

The history of programming languages is quite fascinating in its own right. There are pointers to help you find out more in the Further Reading section at the back of the book. You might want to check out some more details of the history of Java, and perhaps even see what James Gosling himself has to say about it. If you understand how a language has developed you can often understand why it works the way it does.

---

3 You may well have met or heard of the most popular modern incarnation of Unix, the free operating system Linux.



There are no real examples in this chapter (I did say that some chapters wouldn't have any!), but here's an illustration of something that you should remember while you learn to program:

Many programming languages are basically the same.

This is simple to illustrate.[4] A first example is something called a *loop*, and in particular a "for" loop. Look at these for loops written in a few programming languages:

```
Pascal   for i := 1 to 10 do
BASIC    for i = 1 to 10
C        for (i = 0; i < 10; i ++)
C++      for (i = 0; i < 10; i ++)
Java     for (i = 0; i < 10; i ++)
```

Take it from me that each of these five has pretty much exactly the same effect in a program. They are examples of the same type of command in five languages designed for quite different purposes. Even if you don't yet understand what a *for* loop achieves, you should be able to see the similarities (yes, the C, C++, and the Java are all exactly the same).

Computer programming languages evolve. When someone designs a new language, they include all the basic features that are found in other languages, and it makes sense to use a way of expressing them that is similar to these other languages; for one thing it makes the new language easier to learn and therefore more likely to be adopted.

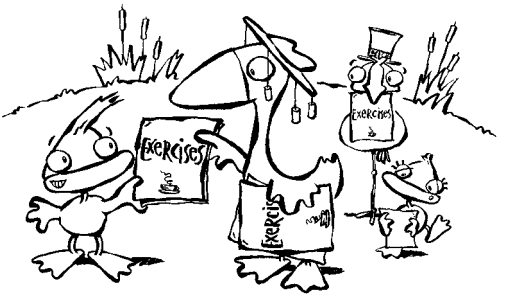
Here's another example. This is a conditional statement (a statement that lets a program make a choice), in this case testing whether some value (called *x*) is less than 10:

```
Pascal   if x < 10 then
Python   if x < 10:
C        if (x < 10)
C++      if (x < 10)
Java     if (x < 10)
```

4 It is also a fine example of some of the things that I'll tell you that are not *strictly* true. It is true that the vast majority of the languages that you are likely to meet for the time being are all the same. These are called *procedural* languages. You might in the future meet *functional* languages such as ML or Lisp which do work in a rather different way.

Once again the C, C++, and Java are the same, which is to be expected as C++ is a development of C (in fact all valid C programs are also valid C++ programs) and Java is based closely on C++. But the examples in the other two languages look very similar, as would examples in many more languages.

Remember this. You are not just learning Java. You are learning to program and you are learning principles that you will be able to apply in many other languages.



1.1 If you have access to the World-Wide Web look up some information about some of the main figures in the development of C, C++, Java, and object-oriented programming. Dennis Ritchie, Brian Kernighan, Bjarne Stroustrup, and James Gosling all have homepages that can be found quickly and easily. There's a list of web search engines at the back of the book.

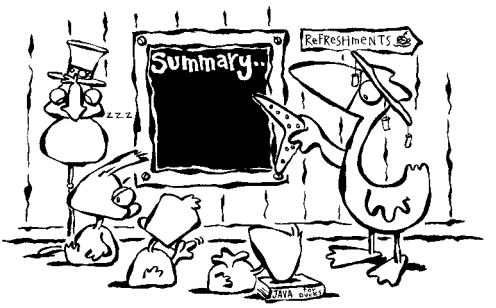
1.2 An important part of your development as a computer programmer is that you know something about other figures whose work has been fundamental to the development of programming. Use the web to find out more about Charles Babbage, Lady Ada Lovelace, Alan Turing, and Grace Hopper.

1.3 In this chapter a recipe for chocolate cake was shown as an example of an everyday thing that has some of the same characteristics as a computer program. List at least two other everyday things that have the same characteristics.

1.4 Look around you know. What things around you have computer programs inside them? Who do you imagine wrote them, and how?

1.5 If you have access to a computer running Microsoft Windows press *Control-Alt-Delete* and click on *Task Manager*. Under the "Applications" tab you'll find a list of the programs running on the computer. See if you can discover what some of them do.

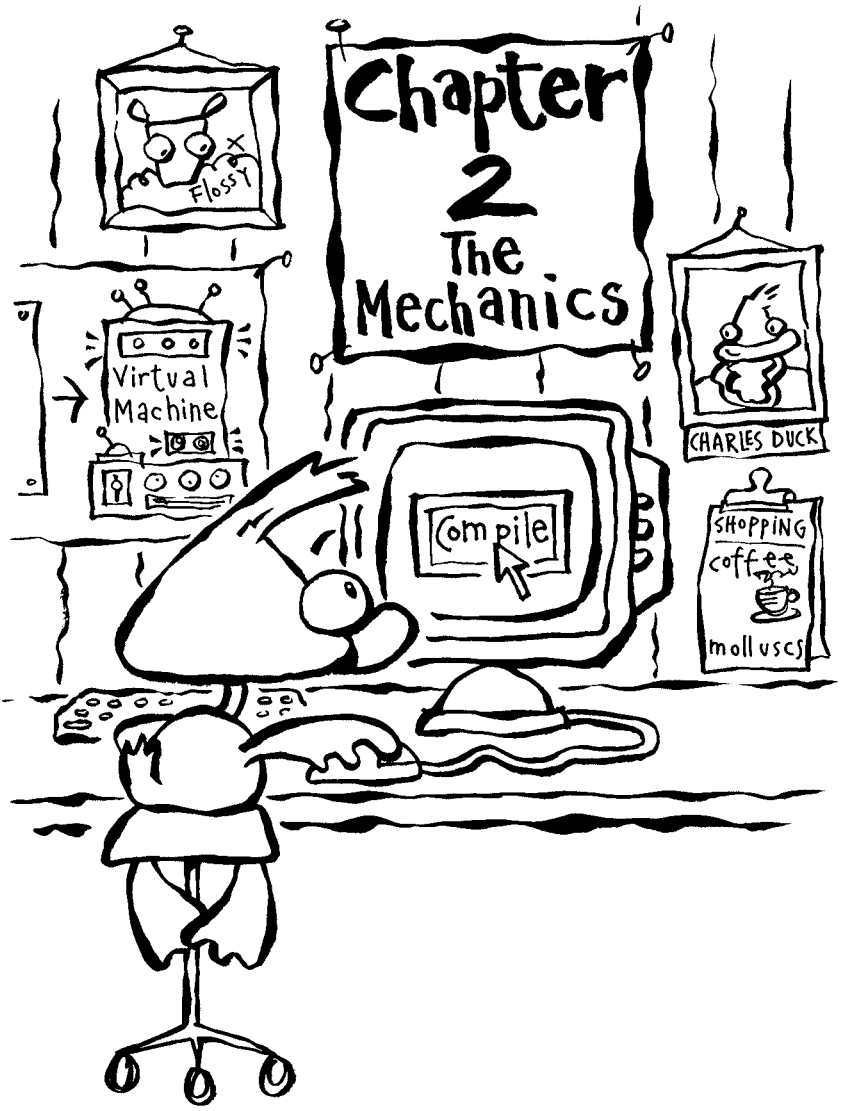
1.6 Alternatively, if you have a Unix system, type *top* at the shell prompt and look at the output. This shows you all the processes (programs) running on the computer. See if you can find out what some of them actually do.

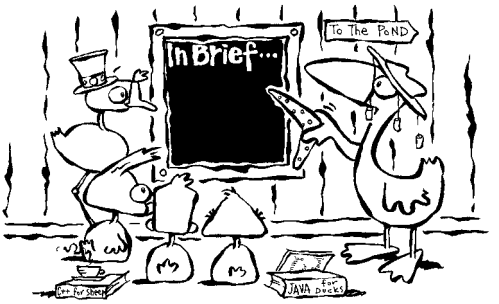


Computers are all around us and so, therefore, are computer programs. Any computer that is carrying out any useful task has somewhere inside it a program. In fact it probably has many programs. All these programs were written by human programmers using a programming language.

As computers have become more sophisticated so have the languages that are used to program them. Early computers were programmed with switches and levers and then by low-level languages. This was so time-consuming and error-prone that higher-level languages were developed. Over the years some of these languages have died out while others have been refined and developed further.

Java is one of the latest languages in this evolution.

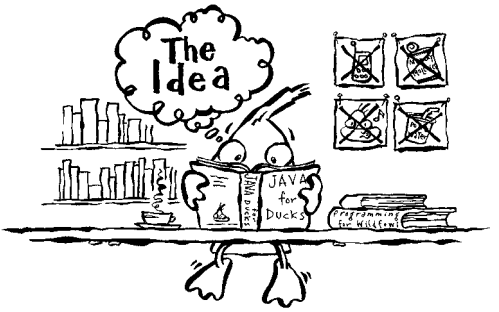

Chapter
2
The
Mechanics
Floss
Virtual
Machine
Compile
CHARLES DUCK
SHOPPING
coffee
molluscs



Humans and computers do not speak the same language. Humans communicate in spoken language, a very complex system of nouns, verbs, adjectives, and so on. Computers on the other hand are much simpler. Their "language" has only two symbols, 0 and 1. Obviously some translation must take place if humans and computers are going to be able to communicate effectively. Something will have to happen if a computer is going to understand what a programmer is asking it to do.

A computer program is a description of a way to solve a problem or carry out a task; we might call it a procedure for solving a problem. It is written in a form that humans can use and understand. This chapter introduces you to the processes that take place when a computer program expressed in this form is translated from this representation into a form that computers can use and understand. Both these forms are actually the same program in the same way that this book translated into Japanese would still be this book; all that has changed is the way it is represented.

After reading this chapter you should understand what a *compiler* is and what the process of *compilation* involves. You should understand how your programs will be linked with the Java *system libraries* to produce versions that can be executed by your computer. You should know what the *Java Virtual Machine* is and you should understand that role it plays in executing your programs. You should also understand how to compile and run your programs in whatever programming environment you are going to use.



The previous chapter explained that a computer programming language is essentially no more than a way for a human programmer to communicate with a computer. Humans find it very hard to express themselves using the 1 and 0

binary language of computers[1] and computers find it equally difficult to understand natural human language. A computer programming language is something of a halfway house between the two; it's something that both humans and computers can understand. In many ways the language is not quite exactly halfway; it's usually rather closer to the human's side than to the computer's. This means that some special software is needed to translate the program into a form that the computer can execute. This translation is a process called *compilation*. This is the process that is explained in this chapter.

But first a word about the computer environment that you're going to use.

The "Local Guide"

Computer systems are all different. This is a book about programming and Java and I don't want all the details of many different computer systems to get in the way. I certainly don't want this book to be useful only to people using a particular type of computer system.

If you read the introduction, you'll remember that I'm going to have to assume that you have a document that I'm going to call the *Local Guide*. This guide might be a manual that came with the system or it might be something specially written for you as part of a course. I'm going to assume that it will explain to you how to create and run programs on the computer system that you're using. I don't mind what that system is just as long as it's capable of running standard Java. What exactly it is doesn't matter at all in the rest of this book.

What I'm going to describe now is the process that happens whenever any program is created and run on any computer. This process is essentially the same on every platform.

Creating and naming a program

A program exists in a file on a computer system. Each file on the system has a unique name,[2] consisting of a memorable name for the file and sometimes an extension:

```
memorable.ext
```

The *file extension* normally indicates what type the file is. You may well be familiar with `.doc` files that contain Microsoft Word documents, for example, or `.jpg` files that contain images. The extension for a Java source file must be `.java` (hence these are often simply referred to as *.java files*), and using any other extension for a Java program is likely to produce a compiler error.

These are possible names for Java programs:

```
Program.java
Ducks.java
Elvis.java
```

The extension should never be left off. Doing so will probably cause errors. Ideally the name of the file should give some indication of the purpose of the

1 Although this is precisely what the users of early computers had to do. A switch that was on was a 1, and one that was off was a 0, for example.

2 Well, unique in its own directory or folder.

program. We will see later on that the name must also match with part of the contents of the program.

Any Java program file is created with a simple *text editor* or sometimes in a more sophisticated *integrated development environment*. Common text editors are `vi` on Unix systems and `PFE` or `Notepad` on Microsoft Windows systems. Many Java systems, especially those designed for use on Microsoft Windows, also provide an *integrated development environment*, normally referred to as an IDE.

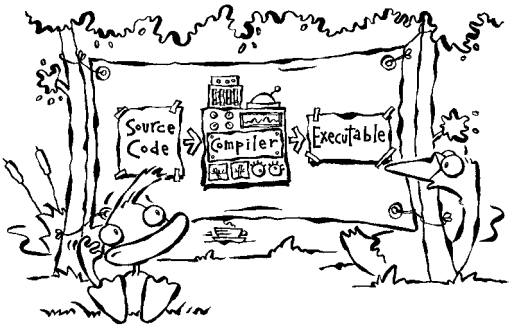
The program prepared in this way, whether with a text editor or with an IDE, is called the *source code* of the program. It is just a file containing plain text; the computer has no idea at this stage of how to carry out the instructions in the program. It is in fact only the name of the file that gives a clue that the file contains a Java program.

It is worth spending some time becoming reasonably proficient in using your text editor or IDE. It's normally possible to do most editing tasks if you learn just a few commands in an editor but many editors also have many much more powerful facilities and tools hidden away. You will spend a lot of time editing files as you work on your Java programs so it's more than worthwhile getting used to using the editor. Hopefully the *Local Guide* gives you plenty of information.

Compilation

When the programmer has created all the source code and is happy with the program it is time for the program to be translated into a form that the computer can actually run. This is a process called *compilation*. Compilation involves the use of another program called a *compiler*.[3] The final result, a version of the program that can be run by the computer, is called an *executable* file. The executable file is written completely in a form that can only be understood by the computer; it is totally impenetrable to most humans.

This gives a process that can be represented like this:



The actual process that a programmer goes through to compile a program is different on different computer systems. There are Java systems available for a great many computer systems and the compilation process used by every one is different. Sometimes compilation involves typing a command at a prompt and other times it is simply a case of clicking a button.

---

3 Here's a poser for you. Compilers are computer programs. They are written in computer programming languages. So, who wrote the first compiler and what did they use? To make it worse I'll tell you that most C compilers are written in C.

On the Unix system used to write the programs in this book the compiler is simply run from a command prompt and is called `javac`, so to compile a program stored in the file `MyProg.java` a programmer would type:[4]

```
tetley% javac MyProg.java
```

If all is well this compilation would usually[5] produce another file called `MyProg.class`. This is a version of the program that can actually be executed; it is effectively the executable version of the program. It contains a compiled form of the `.java` file, known as *bytecode*, which can be run using a Java interpreter on any platform. Often the terms *.class file* and bytecode are used interchangeably.

Once again you'll need to consult your *Local Guide* to find out how to compile programs on the system you'll be using.

Compilation errors

Sometimes a program fails to compile because of some error that the programmer has made. The error means that the compiler cannot process the source code to produce an executable file. Faced with this situation the compiler will do its best to tell the programmer what the error probably is (or at least where in the program the error seems to be) and the programmer must then find and correct the mistake.

The way in which you will find and correct errors in your programs depends again on the system you are using. My Unix system simply tells me what the error is and which line in my program the error is on and leaves me to use the text editor to correct the error. More sophisticated systems will present the programmer with the part of the source code producing the error with the offending section highlighted in some way. Again you should check your *Local Guide*.

New programmers often have terrible problems finding errors in programs. The secret in finding them is to remember two things. First you should always be systematic; correct one error and then try to compile the program again. A small mistake in the program, even one single character in the wrong place, can cause a whole host of different, seemingly unconnected, errors. It is always a good plan to make the error that you correct first the one that the compiler reported first; a small error at the top of a program can have many knock-on effects later on. Second, remember that compilers are not especially clever and get confused easily; what they are reporting is often the first part of the program that they have failed to understand. If the line where the error is reported looks fine, always look at the lines immediately *above* the point in the program where the compiler says the error is.

Running the program

After the program has been compiled into an executable form it is ready to be run. As usual the precise way in which programs are run is different on

---

4 The `tetley%` in this is the prompt on my computer, and isn't part of what I type to compile a program. My computer is named after the famous Joshua.

5 Assuming that `MyProg.java` contained a public class called `MyProg`, that is. Don't worry about this for now, as all will become clear later on.

different computer systems. On a Microsoft Windows system, for example, it is normally enough to double-click the mouse on the executable's icon or the development environment may run the program automatically.

In all cases, though, running a program requires the use of the *Java Virtual Machine*. This is another program that runs the compiled version (the bytecode) of the program on the computer. One of the key advantages of Java is its portability; the compiled version of a program will run with any JVM, running on any sort of computer system.

On the Unix system used for this book the command to execute the program is simply `java`, so to execute the code contained in the file `MyProg.class`, the command is:

```
tetley% java MyProg
```

It's time to consult the *Local Guide* to check how to do it on your system.

Other useful things

When working on a program it is often very useful to be able to print your source code out. This gives you something that you can scribble all over as you try and work out why your program isn't working as you hope and expect.[6] You'll have to consult the *Local Guide* to find out how to print your programs and to find out where the printers are.

There may be other tools available to you to help you as you learn to program. There may be a special program called a *debugger*; this is a tool that lets you step through a program a line at a time to help you find and correct problems. There may be a program that helps you format your source code neatly. Find out what's available and make sure you can use it.

A closer look

The description of compilation in this chapter is a little simplistic and it will help if you have a slightly better knowledge of what's actually going on when you compile a program. A full discussion of how compilers work is way beyond the scope of this book and is actually way more than most programmers need to know, but a little more detail will help you when you start looking for errors in your programs.

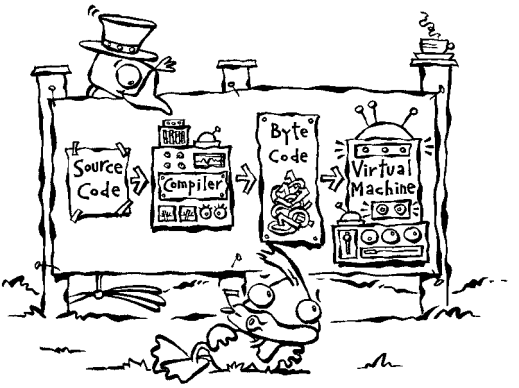
You have seen that there is an intermediate stage between the source code and the executable version of the program. This stage is called *bytecode*; it is the first thing created by the compiler from your source code. You will be able to see the files produced by this first stage of the process; the extension they have is always `.class`.

The bytecode is then *executed* (by another program called the JVM) which can access other program code from various libraries on the system, known as *packages*. These packages contain the code needed for the basic Java functions (so that the program knows how to display characters on the screen, for example). All the code, both from your program and from any packages used, is loaded together when the executable is run.

---

6 Or indeed something that you can show to people to impress them.

So the overall process looks like this:

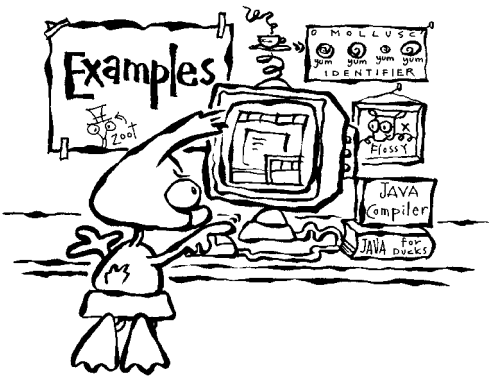


Sometimes the whole compilation process will fail at this final stage. This might be because the necessary packages have not been imported or because the code contained in a package has not been used correctly. What has gone wrong should hopefully be obvious from the error messages generated.

From source code to executable

As you write and test your programs it will help if you remember to think about what it is that you are creating. You are creating a "recipe" for solving a problem and you are using a language that is meaningful to humans.[7] Before this can be run on a computer it must be translated into a form that the computer can execute. The *source code* of the program must be translated into an *executable* version of the program.

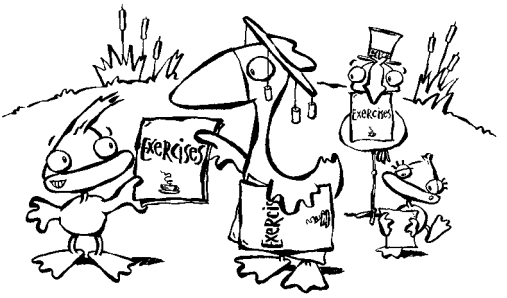
This translation is a process called *compilation*. Compilation is actually a two-stage process. First the source code is translated into bytecode and then this bytecode is executed by the JVM which can make use of various system libraries. The result of this process is the final version of the program, something that is very different in the way it looks to the source code that you have written!



There are no real examples to work through this time. You need to check your *Local Guide* and become familiar with the systems that you'll be using.

---

7 Well, to some humans.



There are no exercises this time either but there are some things I need you to do. Get hold of your *Local Guide* and make sure you find out answers to these questions.

2.1 Does it matter if the name of your `.java` file is in upper or lower case (or a mixture)?

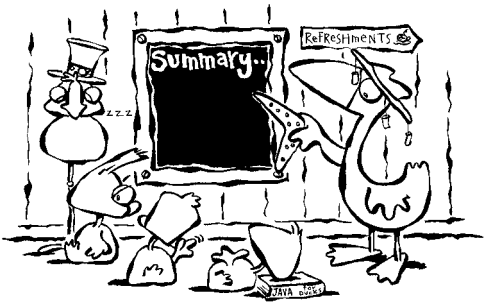
2.2 Which text editor will you be using to create your programs? Or alternatively, will you use an IDE? How do you start it? What are the basic commands you need to edit a file? Can you get a copy to install on your own computer at home?

2.3 How do you compile a program? Where does your system put the bytecode when it has been compiled? How do you use the JVM on your system?

2.4 How does your compiler present error messages to you?

2.5 Where does the standard output from your programs go – to a window, or a log file, or elsewhere?

2.6 How do you print hard copies of your source code? Do you know where the nearest convenient printer is?



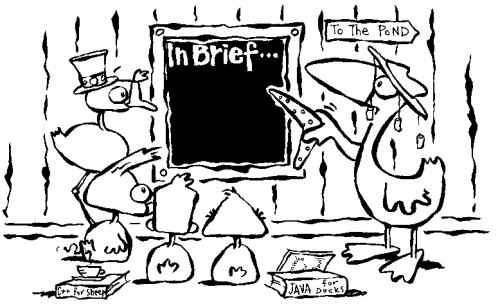
There are two steps in creating a program that can be run on a computer. First the source code must be created in a file; the name of this file will be required to have an extension that indicates that it contains a Java program. When the program has been created it is compiled to produce bytecode; it is this version that can be run on the computer, using the JVM. This second part of the process involves the use of the various Java system libraries to produce a complete working executable version of the program.

There are several skills that you need to master in order to create and run your programs. You need to know how to create files and you need to know how to edit them. You must be able to compile your programs and correct the

errors that the compiler finds. Finally you must know how to run the executable and how to find the output. Unfortunately these skills are different depending on different computer systems; your *Local Guide* will give you the details for your system.

To do any of these things you need to know how to write a simple Java program. So it's nearly time to get on with it. But first there are some things you need to know about learning to program …

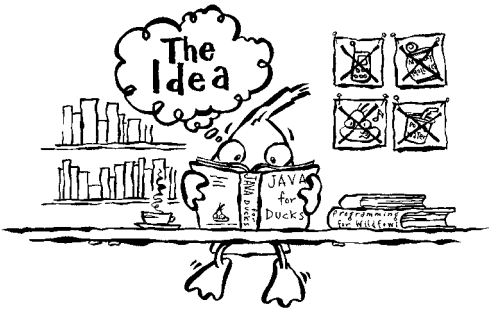

Chapter 3
Before
You start
JAVA KIT
for DUCKS



Learning to program is a complicated business.

Before you start reading the chapters in this book that describe the features of Java, and certainly before you start to do any actual programming, you're going to need to get hold of various things that will come in useful.

After reading this chapter, you should be sure that you have these things. You should have spent a little bit of time experimenting with them, and you should have started to learn the skills that you'll come to need as you learn to program using Java.



Before you start any Java I want to explain to you what learning to program is all about. You need to know what it is that you've taken on and you need to understand something about the skills that you're going to have to call on as you learn to program.

I'm going to start with a quote that could well sum up what you've taken on.

You have chosen a dangerous and challenging path, Adventurer.

A long time ago I used to play a computer game. It was an adventure, and it just used plain text. These were the days when computer games involved nothing more sophisticated than typing instructions to the character you were supposed to be playing. There were no graphics in those days, and certainly no Lara Croft. The computer didn't even have a mouse and we got very excited if a program used some sort of colour beyond different shades of grey. This game involved all the usual things that you'd find in games like this. You'd move around mazes of corridors, battle hideous demons, give small pots of gold to passing dwarves, and fall down the occasional bottomless pit. You get the idea. Much the same happens in many games today only with better graphics.

I still remember how this game started. After it had loaded (something that in those days took the better part of half an hour) the screen cleared and this message appeared:

```
You have chosen a dangerous and challenging path, Adventurer.
You are at a crossroads. Paths lead North, South, East, and
West. There is a sword and a scroll here.
What next?
```

You can guess what happened next. I would dutifully pick up the sword and scroll. I would read the scroll and ponder the pretty useless clues that were on it. Then I quickly rushed off in a random direction. I very soon met the fire-breathing dragon that didn't seem to like me very much, said the wrong thing to it (hitting it with the sword never seemed to help), and was promptly eaten alive or burned to a crisp or both. Then it was back to the start and I started the whole process again.

I never finished this game. I never did work out which dwarf desperately wanted the bucket with the hole in that I found behind the old oak tree. The stone tablet bearing mysterious runes remained a totally useless rock with some odd scribbles on it as far as I was concerned. My friend finished it though. He spent hours on the computer going east, west, down tunnels, bargaining with trees that talked, and many equally strange things. Eventually he found whatever it was he was supposed to be looking for and won the game. He succeeded and I did not.

You are reading a book about learning to program. This shows that you too have chosen a challenging path. I have to admit that it's not very likely that you'll meet a fire-breathing dragon, a talkative tree, or a dwarf with a bucket fixation (all good metaphors break down eventually) but you're certainly in for an interesting and challenging time.

Let no one convince you that this will be easy. Learning to program isn't easy, and anyone who tells you that it is easy is just plain wrong. At the same time, learning to program is not impossible. It can't be, or there wouldn't be any programmers! There's nothing special about people who can program. As you read the rest of this book just remember this. I can program reasonably well and I'm not especially clever. If I can do it so can you!

Let's think about why this is.

Programming is like driving

Imagine this. Suppose this was a book about learning to drive a car, and that you'd never driven a car before. I would write chapters about using the brakes, steering, changing gear, and so on. Imagine that you'd read the book from cover to cover. You might even have memorised parts of it. You might even have passed a test on the contents. Now imagine that I put you inside a car in London and told you to drive to Liverpool. I (and you!) would be very surprised if you or the car arrived in one piece. I hope we agree that no one would seriously consider learning to drive a car like this.

People learn to drive a car by practising. They spend a lot of time sitting next to an experienced driver who tells them what to do and helps them out occasionally when things seem to be going wrong. They drive around and practise the skills they need to develop and perfect. When they've developed these skills sufficiently they take their driving test and hopefully they pass. Sometimes they

don't pass and they need to practise some more. Some people find learning to drive harder than others.

Learning to program is very like learning to drive a car. You are learning a skill. You will need help from people who already have the skill; hopefully, one of those people will be me, and another will be Graham! Above all you will need to practise. There's no way we can teach you to program if you just read this book, but we can teach you if you read the book, then try out the exercises, and then go and practise some more. With any luck you also have some more people who are going to help you. Whatever you do, don't be afraid to use them!

But there's more. Some people pass their driving test the first time. Others have to take it many times, but most people pass eventually. There are very few people indeed who could never pass a driving test if they really put their minds to it and practised. Programming is just like that. Some people take to it straight away, and they're the lucky ones. Most people take a bit longer, but anyone can get there in the end. I've never met anyone that couldn't learn to program. Not if they tried and if especially not if they refused to give up.

If I had taken the same attitude to learning to drive as I took to my adventure game all those years ago I would never have passed the driving test. The game was difficult so I lost interest and gave up. Then again, looking back I probably believe that I could have completed it if I'd really wanted to and if I'd carried on. Learning to program is like that too; if you give up when you find something difficult you'll never succeed and learn. But if you persevere and if you're determined to succeed you will do. I've seen many people struggling as they learned to program but the only people I've seen fail to learn to program are those that gave up. Don't be like them!

Now, don't take this the wrong way. The last thing I want you to do now is throw this book down and set off to do something different. You can learn to program, and you will learn to program. You just need to remember that things will probably get a little difficult at times and when this happens you'll have to sort things out. Giving up should not be an option! Nor should running away and hiding.

Now let's check through what it is that you're going to need to make sure that you succeed. These will be, if you like, the controls that you'll use so you'll need to be familiar and comfortable with them.

What you need

Before you start looking at some programs and some Java we need to check that you're prepared. There are some things you're going to need so let's quickly run through them.

You will need a book or two. You have one in your hand now but you also need something that describes how your local computer system works. I've called this your *Local Guide*, and we've already looked at some of the things it needs to contain. If you're learning to program as part of a course you'll probably also have something else that contains the exercises you're going to complete as part of the course. You'll also need a reference to remind you of the syntax of Java as you program; you can find just such a handy reference in the back of this book. This doesn't cover all of Java by any means, but it covers all the Java that you'll find in this book so it should do for now.

If you have access to the World-Wide Web you can also find plenty of Java documentation on Sun's Java site: `http://java.sun.com/`.

This is a fine source of information because it's constantly updated with all the latest changes and additions to the language. The documentation for the Java *Application Programming Interface* (API) is a couple of clicks away from the main page, and provides you with all the information you need to use all the features of Java. Much of this documentation is, of course, way beyond the scope of the Java you're going to see and use for some time yet, but all you need is there and using this fine resource is a useful habit to get into from the start.

You obviously also need a computer to run all this fine Java that you are going to create. You need something that will run your programs (and let you create them, of course) and will show you the output that they produce. You need a JVM. For writing and executing the Java that you will meet in this book just about any sort of computer will do; you don't need anything especially powerful. A printer so that you can print out your programs and draw all over them would help, but it's not vital.

Your computer needs to have a Java compiler installed; you learned what one of these was in the previous chapter. You need to know how to get the compiler to compile your program to produce the bytecode that the JVM needs, and you need to know how to get the JVM to run this compiled version. There are a great many Java systems and compilers available all of which are very different to use. As I said in the last chapter I'm going to have to assume that you have a *Local Guide* that explains how to use your compiler and the rest of your system.

You will also need a text editor to create the files that will contain your programs. You might be using an integrated system, so this might be part of the whole system that also includes your compiler or it might be a totally separate thing. You will need to know how to use your editor to create your programs and how to get them from the editor to the compiler and into the JVM. Once more, there are lots of editors that you might have available so I'll have to assume that your *Local Guide* explains all the details of that.

If you are missing a compiler or an editor there are plenty of places where you can get them without spending any money. You can check the Further Reading section at the back of this book for a list of web sites where you can download some pretty good free compilers and editors and for a way to get hold of them if you don't have access to the Internet.

If you don't have a Java programming environment at the moment, one that you might want to take a look at is called BlueJ. This is available as a free download from `http://www.bluej.org/`. BlueJ is a Java environment designed specifically for people who are learning to program. The download comes with all the things you need to write and compile programs together with a fine set of examples. It's available for most types of computers, and all you need to know about it is included in the download.

As well as all the technical stuff you will also need access to an expert or ideally to more than one! This is someone who can already program and who is ready and willing to help you. If things get tough this is the person you should be seeking out. There will be times when you just can't work out why your program is producing the wrong answers and you need someone who can help you work out what's wrong. You might not need help very often but most new programmers will need some help and advice at some point while they learn. Get yourself an expert or two and make sure you know how to get hold of them. If you can't find one in person, there are many excellent question-and-answer sites on the web (including one on the main Java site), and these are a fine, if slightly less interactive, alternative.

Above all you need a willingness to learn and you need determination. You need to be sure that you want to do this, and you need to be determined to see it through.

If you've got all these things then read on!

The Console class

Some things in Java are complicated. Unfortunately one of these is getting a user to enter values, something that is obviously rather fundamental to most programs. The program statements needed to do this can throw up all sorts of errors that need to be dealt with, and this can get in the way of new programmers as they learn. The code to do input in Java is, to be frank, horrible.

To lend a helping hand, we've[1] developed a class that reads a variety of values from the keyboard. You can find the listing of the `Console` class in the back of the book or on the web site. The web site is obviously a better bet as it will save you typing and hopefully prevent errors.

Before you can use any of the programs after about Chapter 8 you'll need to get hold of this class and compile it. The programs will assume one of two things. One option is that you have created a *directory* (or *folder*) called `htpuj` below the directory containing your program, and that this directory contains the compiled version of the `Console` class produced in a way similar to:

```
tetley% javac Console.java
```

The second (slightly more complicated) option is that a *jar archive* containing the `htpuj` package is listed in your *classpath*. A jar archive containing the `htpuj` package is available from the website accompanying this book; its inclusion into your classpath should be explained in your *Local Guide*.

You don't need to worry too much about this now, just be prepared for when we start using it later on!

Getting help

If learning to program is difficult then learning to learn to program is equally tricky! The most important thing is that you need to know when you should get help from one of your experts; get help too soon and you'll end up not learning anything, but leave it too late and you run the risk of not finishing your program in time. Knowing when you should get the help you need is a tricky business.

At some point in the not too distant future you'll be sitting in front of a computer trying to write a program and you'll find that you're "stuck". It happens to all of us.[2] You might have some sort of bizarre error message that you just can't work out, your program might be insisting on producing the wrong results, or you might not have the slightest idea how to complete the next part of whatever task you've been set. Whatever the problem is, you need to do something about it.

---

1 The "we" is definitely a "Graham".

2 It happens to me. When it happens I wander down the room to my friend Nick's office and get him to look at my program. It's amazing how quickly he can spot what I was doing wrong.

Staying in front of the computer typing random symbols or copying random sections of programs from your books for hours on end is unlikely to work. It just might (after all an infinite number of monkeys typing on an infinite number of typewriters would eventually produce this book![3]) but it's not very likely. You need to do something more positive. I often find that the best plan is to walk away from the computer and do something totally different for a while. You would be surprised how often the solution can suddenly pop into your head when you least expect it as you stand in the queue for a coffee! But if it doesn't pop you need to do something else.

There are many things you could try. You could try re-reading the relevant chapters of this book. If you've already done that, or if a re-reading doesn't seem to help, the collection of examples at the end of the chapters might help. You could look at some programs that do something similar to what you want to achieve and try to adapt them.

If you don't get anywhere after this go and talk to one of your experts. They should be able to tell you what's wrong and they'll probably be more than happy to do so. At the same time you should resist the temptation to let your expert tell you the answer or write your program for you. If you let them do that you'll learn nothing.

The golden rule for getting help when you're learning to program is:

Try and work it out yourself first but don't be shy about going to an expert.

Always remember that the experts all learned to program for the first time once. The advantage they have over you is just that they learned a while ago and that they've had a lot of practice since then. They've seen all the errors before, and they can remember what it's like for you. Never ever believe that they're more intelligent than you; they've just got a skill that you haven't (yet!). Remember that the only *real* reason that they can do this and you can't is that they're (probably) rather older than you.

Don't panic!

I'm sorry if some of this sounds very negative and off-putting. Please don't take it the wrong way.

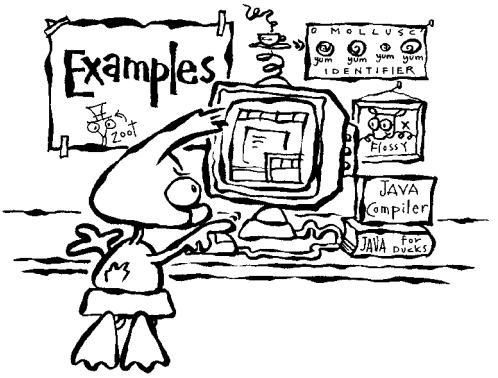
I want you to understand that learning to program is difficult and that everyone who does it gets stuck at some point. There is no shame at all in getting stuck; it happens to all of us and it happens to all of us all the time.

Of course you might be one of those people who finds programming very easy and wonders what all the fuss is about. You might be a "natural" programmer. If you are then I'm sorry for having worried you unnecessarily in this chapter. But I hope you'll do me a favour. Go and find someone who's finding this really difficult and give them a helping hand. Don't finish their program for them and don't tell them the answer; if you did that they wouldn't learn anything. Point them in the right direction, explain an error message to them, or give them a clue how to finish their program. Remember that programming is a social thing. Programmers work as a team and help each

3 Actually, this is not true. A university recently did research and discovered that the monkeys would just tend to type the letter S with the occasional W. They do some wonderful stuff in universities.

other out. You'll probably enjoy lending a hand and the person you help will be very grateful. Thanks.

Now it's time to start on some programming. You'll get the chance to find out whether or not you're a natural programmer!



Since you haven't done any programming yet there still aren't any examples as such in this chapter. While we're here though let me just tell you how I learned to program.

A long time ago, when home computers were a very new thing, you could buy magazines that contained nothing more than listings of programs. You could type these into your computer (they were almost always games and were usually written in a dialect of a language called BASIC) and sometimes they even worked. I learned to program by typing these programs into my computer.[4] Of course, at first I understood practically nothing of what I was typing. For example, I remember looking very hard at this line:

```
50 LET X = X + 1
```

and wondering how this could possibly be right. I mean, *X* can't possibly equal *X + 1*, can it? In time I came to understand parts of the programs I was typing and even that one line became less of a mystery. If you can already see what it does you're doing better than I did! Eventually I could write my own programs, at first mostly by adapting those I had seen before.

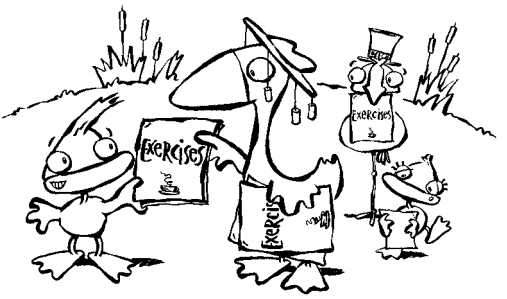
Later I took a more formal programming course. There were some things in it that I found difficult and some others that came much more easily. I got by well enough, by going to the lectures and practising in the labs. There were still some things that remained pretty mysterious and that I never properly understood, though.

I *really* learned to program when I finally got a job as a programmer. I worked eight or more hours a day doing nothing but programming. I worked with programs (this time written in a language called C) that controlled massive test machines in a truck engine factory; if I made a mistake the machines could have exploded and done someone a nasty injury. Sometimes I did it while lying in

4 I lie. It wasn't my computer. It was my friend Mark's computer, and we typed them in together. Programming really is a social thing, you see?

puddles of diesel (which is nasty stuff). That was when I finally became a good programmer.

I'm still learning now. I learn new ways of doing things or new features of languages. No programmer ever stops learning. I have, in fact, learned Java (aided and abetted by my friends Graham and Nick) just so that I can write this book.



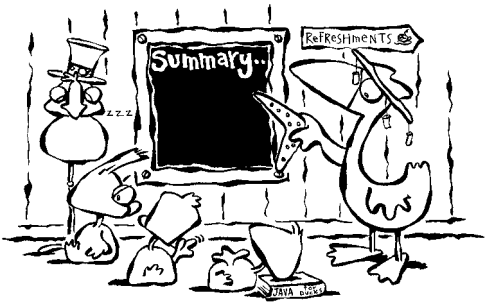
3.1 You are about to start to learn to program. Have a quick look through the rest of this book and through your *Local Guide*. Plan how you are going to spend your time, and write down your plan. Are you going to read a chapter a week or one a day? How much time can you spend on practice? If you're following a lecture course, how are the lectures structured? What topics will you be covering, and when?

3.2 If you have the chance, talk to someone who can program. Find out how they learned and ask them what they remember as being especially difficult or easy. Why not ask your lecturer?

3.3 Take a look at the BlueJ Java environment; you can download it for free from `http://www.bluej.org/`. Have a play with some of the examples that come with the system, and decide whether you want to use it.

3.4 Have a closer look at some of the early BlueJ examples. Take a look at the Java code that lurks behind the diagrams. Is there anything there that you can understand? What have the authors put in the programs to help you?

Finally, take a look at the list of the things you're going to need. Make sure that you've got them all and that you know how to use your editor and compiler. I can't help you with this but hopefully your *Local Guide* can. I'm going to have to assume that you've done it!



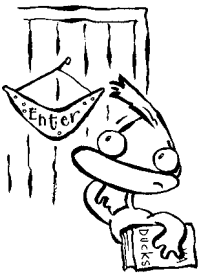
Learning to program is difficult but it is not impossible. You're going to have to do a fair bit of work; in particular, you're going to have to do a lot of practice.

A long time ago, when I was learning to program at university, the author of the book we used[5] (a wise man called Doug Cooper) wrote in the preface that:

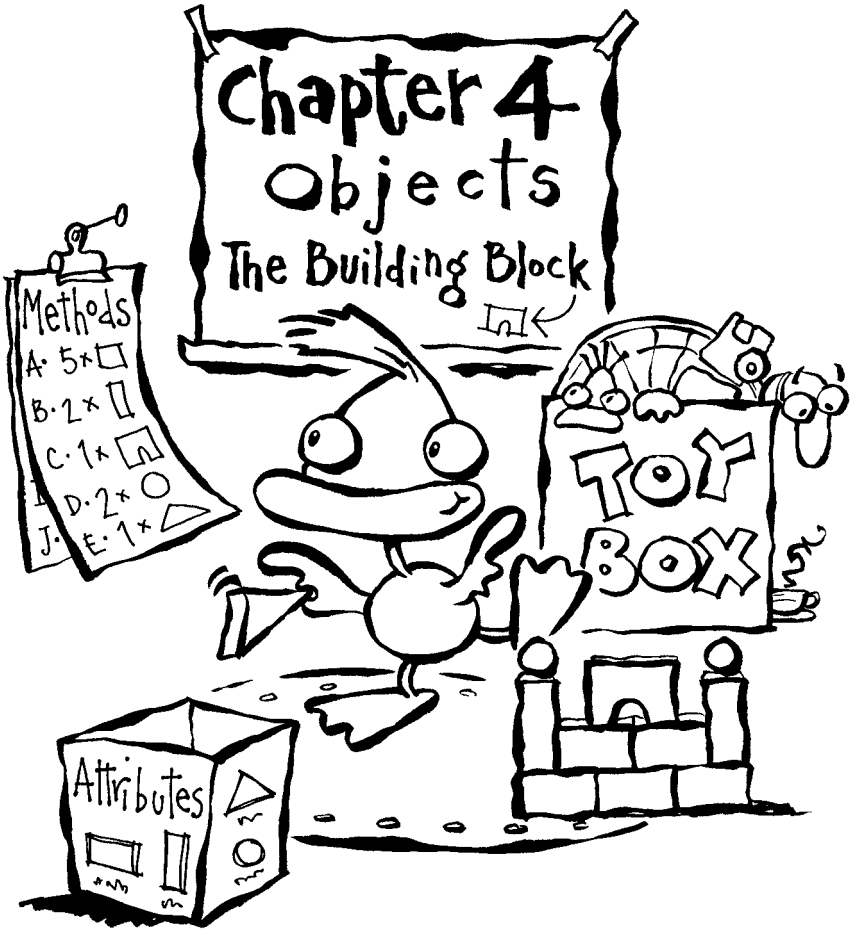
> *Now, when I lecture I encourage any student who isn't so confident to make a smart friend, and to stick by her side for the term. After all, that's how I survived my own first programming class. When I write, I try to be that friend …*

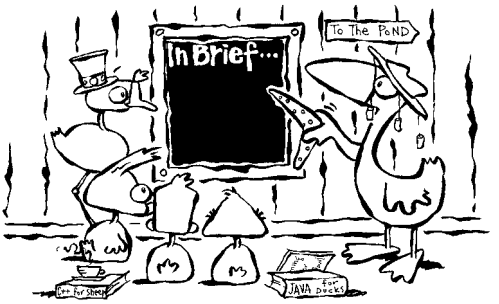
I can't think of any better final advice to give you now than that. That book got me through my first programming course too and I still use it now, even if it is a bit battered. Let's hope this one does the same for you. If you can make a smart friend that's even better! If you can't find a smart friend find a friend who's about as smart as you.

Now that you understand what learning to program is all about and you've got all the stuff you need let's go and learn to program!



5 That book was called *Oh! Pascal!* and was, as you might guess, about the Pascal programming language. Pascal isn't as popular as it used to be, but if you ever need to learn it, this is the book you need.


Chapter 4
Objects
The Building Block
Methods
A· 5×
B· 2×
C· 1×
D· 2×
E· 1×
TOY BOX
Attributes

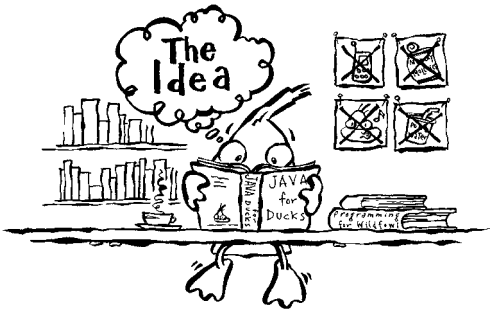


You already know that Java is an *object-oriented* programming language. Many other currently popular programming languages including C++, Python, and C# also use this object-oriented approach. This means, not surprisingly, that programming in Java (and these other languages) is based around the use of *objects*; objects are the basic building block of any object-oriented computer program. Java takes the idea further than some other object-oriented languages to the extent that almost everything in a Java program is an object; even the Java program itself looks rather like an object (even if it isn't actually one).

One of the key advantages claimed for object-oriented programming is that the programs developed to implement an object required for use in one problem can be used again in another problem; the programs to implement the object are *portable*. This potential for *code re-use* is one of the reasons why object-oriented programming is currently so popular; code re-use has the potential to save a lot of time and therefore money.

This chapter introduces you to the idea of an object. After reading this chapter you should understand that objects are the basic component of any object-oriented computer program and you should also understand what exactly object-oriented means. You should understand how an object is characterised by its *attributes* and its *methods*, and you should be able to suggest likely attributes and methods for objects that might be found in some everyday applications.

There is very little in this chapter that is specific to Java. The principles of objects, attributes, and methods are the same in any object-oriented language, and even in some database systems that use a similar, object-oriented approach. In fact, if you've done any work with databases you'll probably see some clear overlaps between what you've learned there and this chapter.



Before starting to write object-oriented computer programs you obviously need to know what exactly an *object* is in this context. Informally an object is

something of interest in a real-world problem. When a computer program is written to solve this problem an object becomes something that is processed or manipulated by the program. The object has values of interest that are processed to produce the required results, so it would probably be more accurate to say that the program manipulates or processes these values rather than the object itself. Objects are sometimes called "entities", especially in the area of database design.[1]

Objects have values that describe them, called *attributes*. An attribute is simply some property that characterises an object; people have attributes such as name, height, shoe size, and so on. In object-oriented programming some objects can also have procedures (some might call them *functions*) that use the values associated with the object to calculate some other value or to achieve some effect; these are called *methods*. A method is some set of steps that can be carried out to do something useful or interesting with an object; obviously a method often makes use of the values of an object's attributes. An object is therefore defined by the values of its attributes and the behaviour of its methods.

More formally an *object type* corresponds to a homogeneous[2] group of objects in the real world. Examples might be ducks, pop singers, or people called James. Each object type has a particular set of *instances*, for example, Elvis,[3] Britney Spears, and James Gosling. An object type has attributes and methods and each instance of the object type has particular values (of the attributes) and therefore different behaviours (of the methods) for each of these. Normally, when taken together, the values for an object instance uniquely describe that instance.

An object type is sometimes called a *class*. This is the name that we will meet later on in Java, and is the name used in most object-oriented programming languages. This chapter is not about a specific language, so we'll stick with the more general "object type", but remember that these are basically the same thing.

Let's look at these ideas in more detail.

Attributes

Objects are all around us. You're holding one in your hand now.[4] The object you are holding is a book. Imagine that you were trying to describe this book to someone else so that they could go and find it in a library full of similar books. You'd probably tell them the title, maybe the name of the author, and perhaps you'd also describe the cover. You would tell them some things that were special about this book; ideally you would tell them the things that uniquely identify it among all other books. You would not tell them that the book is printed on paper because all books are like that; you would concentrate on the things that make this book easily identifiable among all other books. In object-oriented terms you would be telling them about the values of this book's attributes.

An object's attributes are the features that identify or characterise it. A book has a title, an author, an ISBN number, and many more attributes (look at the

1 You may have done some entity-relationship modelling. Objects and entities as used in this sense are much the same thing.
2 It means "all the same sort of thing".
3 The duck, not the singer.
4 Or else you're reading this in a very strange way.

first page inside the front cover). If you give someone the values for these attributes for a particular book they can always find the precise book you mean; the values of these attributes uniquely determine and describe the object.

You are an object too. You have attributes that describe yourself; you have a name, an address, a birthday, and many more. You are an instance of the object type "Person". So are all other people, but they have different values for their attributes. It is highly unlikely that there is another person who has the same set of values for all your attributes, or at least that we couldn't think of a set of attributes that would distinguish you from all other people.

To summarise:

Attributes of a book	Attributes of a person
Title	Name
Author	Date of birth
ISBN number	Address

You can probably think of several more possible attributes to add to each list. It is important that each attribute has one and only one value for each object instance. People do not have two names[5] and books don't have two titles. No attribute of an object instance should be allowed to have more than one value.

As well as a name each of these attributes has a particular *type*. There is a particular type of value that each attribute can take; a book's title is a string of characters, an address is a longer string of characters, a date of birth is a date, and so on. Each attribute has exactly one value and, in most programming languages, that value always has the same type for each object of the type.

Programming languages support a range of possible types of this sort; the names used are slightly different in different languages but the basic types are the same. Common ones are:

Integer	Whole numbers, positive or negative – in C++ and Java `int`.
Floating-point numbers	Numbers with decimal parts, again positive or negative – in C++ and Java `double`.
Character	Single characters, generally anything found on a standard keyboard – in C++ and Java `char`.
String	Sequences of characters, normally spelling words or similarly meaningful sequences – in C++ `string`, and in Java `String`.
Boolean	True or false values, named after their use in Boolean logic – in C++ `bool`, and in Java `boolean`.

Whenever an attribute is identified it should also be possible to determine the attribute's type.

You can also see here that different programming languages will (somewhat annoyingly, it must be said) use slightly different names for the same types.

5 Name here means, of course, forenames and surname (or family name) together.

To keep things simple, we'll just use simple words for the rest of this chapter; the details of how Java refers to these things can come later.

Let's look again at the attributes of a person and add in the type of value for each. A good name for this object type would be, unsurprisingly, `Person`. The precise list of attributes for a `Person` object type would probably depend on the purpose of the program that was going to use it, but some common ones would be:

Person	
Attribute	**Type**
Name	String
Gender	Character (either 'M' or 'F')
Age	Integer
Date of Birth	String
Height	Floating-point (in metres)
Weight	Floating-point (in kilograms)
Address	String

This is far from a complete list; you can probably think of some more attributes to add. Whatever attributes are added each can have only one value of one fixed type. Sometimes, as in `Gender` here, the value can be only one of a few from the type. In most cases it can be any value.

Look again at the type of the "Date of Birth" attribute in the `Person` object. At the moment this proposes that the date is stored as a string of characters. It is indeed possible to use a string like this to store a date, and there are many possible formats that might be used. The format that would be preferred would depend on factors such as how and where the date was to be displayed or the conventions of the country where the program would be used. Examples might be:

```
2nd January 2004
2-1-04
2/1/2004
1/2/2004
```

which could all correspond to the same actual date.

The scheme of storing a date as a string would work if the only operation ever required on a date were to print it out and if all the people who would see this would have the same understanding of the format. Complications would quickly arise if the date were to be processed in some way or needed to be printed out in a different format. Even simple tasks such as finding the next date or a person's twenty-first birthday would be extremely difficult if the date were available only as a string.

For anything but the simplest problems the string representation for dates would be far too inflexible. Something much more powerful is needed. The solution to this problem is quite simple; the "Date of Birth" attribute of `Person` is in fact another object, one that can represent dates. An object's attributes may themselves be objects.

Let's consider what the attributes of a `Date` object type should be. The simplest scheme is to have three integers as attributes; one each for day, month,

and year:

Date	
Attribute	**Type**
Day	Integer
Month	Integer
Year	Integer

There are other possibilities. Many computer systems handle dates by storing the number of seconds since some known fixed point in time; this is a simple scheme involving only a single integer and is cheap for a computer to store. This simple internal representation is converted to a more human-friendly format whenever it is displayed. The advantage of either of these schemes is that the date is stored in a simple neutral form and can then be displayed in whatever format the user chooses. The simple storage makes it easy to develop programs that manipulate dates.

Methods

Displaying the date is an operation (or procedure) that uses an object. This operation is obviously closely linked to the object and in a way it is also an attribute of the object. It is however obviously a special kind of attribute since it is a procedure (a set of steps or instructions) rather than just a single value.

"Attributes" like this are called *methods*. A method can be thought of as a procedure that uses the values of the attributes of an object to produce some useful result or effect. Possible methods for a `Date` object type might be:

- find the next date;
- find the next date and change the values of the object's attributes to the values of the next date;
- find the day of the week of the date;
- determine if the date is valid;
- display the date in the format DD/MM/YYYY;
- display the date in a format similar to "2nd January 2004".

Some of these methods are also like attributes in that they generate a particular value. Finding the next date presumably generates the value of another date and determining whether or not the date is valid generates a Boolean (true or false) value. This value is called the *return value* of the method. This return value also has a type; this is usually called the *return type* of the method.

Some methods, such as displaying the date, do not return a value at all. Usually they just display something on the screen or make some change to the values of one or more of the attributes. These methods are called *void methods*, and `void` is the keyword used in Java (and C++ for that matter) to denote them.

A more complete specification for the methods of a `Date` object could be as follows. This version also adds in some names for the methods; the meanings of these should be reasonably obvious and they are much less cumbersome that the descriptions. Some of the methods in this list return a value while others are void methods.

Date Methods			
Method	**Name**	**Return Value**	**Return Type**
Find the next date	*findNext*	The next date	Date
Find the day of the week of the date	*findDay*	The day of the week	String (or perhaps an integer)
Find the next date and change the values of the object's attributes to the values of the next date	*advance*	Void (the value of an attribute is changed)	
Determine if the date is valid	*isValid*	True if the date is valid, false otherwise	Boolean
Display the date in the format DD/MM/YYYY	*displayShort*	Void (there is output to the screen)	
Display the date in a format similar to "2nd January 2004"	*displayLong*	Void (more output to the screen)	

This is far from a complete list and once again you can probably think of several more possibilities. The precise list of methods for a `Date` object type to be used in a program would depend entirely on what the program was intended to do.

Describing an object

Attributes and methods together define an object type. For example, a more complete definition of our `Date` object type might be:

Date	
Attribute	**Type**
Day	Integer
Month	Integer
Year	Integer

Date methods			
Method	**Name**	**Return Value**	**Return Type**
Find the next date	*findNext*	The next date	Date
Find the day of the week of the date	*findDay*	The day of the week	String (or perhaps an integer)
Find the next date and change the values of the object's attributes to the values of the next date	*advance*	Void	
Determine if the date is valid	*isValid*	True if the date is valid, false otherwise	Boolean
Display the date in the format DD/MM/YYYY	*displayShort*	Void	
Display the date in a format similar to "2nd January 2004"	*displayLong*	Void	

This definition describes all the attributes and lists all the methods available. There is nothing else that this object type can store, represent, or do.

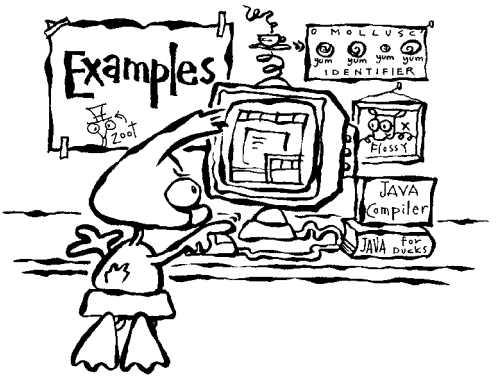
From the point of view of a programmer writing a program that would use this object type, this definition specifies the names and types of the values stored in any instance of the object type, and describes all the methods available. A lot of information is provided about the methods; the specification provides their names, what they return, and the type of that return value. This is all the information needed in order to develop programs that use `Date` objects.

Object-oriented programming

You will probably have guessed by now that object-oriented programming is programming using objects. This is a very popular and powerful way to write programs; after all, the reason that Bjarne Stroustrup proposed the development of C++ from C was to introduce object-oriented features into a language that was already successful and popular. Java is one of the latest languages in this development, and probably the most popular at the moment; it is a much more pure object-oriented language than C++. These developments have also been carried on with another language, C#. To see why this approach is so popular, let's look at how object-oriented programs are developed.

The first step in writing an object-oriented program is to identify the types of the objects that will be used. This can often be done from a description of the problem area and by discussing the problem with those familiar with it. We'll look at this important process in more detail in the next chapter. These object types are specified in terms of their attributes and methods and then programs are written to implement them. Finally the programs that use the instances of these object types (these instances are called, unsurprisingly, *objects* for short) are themselves written.

This is the power of object-oriented programming. If programs have been written to implement a particular object type, the program code can be re-used in many other programs that also make use of the object type. Programmers can quickly build up libraries (or, in Java parlance, *packages*) of working object types that they can use over and over again in many different programs. The savings in terms of time and money can be enormous.



Example 1 – Mr Martinmere's nature reserve

Mr Martinmere is in charge of a small nature reserve that is home to ducks and coots. He wants to develop some computer programs to help with the running of the reserve. What object types will he need?

Mr Martinmere wisely plans to identify and then develop the object types to be used in his programs first. He should find that the object types produced could be used in all the programs he develops, and that he will therefore save a lot of time and effort.

The two object types that he needs are fairly obviously `Duck` and `Coot`. There are many possible attributes, so he would need to think a bit more about the programs that he wants to develop and the information that they will process and produce for him.

The attributes for the two objects would probably be almost the same; this might suggest that one *`Bird`* object type would be better for this application. A decision on which to use would depend on some further analysis of what the programs will eventually do; will they have to distinguish between different kinds of bird? The best choice for the object types would also depend on whether there are any plans for developing more programs in the future and, if so, what those plans are.

Some possibilities for the *`Duck`* object type are:

Duck	
Attribute	**Type**
Name	String
Age	Integer
Value	Floating-point (representing money)

The *`Coot`* object type would be very similar, although differences might be uncovered by discussing the requirements in more detail.

Example 2 – Bruce's library

Bruce has a small library of books that he loans out to the ducks. Each duck is allowed to have only one book at any one time. Bruce wants a computer program that will keep details of all his books and will let him keep track of which duck has which book and when it is due to be returned.

What object types would the program need? What would be their attributes and methods?

Again it is fairly easy to identify the object types in this example. One will be used for details of the books themselves and one for the borrowers. Even though the borrowers are actually the same as the ducks in Mr Martinmere's program from the first example, the attributes will be different and so it is best to use a different object type.

The two object types could, of course, be implemented together as the same object type, presumably called simply *`Duck`*. The drawback with this approach is that there are probably very few situations where ducks borrow books and rather more where ducks are ducks. A general-purpose object could be used but it would involve keeping a lot of attributes and methods that would not be used in the vast majority of the applications that would make use of the object type. At the same time there are many applications where things are borrowed and so a more generic *`Borrower`* object type is likely to be of much more general use.

This chapter has already looked in passing at some of the attributes that a book object might have. The ISBN attribute of the book object is very useful here since it is guaranteed to be unique for every book. Similarly it would be a good idea to introduce a "borrower number" for the borrower object type; two ducks

might well have the same name and this will save any possible confusion and embarrassment. The *Borrower* object type might look something like this:

Borrower	
Attribute	**Type**
Borrower number	Integer
Name	String
Book borrowed	Book
Date due back	Date

Two of the attributes of this object type are themselves objects. The book that has been borrowed is a *Book* object and the date when it is due back is a *Date* object as defined earlier in this chapter.

There would probably also be methods for borrowing and returning a book. The return value from the method to return a book could be used to indicate whether the book was overdue when returned:

Method	**Return Value**	**Return Type**
borrowBook	Void	
returnBook	True if the book is returned in time, false otherwise.	Boolean

Example 3 – Bruce's slightly more complicated library

The ducks like reading but do not like having to make the long walk or flight to return their books. Under pressure, Bruce agrees that each duck should be allowed to borrow up to four books at any one time. How does this change the object types that have been identified?

When designing object types[6] there is often more than one way. There is more than one possible design and many of these possible designs will be suitable for solving the problem at hand. There are no hard and fast rules; sometimes it just comes down to a matter of the programmer's taste or instinct.

Here is a simple way to extend the attributes of the `Borrower` class to deal with the new requirement:

Borrower	
Attribute	**Type**
Borrower number	Integer
Name	String
First book borrowed	Book
First date due back	Date
Second book borrowed	Book
Second date due back	Date
Third book borrowed	Book
Third date due back	Date
Fourth book borrowed	Book
Fourth date due back	Date

6 And when writing computer programs generally.

There is nothing particularly wrong with this model.[7] It stores the new data well enough and the methods could be extended to record which book had been returned.

An alternative approach involves the definition of a *Borrowing* object type. This is an object that does not correspond with a physical object in the real world but with an event. This is quite common. This object type would have two attributes, the book borrowed and the date the book is due for return. The *Borrower* object type would simply have as attributes four of these objects.

A final alternative building on this second idea is worth a mention. It would be possible (and this is probably the neatest solution of the three) to define an object type called *Set of Borrowings*. As the name suggests this would be a collection of *Borrowing* objects. Each borrower object would have one of these collections as an attribute; this is a far neater solution.

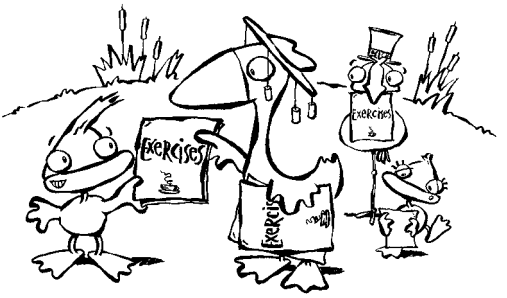
The object types in this scheme would be:

Borrower		
Attribute	**Type**	
Borrower number	Integer	
Name	String	
Books borrowed	Set of borrowings	
Method	**Return Value**	**Return Type**
borrowBook	Void	
returnBook	True if the book is returned in time, false otherwise.	Boolean

Borrowing	
Attribute	**Type**
Book borrowed	Book
Date due back	Date

Set of Borrowings	
Attribute	**Type**
First book borrowed	Book
First date due back	Date
Second book borrowed	Book
Second date due back	Date
Third book borrowed	Book
Third date due back	Date
Fourth book borrowed	Book
Fourth date due back	Date

7 Well, there is. As we will see later (much later – Chapter 17!) this scheme would cause all sorts of unnecessary complications.



4.1 Suggest possible attributes for an object type *Student*.

4.2 Suggest possible attributes for an object type *Professional Footballer*.

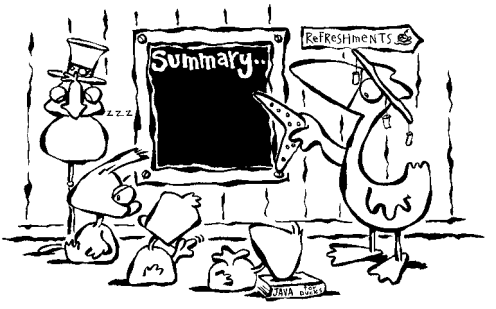
4.3 It is likely that some of the attributes you have suggested for *Student* and *Professional Footballer* are the same. What does this suggest?

4.4 Suggest possible attributes for an object type *Television Programme*.

4.5 Suppose that the *Television Programme* object type were to be used in a program to be used in a video recorder. What methods should be added to the object type?

4.6 A program is to be written to provide details of footballers to a team of match commentators over a season. What methods might be added to the object type you have already defined?

4.7 Bruce plans to allow the ducks to reserve a book that has been borrowed by another duck. What should be changed in the object types identified in the third example?



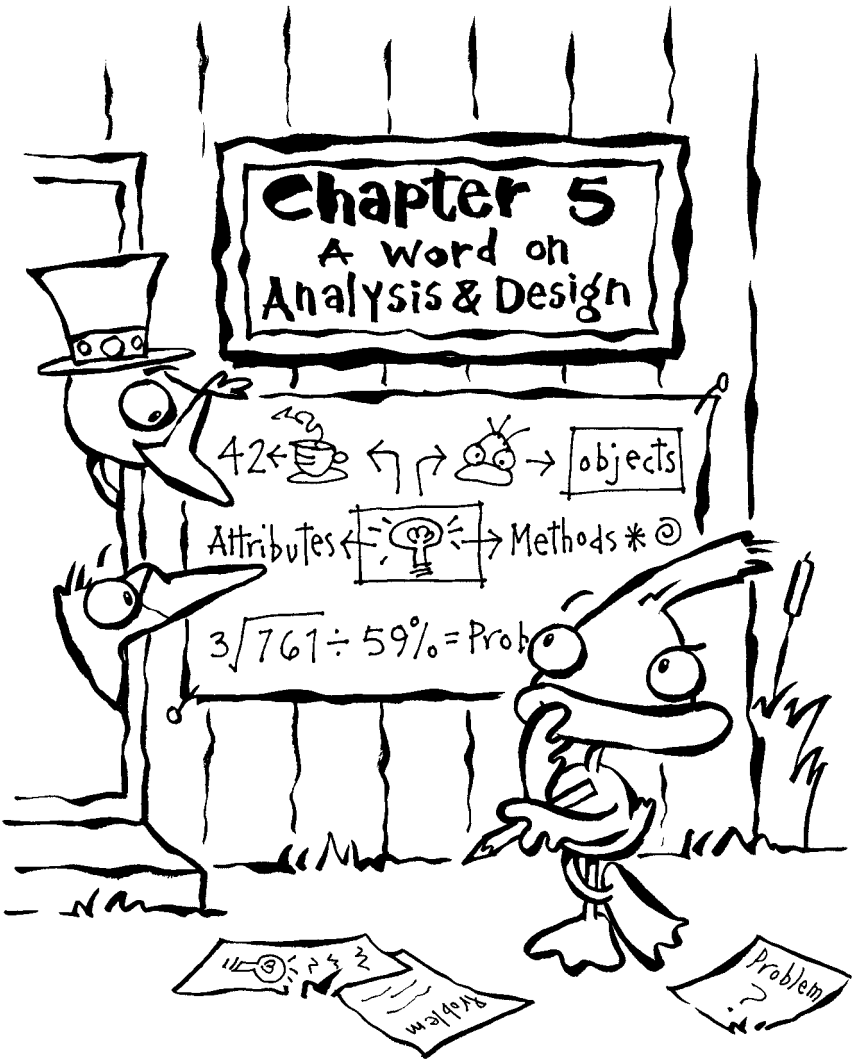
Objects lie at the heart of object-oriented programming. Java is an object-oriented programming language. Java programs define and process objects that represent physical objects or events in some real world problem. The first stage in developing an object-oriented Java program is to identify the object types in the problem and to write programs to implement these. These object types are then used in the programs that are written to solve the problem.

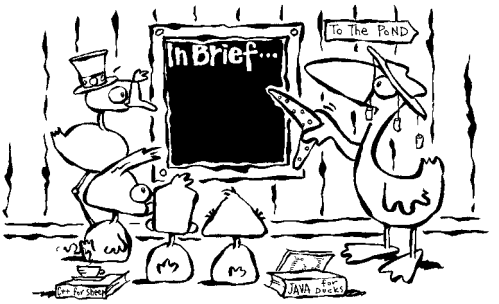
An *object type* is used to represent a group of connected *objects* in a real world problem. Individual instances of this object type (*objects*) are characterised, usually uniquely, by the values of their *attributes*. Attributes are values of interest; they have a *type* and can hold only one value. Attributes may themselves be objects.

As well as attributes, object types also have *methods*. A method is a procedure that processes the values of the attributes of an instance of the object type to produce some effect. A method may or may not change the values of the object's attributes. Some methods will generate and return a value; others, called void methods, will not.

A key principle of object-oriented programming is *code re-use*. When an object type has been written for one problem area it can be re-used in another, ideally with no changes. This has the potential to save a great deal of time and money.

You should now understand what an object is. It's time to move on to look in a little more detail at how the objects in a problem area can be identified.

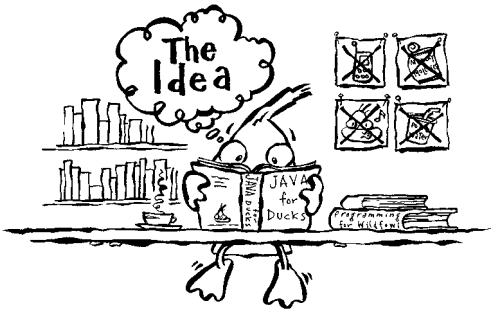

Chapter 5
A Word on
Analysis & Design
42
objects
Attributes
Methods
3√761 ÷ 59% = Prob
Problem
?



The exercises in the previous chapter introduced you informally to the task of identifying objects in a real-world problem. You should now have some experiencing of suggesting the objects that might be used in a program to solve some problem; you should also be able to make some suggestions for possible attributes and methods of the objects you identify.

This chapter takes this a little further to suggest a framework that will allow you to carry out this important task in a structured way. Before this book moves on to describe features of Java, it is important that you realise the crucial importance of good program design. You must understand that, before any program is written, the problem area should be carefully investigated and analysed and the solution carefully designed. It can be a costly mistake to start writing a program before you have properly understood the problem.

After reading this chapter you should have a basic idea of how to identify possible objects and object types in a problem area. You should be able to take a description of a problem and use it to create a list of object types, together with their attributes and methods. You should be able to present this in a way that would be useful to a programmer setting out to implement your solution.



A computer program is written to solve a particular problem or to carry out a particular task. Before this program can be written there are two important tasks that must be carried out. First, the problem must be carefully analysed, and then the solution must be designed. Sometimes more than one solution will be designed to allow the programmers a choice. The activity of analysing a problem and designing a solution is normally called "Systems Analysis", and those who specialise in this task are called "Systems Analysts" or just "Analysts". It is common in today's IT industry for the programmer and analyst to be the same person (an "Analyst Programmer") but this is not always the case.

This is not a book about systems analysis and so this chapter gives you only a very brief insight into the area. At this stage you need to know enough about analysis and design to allow you to make sensible choices as you write your own programs. The activities of analysis, design, and programming are clearly closely linked. You may well find out that you enjoy the analysis or design aspects of developing programs more than the actual programming; that's part of what learning to program is all about!

Before you start on this chapter it is important that you understand that the activities described in this chapter are normally best carried out a long way away from a computer. Analysis involves studying a problem and talking to the people who are affected by it and who understand it (remembering that these two may not be the same!). Design is a pencil and paper exercise, with a computer used, if at all, to do little more than help with drawing diagrams or preparing documents. When you write programs you should never sit down in front of a computer without a design at your side. You should have a clear idea of what your program is going to do and how it is going to do that; this idea is the main output of thorough analysis and design.

There are several stages to analysis and design. These are often described in the form of a structured *methodology*, a formal way to carry out the various tasks. There are many formal methodologies that can be used for analysis, and you might well meet one if you study systems analysis further. For simplicity this chapter does not describe a particular methodology; instead, you will find a less formal approach in the form of a suggested set of steps to take. This suggestion should be particularly suitable for new programmers.

Identify Objects	The first step is to identify the object types in the problem area. In the spirit of code re-use it might even be the case that some of these object types have been met before in previous programs, and so working implementations for them may well already be available.
Identify Attributes	The attributes that describe each object type are identified and added to the definition. The type of each attribute is also identified; this might lead to the identification of more object types that are found to be attributes.
Identify Methods	The methods are added.
Design Methods	Each method is effectively a small program so each one is designed separately.
Design Program	Finally the complete program that will make use of the object types identified can be designed. Only when this step is complete is the program (or any of the methods, for that matter) actually written.

These steps will be illustrated by means of an example. Ready?

Identifying object types

This is the most fundamental step. The object types will be the basic building blocks of the final program and it is crucial to identify the correct ones. A simple way to identify candidates for objects is to take a description of the problem area and to extract all the nouns (or noun phrases); these are the words that refer to "things" that exist in the problem. Such a description would normally be a result of an interview with someone close to the problem.

To illustrate this, here is a description of a problem:

> *The nature reserve is situated just north of Liverpool. It is home to many valuable birds. Mr Martinmere is the ranger in charge of the reserve and together with his family he takes care of all the birds at the reserve. There are many ducks, many coots, and two geese at the reserve. The ranger also has a dog, called Carlos. A program is required that will assist with the administration of the reserve. The ranger wants to be able to keep track of all of the birds; he needs to know the name, age, and approximate value of each. If he sells a bird he needs to be able to record this fact together with the name of the nature reserve he has sold the bird to. He never sells the geese or his dog.*

It's easy to go through this description marking the nouns (here underlined):

> *The nature reserve is situated just north of Liverpool. It is home to many valuable birds. Mr Martinmere is the ranger in charge of the reserve and together with his family he takes care of all the birds at the reserve. There are many ducks, many coots, and two geese at the reserve. The ranger also has a dog, called Carlos. A program is required that will assist with the administration of the reserve. The ranger wants to be able to keep track of all of the birds; he needs to know the name, age, and approximate value of each. If he sells a bird he needs to be able to record this fact together with the name of the nature reserve he has sold the bird to. He never sells the geese or his dog.*

This gives a list of things that might be objects in the final program.

Some of the nouns in the description are actually different names for the same thing, so the next step is to remove synonyms to give a shorter list. At the same time some of the words in the description don't quite describe the potential object fully and so the names should be refined to make them clearer:

- the nature reserve
- Liverpool
- Mr Martinmere
- ranger
- Mr Martinmere's family
- birds
- ducks
- coots
- geese
- dog
- Carlos
- program
- administration
- name
- age
- approximate value
- fact
- name (of nature reserve buying a bird)
- nature reserve buying a bird

Some of these can be eliminated at once. "program", "administration", and "fact", for example, are clearly nothing to do with object types. Proper nouns can also usually be eliminated; they refer to a particular instance of an object type that is very probably already covered elsewhere. Finally, some nouns will actually be attributes of other objects; here "name", "age", and "approximate value" clearly fall into this category. The list is now much shorter:

- the nature reserve
- Mr Martinmere
- Mr Martinmere's family
- birds
- ducks
- coots
- geese
- dog
- nature reserve buying a bird

Information about the tasks to be carried out by the program will enable the list to be trimmed further. "Mr Martinmere's family" is only mentioned in passing, so that can be removed. Much the same goes for the ranger himself and the reserve itself. Finally "birds" is a general term that covers the ducks, coots, and geese; it can also be removed.[1] This gives a final shortlist:

- ducks
- coots
- geese
- dog
- nature reserve buying a bird

The description also provides the information that the ranger would never sell his dog or the geese. Given the nature of the program it might be reasonable to leave them out, but then again the ranger might change his mind in the future. Part of the analysis process would be to go back to the ranger and ask for more details and a more detailed description of the problem. This would allow a sensible and informed decision to be taken.

For the moment the assumption will be that neither is required; this seems reasonable on the basis of the limited information currently available. The list becomes:

- ducks
- coots
- nature reserve buying a bird

The final step is to choose the names for the object types. Singular names are customary, so ducks becomes "duck" and so on. The current name for the object type representing the reserves that buy animals is rather unwieldy; this could be changed to simply "reserve". The final list of object types in this problem

1 This actually shows, of course, that there is an object type `Bird` in this problem and that it is made up of three separate object types that represent particular types of bird. If we were to find out that each type of bird had the same attributes it might well be possible to implement a single Bird object type that would contain an attribute to indicate the type of bird.

area is then:

- duck
- coot
- reserve

Identifying attributes

Finding the objects has provided a useful head start in finding some of the attributes. A check through the discarded candidates for objects reveals "name", "age", and "approximate value", all of which are attributes of the `Duck` and `Coot` object types. It's likely that they are also attributes of the other two discarded animal object types (geese and dog) as well, but this would only be relevant if these were to be implemented.

There is less to go on in the description for the attributes of the object type that will represent the other nature reserves who buy the birds. In practice an interview would again be used to determine these, but a reasonable guess for them would be to include attributes such as "name", "address", and "telephone number".

Finally, there is a requirement to record the identity of the reserve to which a bird was sold. This can be introduced in two ways; either an attribute "buyer" can be added to the `Duck` and `Coot` object types, or an attribute to represent the birds bought by each of the other reserves could be added to the `Reserve` object. There is no good reason at this stage to believe that either of these possibilities is better than the other; the choice could be left to the final programmer or might be determined when some more details of the requirements of the final program are known.

The types of most of the attributes are straightforward to establish. The only one that might present a choice is the telephone number of the reserves. It might seem at first sight as if this should be an integer, but a string is probably to be preferred. Telephone numbers are traditionally written with spaces to make them more memorable and often begin with a 0; handling these values sensibly would be very difficult to achieve if the number were stored as an integer. It is also unlikely that the final program would need to perform arithmetic on a telephone number (an operation on integers), but it is possible that the program might need to extract certain digits from the number (an operation on strings). A string is best.

Finally the names of some of the attributes could do with some simplification; in particular "approximate value" should be simply "value". This gives the final set:

Object Type	Attribute	Type
`Duck`	Name	String
	Age	Integer
	Value	Floating-point
	Sold to	Reserve
`Coot`	Name	String
	Age	Integer
	Value	Floating-point
	Sold to	Reserve
`Reserve`	Name	String
	Address	String
	Telephone number	String

Identifying methods

The task of identifying the methods of the object types involves returning to the description of the problem area. This time the focus is on the purpose of the final program; it is vital that the object types have sufficient and correct methods to support this. In particular the analyst should be looking for any *transactions* that will change the values of any attributes of any instance of an object type. Transactions often show up as events in the problem description.

In this example there is only one obvious candidate; the ranger sells a bird. This transaction would change the value of the "sold to" attribute of the object instance representing the appropriate bird; it is therefore a method of both object types representing birds, `Duck` and `Coot`.

The description also contains the information that the ranger "needs to know the name, age, and approximate value" of each of his birds. This suggests two things; that methods will be needed to display this information to him in some neat format and that methods will be needed to change this information. It seems reasonable to assume that the name of a duck never changes, so methods will be needed only for age and value.

The `Reserve` object type has no obvious methods. The only possibilities seem to be methods to change the values of the attributes, but there is no evidence that this would be a common requirement. This is another example of something that would have to be confirmed with a further interview.

Of course Mr Martinmere will also want to use the information about his birds for many other purposes. He might want to display lists of all his birds or find all the birds of a certain value. These are not, however, methods of the object type; they operate on collections of objects and not on individual objects. These are functions that would be carried out by programs that use the objects and they should be left to the programmer to implement.

This analysis gives the following collection of methods for the `Duck` object type. The methods for the `Coot` object type are the same as those for `Duck` (and so it would be a sound idea to give them the same names), and the `Reserve` type has no methods.

Object Type	Method	Description	Result	Result Type
`Duck`	`display`	Displays details of a duck in a neat format.	Nothing, just display on screen.	Void
	`sell`	Records that a duck has been sold and stores the name of the reserve buying.	Nothing, underlying values are changed.	Void
	`changeAge`	Changes the duck's age.	Nothing, underlying values are changed.	Void
	`changeValue`	Changes the value of a duck.	Nothing, underlying values are changed.	Void

Designing the methods

Each method must now be designed. This involves specifying the effect of each method in detail. It is normal to use a structured format for writing this involving the specification of:

- the purpose of the method
- the inputs of the method
- the values that are changed by the method
- any side effects (such as output on a screen or changes to a file of data) that the method may produce

The format of this information should be fixed and, in practice, might well be recorded on some suitable form.

For example, the specification for the *display* method of the *Duck* object type would be:

```
Name:            display
Object type:     Duck
Purpose:         Displays the details of a duck on the screen
Inputs:          a Duck object
Values Changed:  None
Side Effects:    Display on the screen
```

This specification would be extended in the final design process to include a precise description of the layout for the display; this might well make use of a diagram. In practice it is likely that the analyst would use some sort of form to record the specification for the programmer.

The *sell* method has an input; the programmer must provide a reference to the reserve to which the duck has been sold. This method also carries out a rather more complicated task than simply displaying the values on the screen, so a more lengthy description of its effect would be provided:

```
Name             sell
Object type:     Duck
Purpose:         Records that a duck has been sold
Inputs:          a Duck object and a Reserve object
Values Changed:  "sold to" attribute of the duck
Side Effects:    None
Description:     The value of the "sold to" attribute of
                 the duck should be changed to store the
                 Reserve object provided. If this value
                 already indicates that the duck has been
                 sold an error message should be displayed
                 and no changes should be made
```

Obviously this detailed description can become very complex. Worse than that it can be very hard to write it with precision; if the programmer misinterprets what the analyst has written the final program will not work as expected.

For this reason it is useful to be able to write the description out in a slightly more formal and unambiguous way. The normal way to do this is to write in a form of *pseudocode*. This is a form of language that is much more terse and precise and is, in fact, quite close to a computer programming language. There are no hard and fast rules for writing pseudocode (although some software companies may have a preferred or recommended style), except that anything

written should be clear and unambiguous. The description of the processing carried out by the *sell* method could be written in pseudocode as:

```
IF the value of "sold to" is blank THEN
  Set the value of "sold to" to the Reserve object
OTHERWISE
  Display an error message – the duck is already sold
END IF
```

Armed with the pseudocode, the programmer will be in no doubt about the analyst's intentions. It is good practice to write out all but the very simplest methods in this way before writing them in actual program code.

Designing the program

With the object types designed, the program or programs that will use them can also be designed. As with the design of the methods, once the purpose of the program has been specified it is usual to describe it more formally using pseudocode.

One program that is clearly needed is one that will allow Mr Martinmere to record the sale of a duck. This would presumably ask him to enter the name of the duck and the name of the reserve buying and would then make the required changes in his records. The pseudocode for this might be:

```
Display "Enter the name of the duck:"
Read Input and store as theDuck
Display "Enter the name of the reserve:"
Read Input and store as theReserve

Use sell method of duck object to record sale
```

This simply and unambiguously specifies the method and tells the programmer precisely which other method to use.

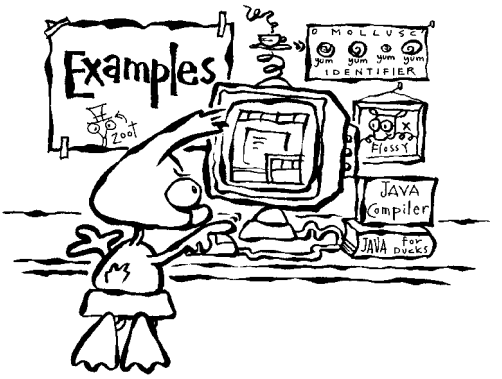
As another example, suppose that Mr Martinmere decides that he wants a program that will display the details of all his coots. This program will have to process and display the details of each coot in turn, using the *display* method. This could be expressed in pseudocode:

```
Get the details of the first coot
WHILE there are coots left
  Display the details of the coot using the display method
  Get the details of the next coot
END WHILE
```

Design of anything more than simple programs will have to wait until you have learned more of the details of programming. For the moment you should just appreciate the importance of design and the great importance of expressing the design in an unambiguous way.

Analysis and design

This example has shown how a description of a problem can form the basis of an analysis of a problem area. This analysis leads to a design of a solution. The approach suggested here is a simple methodology that you should follow whenever you come to develop a new program.



Since this chapter included one long example there are no extra examples this time. On with the exercises!

5.1 Identify the object types that would be needed in a solution to the following problem. For each object type you identify, suggest attributes and methods.

> *The ducks on Mr Martinmere's farm have taken up cricket. They have formed themselves into four teams and are organising a knock out competition. They have asked Bruce and Zoot to be umpires. Elvis has been asked to write a program to record which duck is in which team and to keep a record of the results of the matches.*

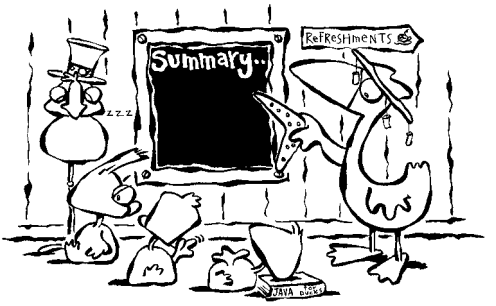
5.2 After the success of their first competition the ducks decide to organise a league. Each team plays each other and two points are awarded for a win, one for a draw, with none for a defeat. Elvis is asked to write a new program. How does this change the design of the object types?

5.3 Elvis has a file containing the results of all the matches in the league. Use pseudocode to design a program that would process the file to discover how many points each team has got so far. You can read through the file many times.

5.4 Express the following description of a program in pseudocode. Assume that the program uses a `Book` object type that has an attribute "borrowed by" storing the duck that had borrowed the book.

> *When a book is returned Bruce enters the name of the duck and the title of the book. The program finds the information about the book and records the fact that it is back in the library.*

5.5 Examine the job advertisements in a computing trade newspaper such as *Computing* or *Computer Weekly*. Note down the job titles for all the jobs relating to programming. How many jobs are offered as Programmers? As Systems Analysts? As Analyst Programmers? What salaries does each job seem to attract? What are the different skills or experience requirements for each type of job?

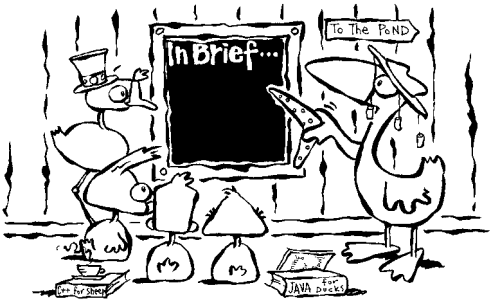


Before any program can be written the problem that it is to solve must be thoroughly analysed, as must the area in which the problem exists. This analysis, which is often carried out by a Systems Analyst, forms for the basis of the design of a solution. The programmer works with this design to produce the final working program.

The first stage in the analysis is the identification of the object types that will be used in the program. Their attributes and methods are also found and the methods are designed. This design normally makes use of pseudocode to ensure that it is unambiguous. The ambiguity that can result from the use of normal natural written language is best avoided. The use of pseudocode is also usual and useful when designing programs.

Thorough analysis and design is essential before programming can start. Now that you've appreciated that you're getting closer to doing some actual programming! As a first stage we'll look at how one of the object types we've identified in this chapter can be implemented as a Java class.

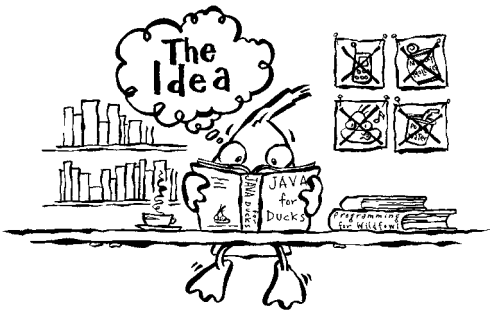

Chapter 6
A First Look
JAVA



Java is big. As you learn more about it you will come to realise just how true this is. Java is a truly general-purpose programming language; you could use a program written in Java for just about anything from flying an airliner to keeping track of numbers on a mobile phone. This is good, because you are going to learn a powerful language with many useful applications. But it is also bad because it's difficult to know where to start. There really is a lot going on in even the simplest Java program.

This chapter is your start with Java. This chapter provides you with a quick tour of the basic Java that we're going to learn about in this book. This is not all the Java in the book, and certainly not all the Java that there is, but it's a reasonable start. At this point I'm going to gloss over some of the details; the idea here is to give you a general feeling for what's going to come later.

In the last chapter we identified object types in a problem area. We identified their attributes and the types of those attributes. We also found some of their methods and used some pseudocode to design how these might work. Now is the time to see how some of these things look in Java.



To start, let's recap the object type that we will model in Java in this chapter. The type is to be used to store the details of ducks on Mr Martinmere's reserve. You might want to go and check over the full details of this object type's attributes and methods at the end of the last chapter.

To keep things nice and simple we'll just worry about three attributes – name, age, and value. The types of the first two are easy to identify, and a string and integer respectively will be fine. The type to be used to store the value is slightly more complicated; this is actually an amount of money (and as such follows some special rules such as only having two places of decimals) but there is no handy type for that in Java. So we'll make do with using a simple floating-point value for that.

We'll also content ourselves with just three methods. There will be methods to display the duck's details in some neat format, and to change the duck's age and value. Again, the details of these are defined in the last chapter.

If we were to be implementing one of Mr Martinmere's programs we would obviously need to be able to process an instance of this object type. One instance would presumably represent one "real world" duck. Of course, the point of object-oriented programming is that the implementation of this object type could be used unchanged in any program that Mr Martinmere might want now or in the future. For that matter the object type could be of use to anyone who wanted to write a Java program that processed basic details of ducks. We must therefore implement[1] the object type before we start on any of the programs. First we will need some basic Java; about the most basic are *declarations* and *statements*.

Declarations and statements

We know that the first step in writing a Java program is to define the object types that will be used. An object type is called a *class* in Java (and in most other object-oriented programming languages as it happens), so that's the name used in this chapter. A class is defined, not surprisingly, using Java declarations and statements.

A declaration simply "declares" that something exists in the problem area. It probably specifies what this thing is called, what sort of thing it is, and also provides some other information determined by what is being described. So, in Java, this declares that an integer (abbreviated to `int`) exists:

```
int age;
```

A string and floating-point value look similar but use less obvious (or more subtle) names:

```
String name;
double value;
```

`String` refers to the required string, and has to start with a capital letter.[2] `double` refers to a floating-point number, with "double" referring to the precision that will be stored. This is basically the amount of computer memory that will be used to store the value; you can think of it as the amount of memory available to store the decimal places for a floating-point number. Your *Local Guide* might tell you how much is used on your system.

These declarations do not really do anything, or at least they do nothing that a user of the program might see. Behind the scenes some memory is allocated to store the value, and various things will be done to associate a name with the memory, but nothing happens as far as someone using the program is concerned.

A statement is a line of Java that achieves something useful. It might set a value or display something on the screen for example. This would set a value by assigning the value 10 to the attribute *`age`*:

```
age = 10;
```

---

1 And, as we will see, also test.

2 And that is the first thing that I'm going to gloss over! If you must know, it's because `String` is a class itself in Java, but the other two types aren't.

and this would display a welcoming message on the screen:

```
System.out.println ("hello, world!");
```

The punctuation in all these lines so far is important. They all end with semi-colons, which effectively mark the end of the line. Lines can spread over more than one line of a program, so the semi-colon is important. This, for example, is strange but allowed:

```
System.out.println
("hello, world!");
```

because the semi-colon clearly marks the end of the statement. This, on the other hand, is an error:

```
System.out.println ("hello, world!")
```

because there is no semi-colon. The Java system would assume that the next line in the program was also part of this line, and all sorts of terrible things would happen.

In a way, that's all there is to it! A class definition in Java is simply a collection of declarations and statements. The declarations declare the attributes of the class and statements are used to implement the behaviours of the methods. A program is much the same thing, of course.

It is a feature of programming that the syntax of the languages used has to be precise. When writing other languages you are probably used to being allowed some flexibility; you can usually make mistakes and get away with it. I can potato make a mistake in a sentence and you will still understand what I'm saying. I can mispell a word and you can still understand this sentence. Programming languages don't allow the programmer as much room for manoeuvre. So this is fine:

```
System.out.println ("hello, world!");
```

but this is an error which would mean that the program would fail to compile:

```
System.out.prntln ("hello, world!");
```

You are going to have to get used to this precision. It can be very frustrating at first, and it does take a bit of getting used to.

Attributes

We now have enough Java to define the attributes of the class. The three declarations are:

```
String name;
int age;
double value;
```

The values of these attributes will store everything that we know about a particular duck. Later on we will see that it's important that the values of these attributes are protected from changes made in error by badly written programs. Programs will be able to change the values of course, but only on the terms

that we set. We want no negative values for age! For this reason we define the attributes of the class as *private* to the class, so:

```
private String name;
private int age;
private double value;
```

That's all there is to it.

Methods

In contrast the methods are the *public* face of the class. Methods will be used to allow access to the values of the private attributes; this will include changing them as well as retrieving them and displaying them. Before worrying about the intricacies of this, let's get together the statements for a simple method.

The first method in this example does little more than display the values of the attributes in some neat format. We have seen the Java for displaying a value:

```
System.out.println ("hello, world!");
```

which would display the text between the quotes as shown:

```
hello, world!
```

This can also be used to display the value of an attribute. The trick is to leave the quotes out. So this:

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

would print whatever string was currently stored in that attribute. The statements to print out the values of the attributes of a duck are then just:

```
System.out.println (name);
System.out.println (age);
System.out.println (value);
```

This is admittedly not very neat, but it will do for now. It would also be a good plan to provide a user with some information on what the values represent. This is another simple statement (with quotes needed this time), and completes the statements needed for this method:

```
System.out.println ("The details of the duck are:");
System.out.println (name);
System.out.println (age);
System.out.println (value);
```

Now the method needs a name. *display* would seem a reasonable choice. A method also has to be declared. In the last chapter we saw that this involves specifying the name of the method and the type of the value that it returns. This simple method will return nothing at all (so it is a void method), which in Java is represented as `void`. The method can now be declared:

```
void display
```

Remembering that it is public:

```
public void display
```

Finally, some methods take values to process (the other two in this example will), and these are listed in brackets the declaration. There are none here, so the full declaration is:

```
public void display ()
```

No semi-colon this time!

The method is completed by adding in the statements. They are written between a pair of curly brackets usually called *braces*, and are indented slightly to show someone reading the program that they are inside the method. We now have a complete method:

```
public void display ()
{
  System.out.println ("The details of the duck are:");
  System.out.println (name);
  System.out.println (age);
  System.out.println (value);
}
```

The remaining two methods are slightly more complicated. The extra complication is that they change the value of an attribute. They therefore need to know what the new value is. For this example we'll assume that the value they are provided with is valid, so we'll ignore the problem of an age being negative, for example.

Values provided to methods are called *parameters* and are listed in the brackets in the method's declaration. This list includes the name of the parameter and also its type; the parameter can then be used inside the method, just like the attributes.

The second method is intended to alter the value of the duck's attribute storing its age. A reasonable name for it would be *setAge*. It is going to need to know what the new age is, and therefore it needs one parameter, an integer representing the new age. It doesn't return a value, so the return type will once again be `void`. Apart from this its declaration is much the same as the simple display method:

```
public void setAge (int newAge)
```

The only statement needed in the method is one to set the value of the attribute *age* to the value of the parameter (*newAge*). We've seen a statement to set a value at the start of the chapter, so:

```
age = newAge;
```

will set the value of the attribute *age* to the value found in the parameter *newAge*. Behind the scenes the value in the memory location storing *newAge* will be found and will be used to replace the value in the memory location storing *age*.

And the complete method now becomes:

```
public void setAge (int newAge)
{
  age = newAge;
}
```

Happily, the method to set the duck's value is pretty much just the same. The only difference is that the parameter is now a floating-point number and that the name of the attribute and parameter have changed.

```
public void setValue (double newValue)
{
  value = newValue;
}
```

We now have all the methods to go with the attributes. All that remains is to combine them into a definition of the class.

Defining the class

The Java files that contain definitions and programs have a fixed structure and format. For this example we're only defining a class, but the structure would be much the same if we were writing a program. Once again this format is fixed; the programmer has some flexibility with the precise details of layout, but there are strict rules that have to be followed.

A Java file can be rather long. It can also be difficult to interpret, especially if it has to be read by a programmer who did not originally write it. It is therefore usual (and a good idea) to start each file with some *comments* to describe the files' purpose. A comment is part of a program that is intended only for a human reading the program; comments are always completely ignored by the compiler.

There are two ways of writing comments in Java. The first is to enclose the comment between two symbols:

```
/*
  This is a class that can be used...
*/
```

The compiler will happily ignore anything it finds in the program after the first `/*` and before the corresponding `*/`. This style of comment is usually used for long comments that extend over several lines. The alternative is to use the `//` symbol to mark that anything to the right on the line is a comment. This would look like this:

```
System.out.println ("hello!"); // Print a greeting
```

The choice is really just down to the style of the programmer,[3] but the style in this book is to use the first choice for long comments that span over several lines and the second for shorter comments that might relate to a single line.

One place where there should definitely be a long comment is at the start of the file. This comment should explain what the class or program in the file does, and should also provide useful information such as the name of the author. In time it might also record any changes made to the program.

Before adding a comment we need a file to add it to! Java has rules about the names of files, so we don't have very much choice about the name. The class is called *`Duck`*, and if a file contains the source code for a public class, then the filename must be the name of the public class followed by a *`.java`* extension. In the course of this book, we will only encounter public classes, but it is worth mentioning that in more advanced projects, you may well find *`.java`* files

3 We'll talk more about programming style later on.

containing more than one class – most will be `private` classes, and at most one will be `public`.

So, given that our file contains the source code for the public class called *`Duck`*, our file is named *`Duck.java`*; your *Local Guide* should tell you all you need to know about creating and opening the file.

The file starts with a comment (called a *header block*) that contains the required useful information:

```
/*
  Duck.java
  A simple Duck class for the "First Look" chapter.

  AMJ
  22nd January 2003
*/
```

Now to add the definition of the class. This starts with something that looks very like the definition of an attribute. A class is obviously going to be public since programs will need to create instances of it:

```
public class Duck
```

The remainder of the definition consists of the attributes and methods, enclosed between braces. It is usual to list the attributes first, followed by the methods. Short comments are used to explain exactly what each component is.

That's almost it. The definition of the attributes and methods of the *`Duck`* class now looks like this:

```
public class Duck
{
  // Attributes
  private String name;
  private int age;
  private double value;
  // Methods
  public void display ()
   {
     System.out.println ("The details of the duck are:");
     System.out.println (name);
     System.out.println (age);
     System.out.println (value);
   }
   public void setAge (int newAge)
   {
     age = newAge;
   }
   public void setValue (double newValue)
   {
     value = newValue;
   }
}
```

The indentation of the file shows clearly what is contained inside the class. It is always a good idea to make sure that corresponding pairs of braces line up, as has been done here.

The final thing that needs to be added is a special method. When an object is declared in a program there are many things that need to be done; one is that some memory must be allocated to store the values of the attributes. This is done by a special method called a *constructor*. The constructor takes care of whatever is needed behind the scenes to create a working instance of the class. It can also be used to set some default values if these are appropriate for the attributes.

This might sound complicated, but it isn't. All the memory allocation is done automatically so the only thing that really concerns us is default values. Java will provide sensible default values for the built-in types (0 for numeric types, for example). Assuming that these are good enough for this case (and there seems no reason why they wouldn't be) the constructor can be left empty. In the definition of the class the constructor appears as a method with the name of the class. So we just need to add:

```
public Duck ()
{
}
```

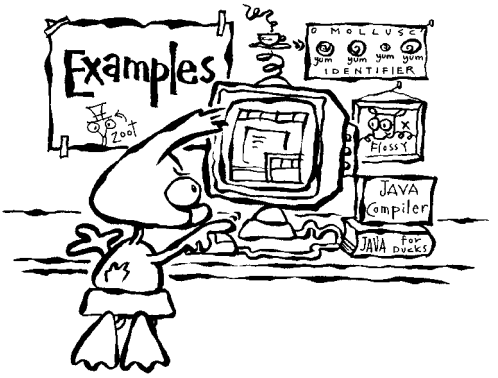
to make the class complete. This can be added together with some comments at the top to give the final version:

```
/*
  Duck.java
   A simple Duck class for the "First Look" chapter.

  AMJ
  22nd January 2003
*/

  public class Duck
  {
    // Attributes

    private String name;
    private int age;
    private double value;

    // Constructor

    public Duck ()
    {
    }
    // Methods

    public void display ()
    {
      System.out.println ("The details of the duck are:");
      System.out.println (name);
      System.out.println (age);
      System.out.println (value);
    }

    public void setAge (int newAge)
    {
      age = newAge;
    }
```

```
    public void setValue (double newValue)
    {
      value = newValue;
    }
}
```

Defining a class

This definition of a class is now ready to be used in a program. Of course, it should be tested first so that any problems in it were prevented from causing any problems in the program. Once it was tested and found to be working it could be used in any program that needed to use the *Duck* object type.



There are no examples of this type since this chapter was one long example. You could now find many more examples by looking quickly through the remaining chapters of the book and looking at the Java you find there. You will hopefully find that you can understand quite a bit of it. You should certainly find that the format and layout of the programs now looks familiar, and you can probably understand a fair bit with the help of the comments.

You should also find that you can take the class definition from this chapter and try it out on your own Java system. Can you make it compile?

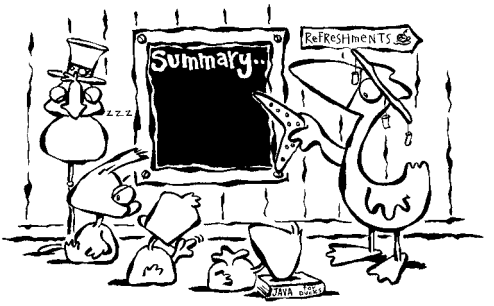
6.1 The last chapter also identified a *Coot* class. It had the same attributes and methods as the *Duck* class. Write out its definition.

6.2 There was also a class for storing details of "Nature Reserves". Write out that definition.

6.3 Take both these definitions and type them in using your Java system. Make sure that they compile and correctly produce a `.class` file for each.

6.4 The methods in this class that set a value did not validate the value provided. What problems might this cause?

6.5 The constructor for the class did not set any special default values. What would good default values for the three attributes be?



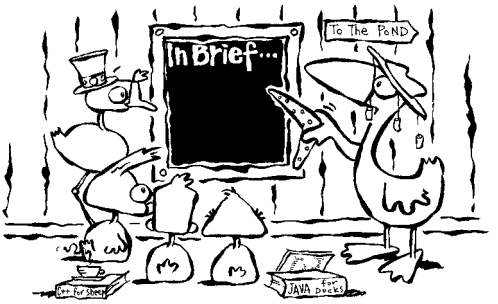
This chapter has been a very quick look at quite a lot of Java. There are a lot of new ideas and concepts in this chapter. They're ideas that we will be meeting in much more detail in the remainder of the book, but hopefully you've now got a general idea of how an object type is defined in Java.

An object type in Java is called a *class*. It is implemented in a file which must have the same name as the class (it follows that only one class can be implemented in a file, by the way[4]). The class consists of a private part and a public part; it is usual to put the attributes the private part and the methods in the public part. A file defining a class should always start with a clear comment explaining the purpose of the class and other comments should be used in the file to explain what is going on. A complete class definition file should compile without errors to produce a `.class` file.

This chapter has raised a few issues about layout and style in programming so, before we get into more Java, the next chapter explains why good layout and style are so important …

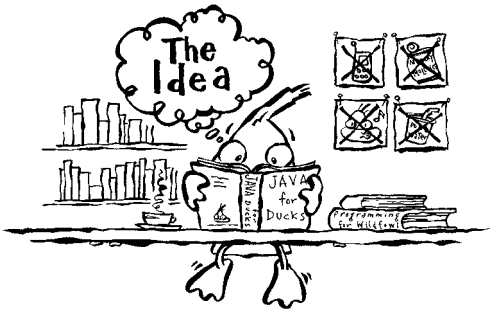
4 Well, only one *public* class. Don't worry about it.


Chapter 7 - Programming
Don't Panic!
Caff
Restaurant
Style
Indentation
Meditation for Ducks



The time has come when you are almost ready to start using Java to write some programs. Before you do, though, you need to understand something about the process of programming. Programming is a highly structured process; some might even call it a discipline. Good programming requires that the programmer adopt a sensible, structured, and measured approach. Above all it requires that the programmer doesn't panic!

After reading this chapter you should understand why a structured approach is essential when writing programs. You should appreciate the importance of always applying good consistent style and layout when you come to write your own programs. You will have met the five golden rules of good programming practice and style. Finally, after a long wait, you will also have seen your first complete Java program that actually does something.



Suppose that you were about to build a new house in the middle of a large empty field. This is one strategy that you might use. You could make a list of all the things that you would need; you would include bricks, cement, wood, paint, windows, and so on. Then you would buy them and arrange them in a huge random pile in the middle of the field. Over the next few days you would poke around at the pile and sort them out and arrange them into a house.

This strategy might just work but many things might go wrong. You might put the bricks on top of the windows and break them, or you might lose the paint under a large pile of wood. The risks of disaster are so great that in practice you wouldn't seriously consider using an approach like this.

Suppose that your car was broken and you took it to the garage for repairs. You would expect the mechanic to listen to your description of the fault and then spend some time diagnosing what exactly was wrong. When the fault had been identified you would expect a quick and easy repair. You would certainly not expect the mechanic to spend time randomly replacing parts of your car

until it worked again. This random approach would probably work, but is far from the cheapest or most efficient way of repairing a car.

These two examples highlight tasks where a structured and methodical approach is essential. You will not be surprised to learn that programming is another such task. There is a set of steps that must be followed, in order, when any program is written. Failure to follow these steps is a sure recipe for disaster. Much the same applies to following the steps in the wrong order or jumping randomly between them.

These two examples of building a house and repairing a car are based on two approaches that new programmers sometimes adopt. Some will create a program of many hundreds of lines before starting to see whether it works or even whether it compiles. Faced with errors, some will then start randomly changing random parts of this program until it works. Either way, these are things that an experienced programmer would never do, and the best idea really is to make sure that you never do them!

Writing a program

A user who requires a new program will expect a programmer to go through a process that is very similar to that you would expect from a car mechanic. The first step is to carry out an analysis to acquire a thorough understanding of the problem. This is followed with a design of the most appropriate solution, which is then quickly and efficiently implemented as object types and programs. It would be a huge mistake to start writing the program before the problem had been fully investigated and understood. It would be equally foolish to start writing the program before it had been properly designed.

The process of writing the program itself follows a similar pattern. The first stage is to implement the object types that will be required. These are tested fully in isolation so that the programmer is sure that they work correctly. Only when they have been thoroughly tested should they ever be used in a program. Once again it is extremely foolish to start writing a program that uses some object types before the object types have themselves been written and tested. Such a mistake would only lead to a great deal of extra and unnecessary work as more and more errors had to be located and corrected. And worse, if a program is written before the object types have been fully tested it is very difficult to know for certain if the errors are in the program or in the object types themselves.

Even the production of the program code itself follows the same pattern. The code should always be developed in a steady incremental manner. Functionality should be added to the program very gradually, and any new function should be tested and shown to work before any further development is attempted. Build the house one storey at a time, and make sure that the bottom one won't fall down before you start upstairs.

Sometime in the future you will be writing a program and you will not be able to get it to work. You will have the chance here to make a huge mistake. You will have the chance to ignore this problem and then starting to work on some other part of the problem. If you do this you are only making worse trouble for yourself! Never start working on a different part of the program when you find that you can't go any further in one area. Find some help and sort out the first problem.

The first golden rule is:

Work on one part of a program at a time. Get it right and then go on to the next one. Never work on many parts of a program at once!

As you learn to program you will probably get the chance to see some experienced programmers at work. One thing that you will notice is that they have to deal with many fewer errors from the compiler than you seem to. This is only natural; you are bound to make more mistakes as you learn. At the same time, though, the experienced programmers are doing some things that will make sure that they have as few errors as possible to find and correct.

An experienced programmer will compile the program many times before it is complete. Sometimes this will just be to check that the syntax of the program so far is correct, and other times this will be to quickly test some new function. An experienced programmer will never, ever, type in a complete program and then start to force it to compile. If you choose to adopt an approach like that you have made a very big mistake; you have chosen the best way to make sure that you waste time and get practice in correcting as many errors as possible. It is in the nature of programming that a simple error at the top of a program file (something as insignificant as a single semi-colon) can cause errors throughout the program; remember the precision that we saw in the small amount of Java in the last chapter. If the program is several hundred lines long this will be several hundred errors, but if the program is only ten lines long there will be far fewer. It is always best to find such errors in very small programs, before their effects become so big as to obscure the original cause. For one thing there are many fewer places for the error to hide!

You will come to need to compile your program just to check the syntax less and less as you gain more experience. As you are just starting out you should remember to compile your program, if only just to test the syntax, every time you add around twenty lines to it.

The second golden rule is:

If you have added twenty lines to your program, finish the part you are working on, compile it and check that your syntax is correct. Fix any errors before you carry on.

Style

A particular aspect of programming is a programmer's style. Every programmer has a personal style in writing programs, something that is probably unique to each individual. It's much the same thing as handwriting; you can probably recognise your own handwriting even if you wrote something many years ago. A programmer can probably recognise a program that they wrote many years ago.

Any programmer would hope to be able to take a program that they had written many years previously and make changes to it. Many programs remain in use for a very long time and changes often have to be made many years after the program was first written. Of course sometimes the original programmer will not be available to make the changes. In this case a different programmer must be able to take the program, quickly understand it and then make the required changes. This is much more complicated to achieve if the program is written in a poor style. Poor style can make even the most useful program

totally impenetrable to even the best programmer. Good style is an essential feature of good programming.

As you learn more and more you will come to develop your own style. Like handwriting, style is very much a matter of taste; you will start to think that certain parts of a program look better to you if laid out in a certain way and you will start to adopt that style. All the programs in this book are written in the style that we are happy with.[1] I'm not suggesting that this is a particularly good or bad style, but it's the one that we have developed over many months and is something that we are reasonably happy with. You are free to copy it as you learn, but you will probably eventually adapt it as you develop your own set of preferences.

There is no such thing as a good or bad style, as long as the style follows some basic rules. It is essential that any style is consistent. If a programmer chooses to use a particular way of laying out a particular part of a program the programmer should use precisely the same layout every time they write a similar piece of program. As a first example, here is a very simple Java statement:

```
number = 1;
```

That line is laid out as I would do so were I writing a program. There is a space before and after the equals sign, and no space before the semi-colon. Other programmers, including many of those that I have worked with, might write it like this:

```
number=1;
```

with no spaces, or even:

```
number = 1 ;
```

with a space before the final semi-colon. None of these styles is better than the others; the programming statement is perfectly clear in each case and it will obviously do the same thing in each case. What is crucial is that a programmer choosing any of these three styles should use the same layout in every similar statement. It should be obvious that a program that contained a section like this:

```
number = 1;
anotherNumber = 2;
aThirdNumber = 3 ;
```

is an example of very poor and inconsistent style. It's a bit like writing a section of a book in normal text, ANOTHER IN BLOCK CAPITALS, *and some more in italics* for no good reason. This does not look good, and does not give the impression that the programmer (or author) has paid attention and taken pride in their work.

For the moment it probably does make sense for you to follow the style of the programs in this book. The only exception to this would be if you were required to follow another for some reason; perhaps the course you are following expects you to use a particular style. Whatever you do, make sure that the style you follow is consistent.

---

1 Now, you might well be thinking that two people wrote the programs in this book, so what did they do about style? We negotiated, and arrived at a style that we were both reasonably happy with. We then both wrote the programs to conform to it. Sometimes programmers have to program to conform to another style, as we will see.

The third golden rule is:

Whatever style you use, make sure that it is clear and neat. Remember that someone may later have to change your program without your advice.

As well as being consistent, a programming style needs to be easy to read. In Java, remembering that the semi-colon is what marks the end of the line, the example statement above could be written:

```
number
=
1
;
```

or even:

```
number       =           1           ;
```

Hopefully it is obvious why these two are not good examples of style! Once again you are free to follow the style of the examples in the book, or you are welcome to develop and adopt the style to something that you prefer.

Sometimes software companies will require their programmers to write programs using a particular style, often called a "house style"[2] and enshrined in a hefty document labelled "Coding Standards". The reasons for this are sound; it ensures that all programs written in the company are consistent in style and so increases the chances that programs will be quickly understood by other programmers. Many programmers do not like working to such restrictive guidelines; programming is an essentially creative process and such strict controls reduce the chances for creativity. It is possible that you are required to write your programs to conform to some house style; if you do then you will have to adapt the programs in this book to fit the style.

The fourth golden rule is:

As you program, adopt and apply a consistent style. As you learn more, develop this style to something that you feel confident and happy with.

Indentation and layout

A particularly important aspect of a program's style is the way that it is laid out. In the same way as a page of a book or magazine can be laid out well or poorly, so can a program. A program's layout can add a great deal of meaning to a human reader. The layout is quite irrelevant to the compiler, which will simply ignore it, but it can be extremely useful to a human trying to read and understand the program.

Layout can emphasise which parts of the program serve a connected purpose; for example, blank lines might be left in a program to show the connection between certain lines or something about the program's structure. A crucial aspect of layout is *indentation*, where spaces are left at the start of lines. The indentation of a program can show a reader a great deal about the program's structure.

As an example, here is the definition of the simple *Duck* class from the previous chapter. At the moment you probably won't remember every detail of what

---

2 Effectively, you see, we adopted a house style for the programs in this book.

it does, but that's not the point. Just concentrate for now on the way that it's laid out. All you really need to remember at the moment is that the lines that start // and the sections marked off with /* and */ are just intended for the human reader; they are *comments* and are completely ignored by the compiler. Comments are used to explain how a program works; you should find that by reading them you are able to see what this Java defines even if you don't understand all the tricky little details of the Java itself.

```
/*
  Duck.java
  A simple Duck class for the "First Look" chapter.

  AMJ

  22nd January 2003
*/
  public class Duck
  {
    // Attributes

    private String name;
    private int age;
    private double value;

    // Constructor

    public Duck ()
    {
    }

    // Methods

    public void display ()
    {
      System.out.println ("The details of the duck are:");
      System.out.println (name);
      System.out.println (age);
      System.out.println (value);
    }

    public void setAge (int newAge)
    {
      age = newAge;
    }

    public void setValue (double newValue)
    {
      value = newValue;
    }
  }
```

Hopefully you will agree that this class definition is set out quite neatly. There are blank lines separating the parts of the definition that have different functions and generally separating the various lines of the definition. The indentation is also important; the first '{' and the last '}' (called *braces*) mark the start and end definition so all the statements inside the definition are indented by the same number of spaces (I usually prefer to use 2) to emphasise this.

So in this definition all the statements that make up the definition are enclosed by a pair of braces to denote its start and end points. Statements inside this (for example, those inside the methods) are indented by the same amount

again to reflect this. To make sure that everything really is clear the braces surrounding each method are aligned; the opening and closing brace in the `setValue` method are aligned, for example.

Now for an example of poor layout. Believe it or not this definition is just about identical to the previous example; the only difference is in the name of the class. The compiler would treat it exactly the same and it would behave identically if it were used in a program.

```
public class MangledDuck{private String name;private int age;
private double value;public MangledDuck(){}public void display()
{System.out.println("The details of the duck are:");
System.out.println(name);System.out.println(age);
System.out.println(value);}public void setAge(int newAge)
{age=newAge;}public void setValue(double newValue)
{value=newValue;}}
```

It is obvious that this definition is much harder to understand. There are no convenient blank lines to break it up into small chunks and there is no indentation to help with an understanding of the structure. There aren't even any comments to provide some clues.

Another point about layout is that all lines in a program should be kept reasonably short. The purpose of this is to make sure that the program will fit neatly onto a printed page. It is customary to make sure that no line in a program extends longer than 80 characters; the number 80 dates from the time when this was the maximum number of characters that could be displayed on one line of a terminal screen or printed across a page by a line printer. These days this practice remains a good one; most experienced programmers will always make sure that their lines are kept short, and many would stop at around 75 characters to be on the safe side. All the programs in this book (including both these in this section) have been formatted to make sure that all lines fit neatly on the page (sometimes this means that the lines are rather shorter than 75–80 characters).

When you write your programs always remember that you are not just writing them for yourself. You are also writing them for other programmers, and these are programmers who may one day have to change your program when you are not around to advise. The indentation and layout that you are going to use are of enormous help to others reading your program. Often when students ask me to find an error in their programs they seem to be amazed when the first thing I do is send them away to go through and correct their indentation; often this alone finds the error but even if not it always helps.

Always write your programs with good layout and indentation. Never be tempted to write your programs with no indentation intending to indent it all properly once the program works. This is simply something that will make more work for you and will make the job of anyone that helps you a great deal more difficult. In many ways it is as important that a program is neatly and consistently laid out as it is that the program works correctly.

You may well find that the Java system that you will be using will lay out your code neatly for you as you program; consult your *Local Guide* to see if you have this facility. If it will do, you should check the style that it will follow and you should decide if you are happy with the style. If you are not, spend some time finding out how to customise the style to your own preferences.

The fifth and final golden rule is:

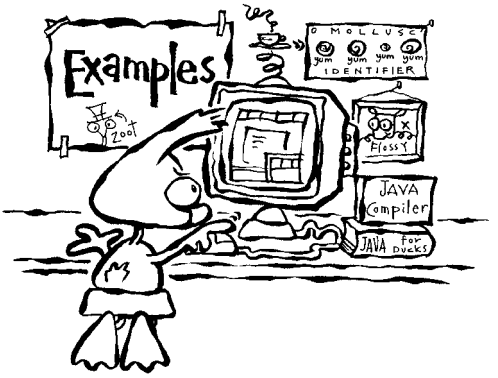
Always pay attention to layout and indentation as you develop your program, and never treat either as an afterthought.

Don't panic!

You will realise by now that programming is a complicated business. As well as producing a program that correctly carries out some useful task, a programmer must always produce a program that is neatly and consistently laid out. A program is only ever of any use if it can later be changed and adapted by other programmers.

Effective programming is all about applying a set of rules to a problem. When you become confident and proficient you will probably find that programming is an extremely rewarding task. A fully working program is a fine creation and something to be proud of. Programming itself is an extremely creative and enjoyable process. With neat layout and good style a program is very nearly a thing of beauty.

As you learn to program, remember that it is a structured process. It is a process that can only be carried out by someone who is calm and collected. As you learn, above all "Don't Panic!".



Example 1 – Cilla's poor style

As a new Java programmer Cilla is very concerned that her programs work. Bruce has set her a lot of exercises and Cilla has to work hard to complete them before the deadline. She decides to ignore the indentation of her program and intends to correct it all when the program works. Unfortunately it takes a long time to make the program work and Cilla doesn't have time to correct the indentation. She is somewhat dismayed when Bruce refuses to mark her program.

Why does Bruce behave like this? Is this reasonable?

This is very reasonable. Indentation is a crucial part of programming style and it is important that new programmers quickly get into the habit of producing correctly indented and well laid out programs. Cilla must remember that her programs may well be used and corrected by other programmers in the future

and her poor layout will make it very difficult for these programmers to discover how her program works. Bruce is quite correct to refuse to mark the program; the poor style will make it very difficult for him to see how it works or to spot any errors.

Cilla should realise that it is just as easy to create the program with proper indentation as it is to not bother. It is probably even easier, as the indentation can often help highlight errors before the program is even compiled. Leaving the indentation to the last minute is a very poor strategy.

Do not make the same mistake. Make sure that your programs are correctly and consistently indented as you write them. As you learn more Java, adapt your style and develop one that you can confidently apply consistently. Whatever you do, do not ignore style and do not treat it as an incidental extra that can be added once the program works.

Example 2 – Buddy's poor style

Buddy is enjoying learning Java but he finds typing difficult. So that he can learn more Java quickly he decides that he won't bother with any comments in his programs. He too is dismayed when Bruce refuses to mark his comment-less programs. Is Bruce right?

Bruce is quite correct again. Like indentation, comments should never be treated as an afterthought. They should be included in programs as the program is written to highlight the various stages that the program is passing through. Comments are crucial when another programmer comes to correct or maintain any program. It is always easier to add comments as the program is written; if they are added at the end, after a long period of development, the original programmer will often find that they themselves have forgotten how the program works!

You can probably understand to some extent what the class definition in this chapter does. You might not understand all of the statements, but you can still read the comments. Well-written comments should tell you all you need to know in order to use and even modify the program.

Again, do not make the same mistake as Buddy. Get into the habit of adding comments to your programs as you write them. But beware of adding too many.

Example 3 – Elvis's comments

Elvis is alarmed at Buddy's comment predicament and decides to add comments throughout his program. He adds comments to each line; many of the comments are quite long and eventually there are more comments than functional lines of program. Elvis is as dismayed as the others when Bruce says that his program is no better. What is going on?

You will not be surprised to learn that Bruce is quite right. A programmer who includes too many comments in a program is almost as bad as a programmer who includes none at all. A program with too many comments quickly becomes cluttered and very difficult to read. The key to good style is that the program is neat and easy to read and understand. Too many comments, and comments on lines whose function is obvious, can get in the way.

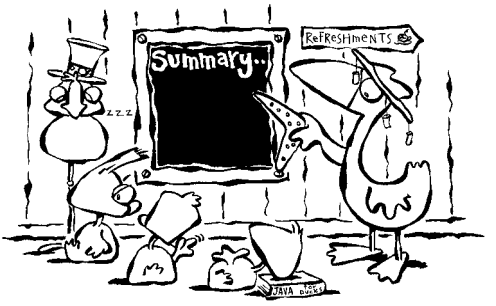
You may well find that as you become more and more confident in Java you will include fewer and fewer comments in your programs. You may be tempted at first to comment almost every line, and this is fine. As you learn more you will come to realise that this is not really necessary as there are some lines that every programmer will understand. Don't be afraid to let your style evolve and adapt.

7.1 Find another Java book and look at some of the example programs. See if you can pinpoint the differences between the style in that book and in this.

7.2 Find out if you are required for some reason to use a particular style as you learn to write Java. If you are, find out what the differences are between the style that you will use and the style used in this book.

7.3 Investigate any facilities that your Java system might have for automatically formatting your Java programs. Do you need to customise any of these features?

7.4 If you can, get hold of Java programs written by two or more different experienced programmers. What common features can you see in these two styles? Do you prefer one to the other? Even though you are still a novice programmer, how much of the programs can you follow? Does the style help? How about the comments?



Programming is a structured task that requires a structured approach. A programmer must build on a thorough understanding of a problem to produce a program; a program must never be developed before the problem that it is intended to solve has been fully analysed. A programmer must follow this process strictly.

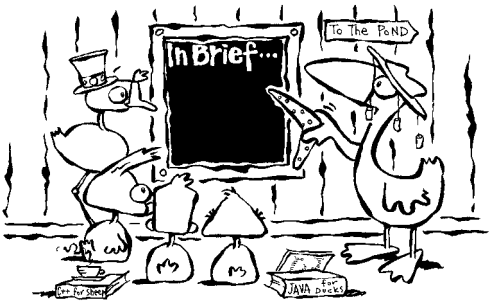
Good and consistent programming style is as important as a sensible structured approach. The guidelines for good style can be summarised in five golden rules:

1. *Work on one part of a program at a time. Get it right and then go to the next one. Never work on many parts of a program at once!*
2. *If you have added twenty lines to your program, compile it and check that your syntax is correct. Fix any errors before you carry on.*
3. *Whatever style you use, make sure that it is clear and neat. Remember that someone may have to change your program without your advice.*
4. *As you program, adopt and apply a consistent style. As you learn more, develop this style to something that you feel confident and happy with.*
5. *Always pay attention to layout and indentation as you program, and never treat either as an afterthought.*

Remember these rules as you learn to program. If you break them you will be making more and more trouble for yourself.

Now, finally, the time is right for you to take a serious look at some Java and to write your first program.

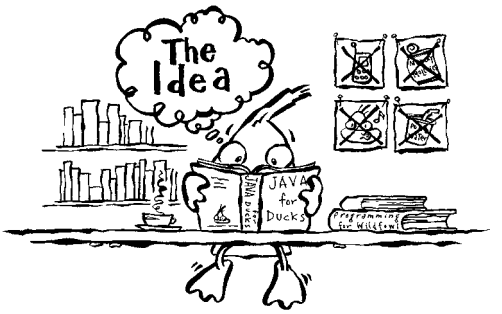

Chapter 8 - The Basics
boolean
true
int
Elvis
String
double
char



This chapter is the first one that looks in detail at some Java. You should now understand what a program is and you should have some idea of how to analyse a problem area and design a solution that will solve a particular problem. You should also know how to create, compile, and execute the programs you write. Now it's time to actually write some!

A Java program is basically nothing more than a set of *statements* that instruct a computer to perform some task. At the simplest level a program takes some values and processes them to produce some other values. In this chapter you will meet a full Java program in detail for the first time and will see how it stores and processes these values. The next chapter will go on to explain how to display the values and how to ask a user to enter the values. Together these two chapters cover the most basic tasks that can be carried out by a Java program.

After reading this chapter you should understand the basic structure of a Java program. You should understand how values can be stored in *variables* and how they can be manipulated. You should then be able to write some simple Java programs.



Let's start by looking at a very simple Java program. You have seen some simple Java definitions and programs in the previous chapters and looked at what some of the statements might achieve, but the focus there was just on the layout and style and an overall view. Now it's time to start understanding what the Java statements actually do. Still, this program won't do anything especially useful just yet; the thing to concentrate on this time is just its general structure. Here it is:

```
/* First.java - a very simple Java program.
   Demonstrates the basic structure of a Java program.
```

```
  Author   : GPH
  Date     : 14th December 2002
  Platform : Linux (Red Hat 7.3), JDK 1.4.1
*/
public class First
{
   public static void main (String args[])
   {
     int aNumber;
     aNumber = 0;
   }
}
```

This is the *source code* of a program.

The first six lines in this program, enclosed between `/*` and `*/` are *comments* (you saw some in the program in the previous chapter). The compiler ignores these lines; their only use is for a human reading the program. In this case they provide the name of the file containing the program, give a brief description of the purpose of the program, and tell us who the author was and when and on what system the program was written.

This is the *header block* of the program, a collection of comments that provides basic but important information about the program. A header block should as a minimum tell the reader who wrote the program, what the program does, why it was written, where it was written, and when it was written. As a program is developed, changed, and maintained the header block should record the same information about all the changes. This will then provide all the information a human would need in order to understand how and why the program was originally written and how and why it has been changed since.

More generally, comments are used for many purposes; common uses are to explain parts of programs (especially complicated parts) and to record modifications to parts of a program. Their purpose is only ever for a human reading the program.

The next line:

```
public class First
```

marks the start of the definition of a class called *`First`*. Here *`First`* is the name of the program (or the name of the single public class defined in the program); the braces immediately after this line mark out the definition of the attributes and methods that this class has.

In fact this class has only one method and no attributes. The method is called `main`, and the start of its definition is the line:

```
public static void main (String args[])
```

This marks the start of the program. Once the program has been compiled, this is the part of the program that the computer will execute first, the `main` method. There's a lot going on in this line, and more detailed explanations will have to wait until you've learned a lot more Java, but briefly:

- `public` declares that this method is available to anything that might want to call it;
- `static` declares that this method can be called without requiring an instance of the class it is defined in (so, for example, you can call a static method

of a class *Duck* without needing to create a *Duck* object first). In the course of this book, the only static method you will meet is `main`;

- `void` declares that the method doesn't return a value;
- `main` is the name of the method;
- `(String args[])` is the single parameter of the method (in this case it happens to be a list of values that are typed on the command line when the program is executed).

Occasionally, you may see:

```
return;
```

as the last line of a void method such as this. This usually occurs where a crusty old C or C++ programmer writes a Java program, and is perfectly acceptable, but is regarded as bad style by a good many programmers. The `return` statement simply tells the JVM to terminate execution of this method here and return control of the program to the code that called the method. The closing brace of a void method does this automatically, so there is no real need to include a `return` statement as well.

You can treat this program as a template for your own programs. All of the lines explained so far (or lines very similar to them with the name of the program changed) will appear in every program you write. In fact it's a good idea to keep them in a handy file that you can use over and over again, so that all you need do is change the comments at the top to explain a new program. It's also a fine idea to complete the parts of the header block in this file, such as your name, that will not change whatever the program you are writing.

The rest of the program is the *body*, or the definition of the `main` method. The statements that make up body of the program (in this case the program and the body of the `main` method are effectively the same thing) are contained between two braces, `{` and `}`, just like the definition of the class. The programmer has indented the statements between the braces to emphasise that they are enclosed within them. This layout is ignored by the compiler, but is very helpful for the human reading the program. You have seen the importance of layout and style in the previous chapter.

There are two lines left. The first:

```
int aNumber;
```

is a *variable declaration*. This line tells the compiler that the program uses an integer value and that its name, more correctly called its *identifier*, is *aNumber*. Such values are called *variables* because the value they hold varies (or has the potential to vary) as the program is executed. Any program that does anything even remotely useful will use a lot of variables.

The second line in the program body is:

```
aNumber = 0;
```

This is an *assignment statement*. It simply assigns the value on the right-hand side of the = sign to the variable on the left-hand side. So here the variable *aNumber* is assigned the value 0. The = sign is not being used here in the sense that you are probably familiar with from mathematics or algebra; it is not a comparison but an assignment. It's probably helpful to get into the habit of

reading = used in this way as "becomes equal to" or "is assigned" rather than just "equals".[1]

These statements illustrate the two fundamental new ideas in this chapter. Variable declarations and assignment statements are two of the most fundamental ideas in programming. Programmers need to declare the variables that their programs will use and they need to be able to assign them values. The following sections cover these ideas in more detail.

Variables and types

A variable is a value used in a program. It is created in a *declaration* at the top of the program where it may also, if required, be given an initial value. The value then changes as the program progresses, as the statements in the program manipulate the value. A variable's final value is sometimes one of the outputs of the program. A variable has an *identifier* and a *type* of value that it can store. The variable's declaration tells the compiler what these two things are. For example:

```
int aNumber;
```

declares a variable with the identifier *aNumber* that can store integer values (in Java, `int`). It is good practice to provide an initial value for a variable at the same time as it is declared.[2] This assignment is simply added to the declaration. So, for example, to assign *aNumber* an initial value of 0 the declaration is extended to:

```
int aNumber = 0;
```

Declaring the type of a variable allows the compiler to allocate an appropriate amount of the computer's memory to store the variable's values, and allows the compiler to check that sensible operations are being carried out on values in the program. Java has many data types, but for the moment just a few of these will be quite enough:

`int`	an integer value or whole number – 0, 1, 2, −1 and so on.
`double`	a (double-precision) floating-point (decimal) number – 1.415, 1.25.
`char`	a single character – 'a', 'Z', '$'.
`String`	a sequence of characters – "hello", "Walrus", "Gumboot", "%$%!%%".
`boolean`	a Boolean value – either true or false.

There is an important distinction between the ways in which characters and strings are represented. Characters are enclosed by single quotation marks but strings are enclosed with double quotation marks. This means that:

```
'a'
```

is the single character *a*, with the type `char`, whereas:

```
"a"
```

is a string containing just one character with the type `String`. This distinction is subtle, but will become important. Take special note of the capital `S` on

1 We will in fact see a different Java symbol that is read as "equals" later on.

2 This is sometimes not necessary but it is a good habit to get into. If in doubt you should always assign the variable an initial value.

`String`; this is not a misprint! The reason for this important distinction will become clear later on when we look at strings in more detail.

A variable can only ever hold values of its own declared type. It is an error to assign, say, an integer value to a string variable. The reason for this should be quite obvious; the compiler has allocated the variable the right amount of memory to store an integer and it's very unlikely that a string will fit in the same space.

Identifiers

The identifier chosen for a variable is pretty much irrelevant to the computer. The computer can't understand the language used for the name and certainly can't learn anything about the variable's purpose from it. However, the identifier is certainly very important for the programmer and for anyone else reading the program. It is essential that the name chosen should neatly describe the purpose of the variable. At the same time, the name should not be so long and descriptive that the program becomes unwieldy and hard to read. A programmer must strike a delicate balance between descriptiveness and brevity.

As an example, suppose a simple program was used to monitor the birds at a pond and that as part of this the program stored the number of ducks that could be seen. Here are some possible identifiers for that variable and some comments on them:

geese	This is not a good name. In fact it's a very bad name because it's positively misleading!
x	At least this name isn't misleading, but it still doesn't tell the reader much about what the variable stores. This is poor.
thenumberofducksonthepond	This name tells the reader what the value is but it's very hard to read. It's better than the last two suggestions, but a program using this would be an example of very bad style. It would also require a lot of typing!
ducks	This is better still but it still doesn't really tell the reader what the value is.
numDucks	This is the name we would use and is the best of these suggestions. It's short enough to keep the program short and neat and it tells a reader what the value is. *numOfDucks* would be a reasonable alternative.

Choosing identifiers requires care and thought. The ones that you choose must provide some information for people reading your programs. Also, you might come back to a program after many months and have to understand it quickly. Bearing this in mind, identifiers must also not be so long or unwieldy as to make the program unreadable. There is a balance to strike.

Most programmers have a convention that they use for their identifiers, or sometimes they will be asked to use one by their employer as part of some prescribed standards. Like programming style, the choice of identifiers is

a personal thing and you will probably develop your own style for doing this as you gain more experience.

Our style is to use a single word if possible or otherwise to combine two or three words and to capitalise the first letters of any words after the first (hence `numDucks` in the example). To save typing and to keep the program short and readable we abbreviate where the abbreviation is obvious; so number can become "num", value becomes "val" and so on. That's the convention we will use in the programs in the rest of this book. You're free to copy it, to follow your own convention, or to follow some other system that you might be required to use. Whatever you do, think about the identifiers you choose and use a consistent system for choosing them.

Declaring variables

Now it's time for a more detailed look at variable declarations. The basic form of a declaration is:

<type> <identifier>;

The declaration specifies the type of the variable (the type of value that it can store) and states its identifier (its name). The line finishes with a semi-colon. If the variable is to be given an initial value the declaration can be extended:

<type> <identifier> = <initial value>;

Here are some examples with comments to explain them:

```
int grade;    // an integer to hold a grade, no initial value
int grade = 0; // an integer to hold a grade with initial value 0
char answer;  // an answer to something - one character
String name;  // a string to hold a name
String name = "Gumboot";  // as above, with an initial value
boolean finished;        // to hold "true" if finished
boolean finished = false; // as above, but with an initial value
```

If more than one variable of a particular type is to be declared in a program, they can both be named on the same line:

```
int aNumber, anotherNumber;
```

This version is used especially when the two variables have closely connected purposes. It is identical in meaning to two separate declarations:

```
int aNumber;
int anotherNumber;
```

The choice between the two is largely a matter of the programmer's style and preference.

Values and variables

Variables aren't much use if a programmer can't give them values. Variables are assigned values in an *assignment statement*. The general form of such a statement is:

<identifier> = <value>;

The left-hand side of the = sign is the name of the variable that we want to assign a value to and the right-hand side is the value. The effect is that the value of the variable whose identifier is on the left becomes equal to the value that is

on the right. For example:

```
// Declarations
int grade;
char answer;
String name;

// Assignments
grade = 70;
answer = 'a';
name = "alan turing";
```

These three statements assign the three values on the right to the variables named on the left. In each case the variable has to have the same *type* as the value assigned to it; it doesn't make sense to assign a string value to an integer variable. As examples of what cannot be done, assuming the types in the declarations above, these assignments are not allowed:

```
grade = "a*";      // cannot assign string to int
name = 100;        // cannot assign int to string
answer = "hello"; // cannot assign string to char
```

Unlike the algebraic use of = it is not possible to write assignment statements with the variable's identifier on the right. This is not allowed and will cause a compilation error:

```
'a' = grade;       // Not allowed!
```

To make sure that you remember this, remember to get into the habit of reading the = sign in an assignment statement as "is assigned" or "becomes equal to".

In all the correct assignment statements so far the right-hand side of the assignment has been a simple value, called a *literal value*. It is probably more common for it to be an *expression*. An expression is something that calculates a new value. For example:

```
grade = 70 + 2;              // grade is 72
name = "ada " + "lovelace"; // name is "ada lovelace"
```

These expressions can also contain variables:

```
int birds;
int geese;
int coots;

geese = 50;
coots = 100;

birds = geese + coots; // birds is assigned 150
```

Finally, the variable that appears on the left-hand side can also appear on the right-hand side:

```
int ducks;
ducks = 50;

ducks = ducks + 1; // ducks is now 51
```

You might remember that programming statement that Tony could never understand all those years ago! The one we mentioned long ago in Chapter 3:

```
50 LET X = X + 1
```

This is, of course, an assignment statement (actually in the BASIC language, but it's very similar to the Java equivalent). Its effect is to add one to the value of the variable *X* (sadly, most versions of BASIC only allowed single character variable identifiers).

The important thing to remember, especially when the same variable appears on both sides of the assignment statement, is that the value of the expression on the right is calculated first and the result is then assigned to the value of the variable named on the left. This means that a statement such as:

```
aNumber = aNumber + 1;
```

has the effect of adding one to (called *incrementing*) the value of *aNumber*. The right-hand side is calculated first, and the result is then assigned to *aNumber*.

Initialising a variable

Now we know what a variable is, and how to declare one, we need to know how to go about assigning an initial value to it. This is called *initialisation* and is very important in programming. When a variable is declared within a class definition, it will be assigned a null value (that is, for example, *null* for a `String` or `char`, 0 for an `int`, 0.0 for a `double`, false for a `boolean`). When a variable is declared within a method, it has no initial value, and any attempt to use that variable before its value is set will result in a compiler error along the lines of "this variable may not have been initialised".

To recap, the declaration of an integer variable looks something like this:

```
int number;
```

and to assign it a value, we use the following:

```
number = 0;
```

This operation is so common that the two statements can be combined. This gives the extended form of a declaration that also includes an implicit assignment statement:

```
int number = 0;
```

This single statement has the identical effect to the two lines above.

Never assume that your variables have a value just after they have been declared. Get into the habit of initialising them. Wherever possible, declare and initialise a variable in a single statement, which reduces the likelihood that you will later attempt to use an uninitialised variable. Finally, never try to use a variable before it has been initialised, unless you are assigning an initial value to it, of course!

Operators and their precedence

Some of the examples of expressions above used the addition operation "+". Expressions in assignment statements can also contain all the other usual arithmetic operations (plus, minus, times, divide) that you might expect. The symbols might not be quite what you are used to in arithmetic. They are:

- `/` Division
- `*` Multiplication
- `+` Addition of numbers or Concatenation of strings
- `-` Subtraction

When more than one of these is used in a single expression the operators have a *precedence* that determines the order in which they are executed. This is the same as you might be used to in arithmetic or algebra. The order of precedence for these four operators is: / and *, then + and −. In other words, division and multiplication are carried out before addition or subtraction.

The idea of an order of precedence often catches new programmers out and does indeed take a fair bit of getting used to. Look at the following statement:

```
int number = 2 + 8 / 2;
```

You might think that *number* is assigned the value 5. You might expect that 2 is added to 8 to give 10 which is divided by 2 to give the answer. Not so! The value *number* is actually assigned is 6. Division happens first (it has a higher precedence), so 8 is divided by 2 to give 4 and that result is added to the first 2 to give 6.

Since this can be confusing, brackets can be added to the expression to make things clearer. Again, this is the same as in algebra – the part of the expression in brackets is worked out first and then the precedence of the operators takes over. In other words, brackets have a higher precedence than the arithmetic operators. The statement above could be written:

```
int number = 2 + (8 / 2); // number is assigned 6
```

to emphasise that the 8 is to be divided by 2 first. If this is not what is wanted, it could be written:

```
int number = (2 + 8) / 2; // number is assigned 5
```

which changes the expression to make the addition happen first. In this case this also changes the result of the expression.

It is good practice to always include brackets in expressions even when the precedence of the operators would achieve the desired result. It makes the programmer's intention clearer and removes the possibility that other programmers will suspect a mistake!

Some shorthand

A common operation that happens in almost all programs is to add one to or *increment* the value of a variable holding a numeric value. For an integer variable the assignment statement for this is:

```
number = number + 1;
```

This is so common that Java provides a shorthand form for this. A variable can be incremented using the ++ operator.[3] An integer variable can be incremented using this operator:

```
number ++;
```

This has exactly the same effect as the longer form:

```
number = number + 1;
```

The only advantage of using this shorthand is in the amount of typing the programmer has to do. The operation is so common that it does save a lot of typing.

---

3 This operator explains why C++ is so-called. It is after all C with some extra things added.

Not surprisingly there is similar shorthand for taking one from (called *decrementing*) a variable's value:

```
number --; // same as number = number - 1
```

Sometimes a variable must be incremented by more than one, and so another shorthand is available than allows the programmer to add a value other than one to a variable:

```
number += 2; // adds two to number
```

Equivalents exist to subtract a value, multiply by a value, or divide by a value:

```
number -= 3; // takes 3 off number
number *= 3; // multiplies number by 3
number /= 3; // divides number by 3
```

Remember that the only purpose of these shorthand forms is to save typing. You can always write out the statement in full if you prefer, and many new programmers prefer to do so.

Casting and division

3 divided by 2 is sometimes 1. Really.

Java provides several data types for storing numeric values. The two described so far are `int` (integers, whole numbers) and `double` (double-precision floating-point numbers). These two types of value are clearly both numbers and thus are the same sort of thing; it's often useful to be able to assign values of one type to the other. For example, you may think of using the following:

```
int number;
double cost;

cost = 12.0; // .0 indicates a floating-point value
number = cost;
```

to assign the value of *`cost`* to the value of *`number`*.[4] This piece of program generates a compiler error, though. The error will probably differ depending on which platform you try to compile the code on, but will be along the lines of "possible loss of precision". This is because the fourth line above would cause the decimal part of the double-precision number to be lost during the assignment to an integer value. Thankfully, help is at hand.

There is a way around the loss of precision problem, known as *casting* from one type to another. In other words, it is possible to force the compiler to convert a variable of one type into a different type. As an example, we can cast from `double` to `int` to achieve the effect we wanted above:

```
number = (int) cost;
```

So the type we want to convert to is placed in brackets before the variable we wish to store its value. In this example, *`number`* now has the value 12, and in effect there is no loss of a decimal part.

---

4 Note that the .0 on the value of `cost` isn't really necessary. It's included it here to clearly show that this is a floating-point value. Pedantically, it's a double-precision floating-point value.

But suppose that *cost* had the value 12.75. The integer variable *number* can't store the decimal part, and in this case it would be lost. This issue crops up most often in division. The thing to remember is:

an int divided by an int is an int and nothing but an int

This means that any decimal part of the result is lost. For example:

```
int number;

number = 5 / 2; // number is 2
number = 1 / 2; // number is 0
```

It might seem that this should not be allowed to happen and that any case where it does it an error. The fact is that sometimes this result can actually be what it required; it is actually a potentially useful result, the number of times the right-hand side of the division "goes into" the left-hand side. You might remember "remainders" in division; all that's happening here is that the remainder is being lost. Since this is likely to be correct some compilers will not provide a warning of a possible error.

Statements could be written to calculate the result of a division including the remainder:

```
int number;
int remainder;

number = 5 / 2;               // 2 goes into 5 twice
remainder = 5 - (number * 2); // remainder is 1
```

An easier way to get the remainder of a division calculation would be to use the modulus operator (%). This works as follows:

a % b gives the remainder for a / b.

So in the above example :

```
remainder = 5 % 2;
```

Be careful when combining integer and floating-point values in your programs. If your programs are processing a lot of integer and floating-point values and are giving you incorrect results, always check for problems with integer division.

If you are dividing two integer values and you want the result to be a floating-point value you can force the result to be a float by using a cast to convert one of the integer values to a floating-point value, like this:

```
int aNumber = 12;
int anotherNumber = 5;
double result;

result = (double) aNumber / anotherNumber;
```

This would assign float the expected floating-point result, in this case 2.4. If you find yourself having to do a lot of this it's probably time to consider whether the integer values shouldn't really be floating-point values.

An alternative, less elegant way to convert an integer value to a floating-point value is to multiply the integer by 1.0:

```
result = (aNumber * 1.0) / anotherNumber;
```

This technique can be useful at times, but a cast is probably usually to be preferred.

Constants

Variables by their very definition vary. Sometimes a program needs something that behaves in the same way as a variable but that doesn't vary; this is unsurprisingly called a *constant*. Constants are declared in exactly the same way as variables except that the word `final` is added to the start of the line, and that the assignment of the value may only be performed once at most. So, to declare a `final` variable of type `int`:

```
final int MAXTURNS = 10;
```

It is important to note here that we can declare `final` objects, but that these may still be changed. This sounds paradoxical, but is quite simple to illustrate. If a class provides methods to alter its attributes (the so-called *set* methods, or *mutators*), then, even though an object is declared `final`, it can still be altered by calling one of these methods. However, declaring an object to be `final` does mean that it can only be initialised once, so an object variable can only refer to a single object for the duration of the program.

The same guidelines apply for choosing identifiers for constants. Many programmers also choose to give them an identifier that clearly shows that they are not allowed to change. The convention that is often used (and that is used in the programs in this book) is simply to write the name in CAPITALS. Capitalising the initial letter is also a possibility, but this can lead to clashes with existing built-in Java classes if done without careful thought.

Constants have two main uses. Sometimes they are used just for readability. Suppose a program made a lot of use of the mathematical constant *PI*. A programmer could write:

```
final double PI = 3.1415;
```

at the start of the program and then later use the constant in expressions such as:

```
area = PI * radius * radius;
circumference = 2 * PI * radius;
```

This makes the program much more readable. It also saves typing if the constant value is used in many places in the program and ensures that the value used is always precisely the same. It should be obvious that the statements above are far preferable to:

```
area = 3.1415 * radius * radius;
circumference = 2 * 3.1415 * radius;
```

and that this a fine way to avoid errors or inaccuracies that might be caused by:

```
area = 3.14 * radius * radius;
circumference = 2 * 3.1415 * radius;
```

Another use of constants is when a particular value is used in many places in a program and might need to be changed later. The value will never be changed while the program is running but it might change later for some other reason. Examples might be the number of games in a football season, the number of some item packed into a box, or the number of attempts allowed in a game. It's far easier to declare that value as a constant; if it changes you only have to change it in one place.

Output

Up to now, we've learned how to write exciting programs, which might even do some useful things. However, I'm sure you'll agree that these things are neither exciting nor useful unless we can see what's going on. As it stands, you know how to create a variable, assign it a value, and manipulate this value. However, so far you've had to assume that the manipulated value is, in fact, whatever I say it is. Therefore, this would seem to be a good time to show you how to output messages to the screen.

In most books about programming or about a programming language the first program the reader sees is always the same. This first program always does nothing more than display the message *`hello, world`* or something very similar on the user's screen.

This is such a fine and longstanding tradition that it would be a shame to not to include this memorable program somewhere in this book, even if it isn't the first complete program. So here is my version of that very program in Java:

```
/* Hello.java - the traditional first program
                demonstrating output.

   Author  : GPH
   Date    : 14th December 2002
   Platform : Linux (Red Hat 7.3) with JDK 1.4.1
*/

public class Hello
{
  public static void main (String args[])
  {
    System.out.println ("hello, world!")
  }
}
```

Again, this program defines a single method, and no attributes. You will recognise all the lines in it except one – line 13 – and you might even remember that from Chapter 6. This line actually uses a method already defined for you, in order to output a string to the screen. This method is called `println` and is defined as part of an object called `System.out` which we won't worry about too much, except to say that it almost always corresponds to the terminal from which the program was run. This method takes a single `String` parameter, which will be output to the screen, followed by a newline character.

Now, you may be wondering how on earth we can have single characters, integers or floating-point numbers output, if the method we're using will only accept a `String` as a parameter. Thankfully, the people who developed Java are clever, and thought of this. They designed Java so that `char`, `int` and `double` values can be added to the end of `String` variables automatically. This is also known as appending a value to a string.

So, for example:

```
int numWheels = 4;

System.out.println ("There's " + numWheels + " wheels "
                    + "on my waggon.");
```

will produce the following output on the screen:

```
There's 4 wheels on my waggon.
```

If you need to do any sums in the middle of an output statement, you should put the sums inside parentheses just to make it clearer what you're trying to do. In fact, this is usually only a problem if the sum is an addition. Consider the following variable assignments:

```
int numDucks = 4;
int numCoots = 2;
```

And the following two, subtly different, output statements:

```
System.out.println ("There are " + numDucks + numCoots
                    + " birds here.");

System.out.println ("There are " + (numDucks + numCoots)
                    + " birds here.");
```

Here we have a potential problem. In the first output statement, the values of *numDucks* and *numCoots* are appended to the output string one after the other, while in the second statement, the values are added together, and the result of this addition is appended to the output string. So the first statement outputs a nonsense figure (42), while the second statement gives the correct figure (6).

Finally, you may sometimes wish to output information to the screen, but without the newline character at the end. Thankfully those clever Java people thought of that too, and provided another method with `System.out`, called `print`. So, as a last example for this chapter:

```
System.out.print ("This is a line.");
System.out.println ("This is another line.");
```

will produce the output :

```
This is a line.This is another line.
```

Object objection!

The programs in this chapter are all very well and good. You will find that you can type them in using your Java system, and you will be able to run them. But there is something missing. Java is an object-oriented programming language; where have the objects gone?

It is true that the programs that you have seen in this chapter (so far!) have had no objects in them. In these examples Java has effectively been used without objects, as a procedural language. This is fine and allowed, and is sometimes what is needed. The real power of using Java comes when objects are used, however.

Let's look at the traditional first program once again, but this time with some added objects. An application is needed here – we need something in the "real world" to model as an object – so we'll decide that Bruce has recently installed a new illuminated sign outside his hut. He plans to use it to display interesting messages to all the other birds. Obviously his first step is to find out whether his new sign works properly, and displaying "hello, world!" on it would seem to be a good start.

The first step here is to model Bruce's sign as an object. More accurately, we need a model of a class that can represent all illuminated signs, and Bruce's sign will be an instance of the class. We need to identify the type's attributes and methods.

There is one obvious attribute, the message that is displayed on the sign. This is a sequence of characters, or a `String` in Java. There are no other attributes that we need to worry about at the moment.

There are two obvious methods. First, Bruce will need some sort of way to set the message that is to be displayed. This would presumably involve typing it in. There will also need to be some way to activate the sign; there must be some way for Bruce to confirm that the message is correct and that the sign should start displaying. These two methods might be called *`setMessage`* and *`display`* respectively.

There will be one other method, since the class will need a constructor. This will provide some sort of initial value for the message; you could think of it as Bruce turning the sign on.

The class might be called *`Sign`* and the definition would be created in a file called *`Sign.java`*. The first stage would be to add the single attribute:

```
public class Sign
{
    // Single Attribute – What to Display
    private String message;
```

Now the methods are added. The first, the constructor, has the same name as the class and just sets an initial value for the attribute – an empty string will do.

```
public Sign ()
{
  message = "";
}
```

The other two methods are slightly more complicated, so let's take them one at a time. The first sets the message that is to be displayed. It will need a single parameter representing the message (a `String`). There is no need for it to return a value, so it can be a void method. All it does is take the value from the parameter and copy it into the attribute, so:

```
public void setMessage (String newMessage)
{
  message = newMessage;
}
```

The other method displays the message. It needs no parameter and does not need to return a value. For the moment we'll assume that displaying the message in the usual default place will be enough, so:

```
public void display ()
{
  System.out.println (message);
}
```

And that's enough to define the class. Some comments should be added, of course, to give the final definition.

```
/* Sign.java

   Bruce writes the traditional first program, but this
   time with objects.
```

```
    Author : AMJ
    Date   : 13th December 2002
*/

public class Sign
{
  // Single Attribute - What to Display
  private String message;
  // Methods
  public Sign ()
  {
    message = "";
  }
  public void setMessage (String newMessage)
  {
    message = newMessage;
  }
  public void display ()
  {
    System.out.println (message);
  }
}
```

This is just a definition. You could type it in and Java would happily produce a `.class` file, but it wouldn't actually do anything. A `main` method is needed. We'll add a `main` method into the class; this is quite a common way of providing a basic program to check that all the parts of the class work as expected. Such programs are sometimes referred to as *driver* programs.

The main method will have to do three things:

1. Create Bruce's sign (as an instance of the *`Sign`* class).
2. Set the sign's message to "hello, world".
3. Display the message.

These steps conveniently correspond to the three methods. The first is the constructor, then *`setMessage`*, and finally *`display`*. There's more about using (properly called "calling") methods later on in the book, but here's a quick call to the constructor for this class:

```
Sign brucesSign = new Sign ();
```

We know that this constructor sets the sign's message to an empty string. The next step is to use `setMessage` to set it to something more useful. The syntax for calling a method on an object is:

<object name>`.`*<method name>* `(`*<parameters>*`);`

This method has one parameter, so this will do the job:

```
brucesSign.setMessage ("hello, world");
```

The call to the method to display the message is similar:

```
brucesSign.display ();
```

And that's all there is to it. All that remains is to include the headers for the `main` method to give a complete program.

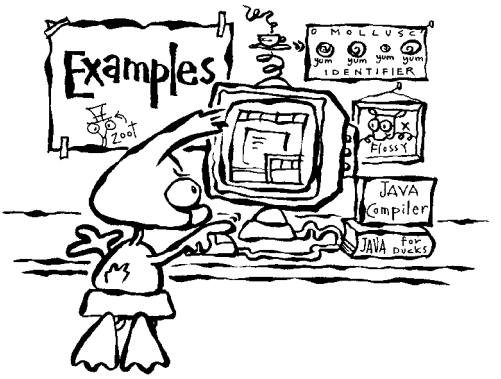
```
/* Sign.java

   Bruce writes the traditional first program, but this
   time with objects.

   Author    : AMJ
   Date      : 13th December 2002
*/ Tested on : Linux (Red Hat 7.3) with JDK 1.40

public class Sign
{
  // Single Attribute - What to Display
  private String message;

  // Methods

  public Sign ()
  {
    message = "";
  }

  public void setMessage (String newMessage)
  {
    message = newMessage;
  }

  public void display ()
  {
    System.out.println (message);
  }

  public static void main (String[] args)
  {
    // Create an object for the sign
    Sign brucesSign = new Sign ();

    // Set the Message...
    brucesSign.setMessage ("hello, world!");

    // ...and display it.
    brucesSign.display ();
  }
}
```

Some programmers would complain that this program is overly complicated and that there is no need to use objects in this simple case. Others would say that all Java programs should use objects, and so all Java programs should be written like this. They might well fight over it.

One advantage of this approach is that the class can be used in other applications. As we look at more Java we'll expand Bruce's class until he has absolutely the best sign on the pond.

The basics

Even though there are not that many new ideas here, there is a lot to understand in this chapter. This is also the most important chapter in this book. Everything that follows has to assume that you understand how to use and declare variables and constants. Now, try the examples and exercises …



Example 1 – Mr Martinmere's reserve poser

Mr Martinmere wants a computer program to help him keep details of the birds living on the reserve. He wants to know how many birds there are in total and what percentages are ducks and coots; on the reserve at the moment there are many ducks and many coots. What variables might his program use?

Taking things one at a time, the program will need variables for the total number of ducks and the total number of coots and, perhaps, the total number of birds. These values are all integers, but we will have to remember that the percentages will most likely be floating-point numbers.

Some of the variable declarations might look like this:

```
int numDucks;
int numCoots;
int numBirds;
```

In the program itself there would be a line that calculated the total number of birds at the reserve:

```
numBirds = numDucks + numCoots;
```

There would also need to be statements to calculate the percentages of ducks and coots, given the total number of birds at the reserve. It would be prudent to include some variables to store these percentages!

```
double duckPercentage;

duckPercentage = (double) numDucks / numBirds * 100;
```

It would also be reasonable to have a variable *cootPercentage*, along the same lines as *duckPercentage*, but since we are only counting ducks and coots, it makes sense in this case to calculate the percentage of coots as:

```
100.0 - duckPercentage;
```

Finally, Mr Martinmere would want to see the results of the calculations, so it would be wise to include some output statements. Here is the complete program:

```
/* BirdCalculator.java - Calculates numbers and percentages of
                         ducks and coots on the reserve.
   Author    : GPH
   Date      : 30th December 2002
   Tested on : Linux (Red Hat 7.3) with JDK 1.4.1
*/


public class BirdCalculator
{
  int numDucks;
  int numCoots;

  public BirdCalculator () {}

  public void setNumDucks (int d)
  {
    numDucks = d;
  }

  public void setNumCoots (int c)
  {
    numCoots = c;
  }

  public void printInfo ()
  {
    int numBirds = numDucks + numCoots;
    double duckPercentage = (double) numDucks / numBirds;
    System.out.println ("There are " + numDucks + " ducks and "
                        + numCoots + " coots on the reserve.");
    System.out.println ("That equates to " + duckPercentage +
                        "% ducks and " + (100.0 - duckPercentage)
                        + "% coots.");
  }

  public static void main (String args[])
  {
    BirdCalculator bc = new BirdCalculator ();

    bc.setNumDucks (24);
    bc.setNumCoots (15);

    bc.printInfo ();
  }
}
```

Example 2 – Elvis's pocket money

Elvis wants to use a computer program to keep track of his pocket money. He wants to know how much he's got left at any time and so he decides that he needs to record

how much he receives and how much he spends. What variables is the program going to need?

Money is a floating-point value,[5] so all the values in Elvis's program would be declared as *double*. There would be a variable for the amount of money that he's currently got and another to store anything he spends. These would be used in a suitable expression to update the amount he's got after he spends some.

The amount that he receives is unlikely to change very often, but it will probably change sometimes, so a constant would be used for that. The declarations might look like this:

```
final double weeklyPocketMoney = 15.00;

double pocketMoney;
double amountSpent;
```

In the program there might be assignment statements like this:

```
pocketMoney += weeklyPocketMoney;
```

This is of course equivalent to:

```
pocketMoney = pocketMoney + weeklyPocketMoney;
```

Another assignment would alter the amount Elvis has after he has spent some:

```
pocketMoney = pocketMoney - amountSpent;
```

Remember that this last statement could also have been written:

```
pocketMoney -= amountSpent;
```

See how this shorthand saves typing! Here's a full example:

```
/* MoneyTracker.java - A simple pocket-money tracker
   Author    : GPH
   Date      : 30th December 2002
   Tested on : Linux (Red Hat 7.3) with JDK 1.4.0
*/

public class MoneyTracker
{
  private final double WeeklyAllowance = 15.00;
  private double pocketMoney;

  public MoneyTracker (double amount)
  {
    pocketMoney = amount;
  }

  public double getAmount ()
  {
    return pocketMoney;
  }
```

5 An issue here is that a `double` in Java can store many places of decimals, but money values only ever have two decimal places. This can lead to errors, but only when a great many values and calculations are involved – we'll assume that Elvis doesn't get enough pocket money to make this a serious problem.

```
  public void spend (double amountSpent)
  {
    pocketMoney -= amountSpent;
  }

  public void addAllowance ()
  {
    pocketMoney += WeeklyAllowance;
  }

  public static void main(String args[])
  {
    MoneyTracker mt = new MoneyTracker (13.50);
    double amountSpent = 5.30;

    System.out.println ("Money left from previous week : £"
                        + mt.getAmount ());

    mt.addAllowance();
    System.out.println ("After this week's allowance : £" +
                        mt.getAmount ());

    mt.spend (amountSpent);
    System.out.println ("After spending £" + amountSpent +
                         " this week, there is £" +
                          mt.getAmount () + " left.");
  }
}
```

In this example the constructor of the class takes a parameter (representing the amount of money that Elvis has), this is the first time that we've seen this, so take a close look!

Example 3 – Cilla's cricket poser

Cilla and her friends all like cricket. Because he likes to keep all the ducks happy Mr Martinmere allows them to play cricket every Thursday afternoon. The number of ducks prepared to play changes frequently, depending on the weather, sore wings, and so on. Cilla has written a computer program to work out how many teams can be made up from the eligible ducks. Her variable declarations look like this:

```
final int TEAMSIZE = 11;
int numDucks = 36;
```

Cilla works out how many teams there can be like this:

```
int numTeams = numDucks / TEAMSIZE;
```

When she runs the program it tells her that there are three full teams available this week. She is surprised to find some ducks left team-less. What's going on?

Cilla has mixed up her data types, or at least she has not thought carefully enough about what the results of her division will be. Because all the variables in her assignment are integers, integer division has taken place and the remainder from the division has been overlooked. She needs an expression to work out how many spare players there are; this would look like this:

```
int spareDucks = numDucks % TEAMSIZE;
```

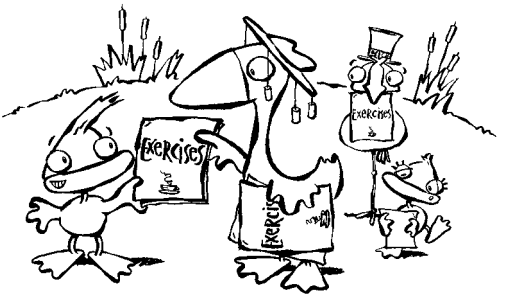
This uses the *modulus* operator, which, as we learned earlier, gives the remainder when the first operand is divided by the second. This would tell Cilla how many eligible ducks will be at a loose end on Thursday afternoon. This expression is equivalent to the rather more cumbersome:

```
int spareDucks = numDucks - (TEAMSIZE * numTeams);
```

Having said all this, it makes sense to move these calculations into separate methods (*getNumTeams* and *getRemainder*, say), so that if the calculations need to change drastically, they can be done once in the method body rather than (potentially) several times in the main method.

Here is the complete program:

```
/* CricketScheduler.java - Cilla's cricket scheduling software
   Author    : GPH
   Date      : 9th June 2003
   Tested on : Linux (Red Hat 8.0), JDK 1.4.1
*/

public class CricketScheduler
{
  public CricketScheduler () {}

  public int getNumTeams (int num, int size)
  {
    return num / size;
  }

  public int getRemainder (int num, int size)
  {
    return num % size;
  }

  public void printInfo (int num, int size)
  {
    System.out.println ("There are " + num + " ducks available.");
    System.out.println ("This means " + getNumTeams(num, size)
                        + " teams of " + size + ".");
    System.out.println ("This leaves " + getRemainder(num, size)
                        + " substitutes.");
  }

  public static void main (String[] args)
  {
    CricketScheduler cs = new CricketScheduler ();
    int numDucks = 36;
    final int TEAMSIZE = 11;

    cs.printInfo (numDucks, TEAMSIZE);
  }
}
```



8.1 Assuming that *number* is declared as an integer variable, what value is assigned to *number* by each of the following statements?

```
number = 10;
number = 10 + 6;
number = 10 + 6 * 4;
number = 10 + 6 * 4 / 2;
number = 10 + 6 * 4 / 2 - 2;
```

8.2 Assuming again that *number* is declared as an integer variable and that it has some value greater than 0 all of the following statements achieve the same thing. What?

```
number = number + number / number;
number += number / number;
number = number + 7 / 7;
```

8.3 Write two more statements that would achieve the same results as the statements in Exercise 8.2.

8.4 Identify and explain the problem with the following code fragment:

```
int coots;
char answer;

coots = 10;
answer = coots;
```

8.5 Identify and explain the error (or, indeed, the errors) in this fragment.

```
string name;
name = 'Buddy';
```

8.6 A program stores the name of the favourite chocolate bar of a group of ducks in a string variable. Which of the following is a good identifier for this variable?

thefavouritechocololatebaroftheducks
theDucksFavouriteBar
walrus
favBar
THEDUCKSTOPBAR

8.7 Assuming that *number* is declared as an integer variable, what effect do these statements have on the value of the variable?

```
number --;
// number = 10;
number *= 2;
```

8.8 Suggest the most likely data types to be used for a variable to store:

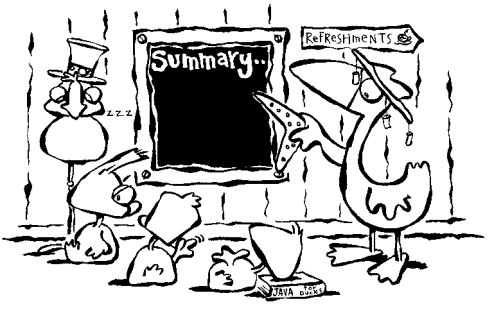
Your name
Your age
Your birthday
Your height
Your initials

8.9 Identify, explain, and correct the errors in the following Java program.

```
Public class Days
{
  Public static void main(Sting args{})
  {
    constant int daysInYear = 365;
    double numYears == 64;
    int totalDay = daysInYear*double;
    System.out{"There are " + total +
    " days in " daysInYear + " years.")
  }
}
```

8.10 Eggs are always sold packed in half dozens (packs of six). Write a Java program that declares a variable to store the number of eggs available for packing and then calculates the number of boxes that are needed. You can assume that each box must contain exactly six eggs. Your program should also calculate how many eggs are left over. What would you have to change in your program if new regulations meant that eggs had to be sold in boxes of ten?

8.11 All of the examples in this chapter were written with objects. They could have been written without and, we admit, sometimes the use of objects is a little cumbersome. Have another look, and ask an experienced Java programmer to show you what they would look like without objects.
There has been a lot to learn in this chapter so you might want to go back soon and have another read! The ideas described here are fundamental to everything else in this book so you should be sure that you're confident with them before reading on.



You should now know what a variable is and how one is declared. You should understand that a variable has a type and an identifier. You should know how to identify the appropriate types and choose good identifiers for the variables in your programs.

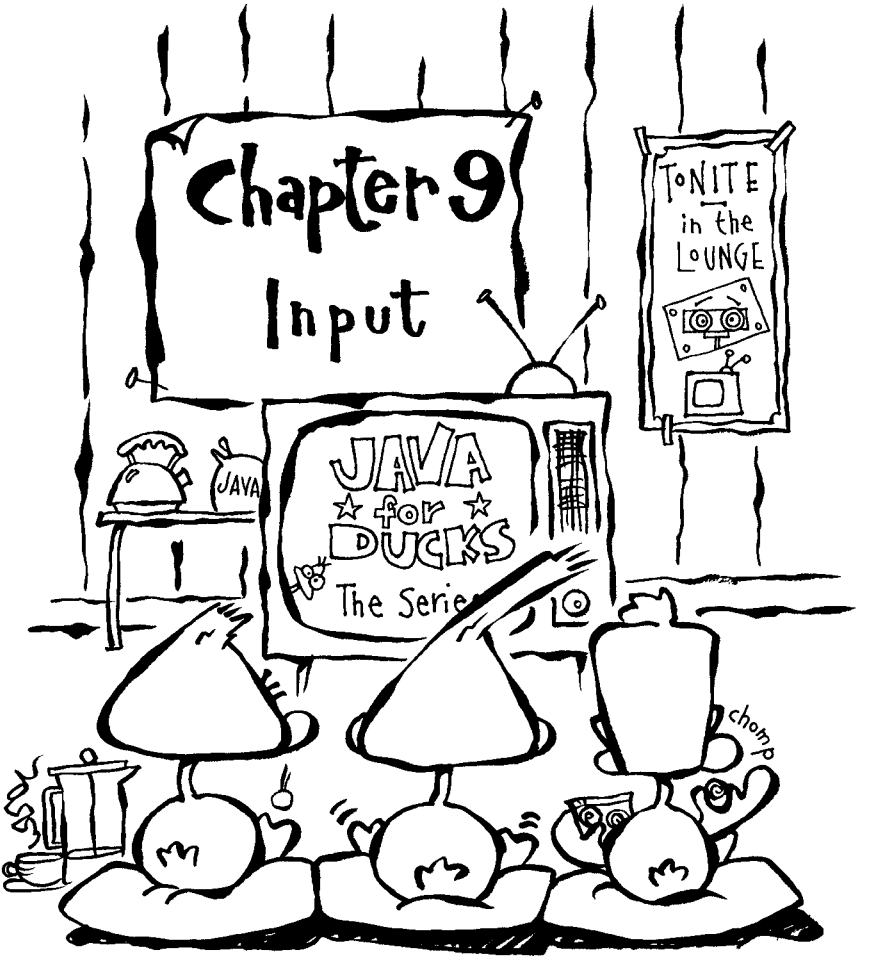
Constants are a special kind of variable that don't vary. You should be able to identify situations where it is sensible to use a constant in a program.

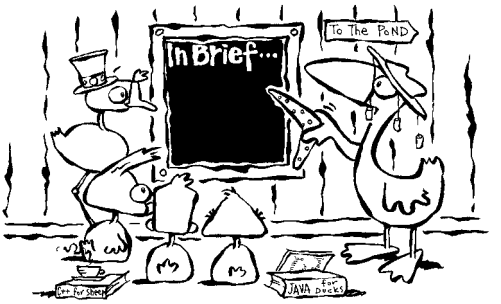
You should be able to write simple expressions using the basic arithmetic operators and you should understand the precedence of these operators.

You should also be able to produce simple output (probably on the screen) using `System.out.println`.

This chapter also included the traditional first program (displaying "hello, world") written in two rather different ways. One used objects, and one did not. Take a look at both programs and make sure you understand how they achieve what they do. They both in fact do exactly the same thing, so think about which is better. Think especially about which would be better if the program had to be extended.

The one thing that you can't do now is ask the users of your program to enter values for your program to process. Once we know how to do that we can start writing some really useful programs so let's move on to the next chapter!

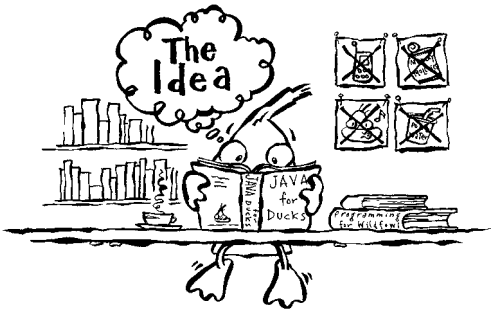

Chapter 9
Input
TONITE
in the
LOUNGE
JAVA
JAVA
for
DUCKS
The Serie
chomp



Up to now, we have looked at some simple programs that take values which have been set in the source code, manipulated these values and printed the results of these manipulations. You have to agree that while what we have learned so far is important, it is rather limited. In order to write any particularly useful programs, the programmer would need to know all the values the program needs in advance. This is hardly ever the case, and it is far better from the user's point of view if the program can receive values at the time it is run, rather than the time it is compiled.

Never fear! This chapter will teach you the basics of how the user can provide data to your programs to make them more flexible. This data can be provided in two ways – either passively, at the time the program is first executed (via what are known as *command line arguments* or *CLA*s for short) or interactively while the program is already running. You will learn some important differences between these two methods, and how to choose between them for a given task.

By the end of this chapter, you should be able to differentiate between the two sorts of input (command line argument and interactive), understand how both forms of input can be used in your programs, and determine which form to use for which purpose.



Virtually every program you will ever run from a command line, via a terminal on a Unix system, or via a DOS or Command Prompt on a Windows system, can be made to perform its task slightly differently depending on how it is invoked. Nowadays the command line is often hidden from us behind a "point-and-click" interface, so here's a reminder of what it's all about.

Users give instructions to computers as commands, words that have special meanings. These commands are interpreted by the operating system, which causes the computer to do something useful by executing a program. Usually,

the user will decide how to influence the program's execution by passing values to the program at the command line. Unsurprisingly, these values are known as *command line arguments* (CLAs) and as an example, let's look at the DOS `dir` command:

```
C:\zep>dir
   Volume in Drive C is DRIVE1
   Volume Serial Number is ABCD-9876

   Directory of C:\
12/06/2003      12:55    <DIR>          .
12/06/2003      12:55    <DIR>          ..
11/06/2003      10:43             6,150 wanton_song.txt
05/06/2003      22:42             5,341 white_summer.txt
21/05/2003      19:56            17,542 whole_lotta_love.txt
16/06/2003      09:21             9,974 wiawsnb.txt
```

The `dir` command is used to list files. When invoked with no arguments, the output is a list of filenames found in the current directory. It is obvious why this might be useful, but the `dir` command can be all the more powerful by making use of the command line arguments it understands. One is the `/O` argument (or *flag*) which tells the `dir` program to sort its output by some rule other than the default (alphabetical order). `/OD` is used to sort by file time, `/OE` sorts by extension, `/OS` by file size, `/ON` by name (which is the default). So this provides a list of files ordered by their date:

```
C:\zep> dir /OD
  Volume in Drive C is DRIVE1
  Volume Serial Number is ABCD-9876

  Directory of C:\
21/05/2003      19:56            17,542 whole_lotta_love.txt
05/06/2003      22:42             5,341 white_summer.txt
11/06/2003      10:43             6,150 wanton_song.txt
12/06/2003      12:55    <DIR>          .
12/06/2003      12:55    <DIR>          ..
16/06/2003      09:21             9,974 wiawsnb.txt
```

There are also flags available to list hidden files, list the contents of other directories, and other useful actions. From this simple example, it should be obvious why command line arguments are such useful tools to be able to incorporate into your programs. If you've used a Unix system you'll certainly have seen how widely used command line arguments are there.

Now, there are some programs which accept data interactively, after they have started execution. Often this is because the values the program receives cannot be known for some reason when the program is invoked. Since we looked at an example to demonstrate command line arguments, let's look at another for interactivity. This is the humble Unix `login` program.

```
tetley login:
```

Anyone familiar with the Unix text console will instantly recognise this prompt, as they have to enter their username and password when they see it in order to gain access to the system. The `login` program is automatically invoked after the system is rebooted or when another user has logged out. The system cannot possibly know which user will log in next, so there is no point in trying to pass the user's name as a CLA. Therefore it makes sense for the `login` program to receive the username when the user comes to log in.

```
tetley login: gph
Password: <password>
Last login : Wed May 28 20:04:53 on tty4

tetley%
```

Assuming the username and password are entered correctly, the user now has access to the system (hence the prompt is displayed). If, however, the credentials are incorrect in any way, the console would look something like this:

```
tetley login: gph
Password: <password>
Login incorrect

login:
```

So, it is easy to make an interactive program fault-tolerant without the need to terminate it at the first sign of a problem. In the above case, the program simply prompts for a username and password again after displaying an error message. If such a program were, for any reason, non-interactive, it would be much harder to incorporate the same level of fault-tolerance, and in fact it would probably be easier (though more untidy) to display an error message and terminate the program immediately.

There is no reason why a program cannot benefit from both command line arguments and user interaction. For example, there are text processors which take the name of the file to be processed as a command line argument, and if the text contains a mistake, the user is prompted to enter the correct text.

Using command line arguments in Java programs

As with many things in programming, command line arguments are easy to use, but rather tricky to use correctly. In the *dir* example above, each argument needs to be identified, its function invoked, and the result combined with the rest of the output to give the user what they expect. Java is no different in this respect; it is down to the programmer to perform checks on the arguments and have the program perform accordingly; there is no magic involved.

Believe it or not, you have been specifying command line arguments since the very first program you wrote. They are hidden away in the declaration of the *main* method:

```
public static void main (String args[])
```

You should be familiar with this line by now since every executable Java program contains this line, in exactly this form. Up to now, the `String args[]` part has been a mystery to you, but this actually specifies the list of command line arguments provided to the program at the command line. The square brackets tell the compiler to expect a list (known as an *array*, which we will look at in more depth in Chapter 17), and this list contains objects of type `String`.

A simple example would be helpful at this point:

```
/* CLAtest.java - a nonsense example demonstrating CLAs.
   Author    : GPH
   Date      : 28th May 2003
   Tested on: Linux (Red Hat 8.0), JDK 1.4.1
*/
```

```
public class CLAtest
{
  public void printTwoCLAs (String cla[])
  {
    System.out.println ("First argument is : " + cla[0]);
    System.out.println ("Second argument is : " + cla[1]);
  }

  public static void main (String args[])
  {
    CLAtest ct = new CLAtest ();
    ct.printTwoCLAs (args);
  }
}
```

Here we simply take the first two elements of the command line argument list and print them out with an appropriate message. For the purposes of this and subsequent chapters, the notation for obtaining the separate elements of a list is:

```
name[element-1]
```

So above, the list is called *cla*, and the first element is accessed by using the value 0 (1–1). Remember what we said in the first chapter about computers counting from 0? Java programs always count this sort of list from 0.

This program is extremely limited, and it is easy to see why. It is left entirely up to the user to ensure that the correct number of command line arguments is supplied. You can try experimenting with the program yourself to see what happens when they don't; any fewer than two and the interpreter complains, and the program stops running before it can do anything useful. Any more than two and the arguments from three upwards are discarded without warning.[1]

The list of command line arguments can have any name the programmer chooses. This program used *cla* as the name, but it is more usual to use *args* (short for arguments) which is what we will do for the rest of the book. You might also find C or C++ programmers who insist on using *argv* as the name for the list since this is what C and C++ use for the same thing. It's another style thing.

Let's try another, slightly more useful, example which uses exactly the same principles as the nonsense example above. In the last chapter we encountered Bruce's *Sign* class, which allowed Bruce to set a message for the sign to display. The version we have seen was extremely inflexible, in that the message has to be hard-wired into the source code before compilation. This is not ideal, especially if a novice user has to run such a program. So it makes sense to incorporate the command line argument input idea into this program.

The only change necessary is to read the message from the command line arguments list rather than a predetermined value. Assuming the message is passed as the first (and probably only) argument and that the list of arguments is given the customary name *args*, the relevant code would be:

```
brucesSign.setMessage (args[0]);
```

1 These arguments are still there, in the array, of course. It's just that we're not doing anything with them. Later on we'll see how to check the number of arguments supplied, and how to display error messages if this number is not as expected.

So, incorporating this single change into the previous version of Bruce's sign makes it much more flexible since we can now change the message without needing to edit the source code and recompile. The new version is:

```
/* CLASign.java

   Bruce's sign, which receives its message from the
   CLA list.

   Author   : GPH
   Date     : 4th July 2003
*/
public class CLASign
{
  // Single Attribute - What to Display
  private String message;

  // Methods
  public CLASign ()
  {
    message = "";
  }

  public void setMessage (String newMessage)
  {
    message = newMessage;
  }

  public void display ()
  {
    System.out.println (message);
  }

  public static void main (String[] args)
  {
    // Create an object for the sign
    CLASign brucesSign = new CLASign ();

    // Set the Message...
    brucesSign.setMessage (args[0]);

    //...and display it.
    brucesSign.display ();
  }
}
```

Interactive input

So far we have looked at a way of obtaining data from the user passively via command line arguments. While this certainly allows you to write more useful programs than you have been able in previous chapters, it is still fairly restrictive – the user must know the data values that need to be passed *before* running the program. A program would be more flexible if the user were given the option to pass data to the program while it executes – in other words, interactively. Thankfully, help is at hand, as Java has mechanisms available to allow such a thing.

First, a word of warning. The example programs we give here are not strictly speaking "correct" Java since the code required to accept interactive

user input is complicated, and far beyond your capabilities at the moment, so we have chosen to hide it in classes of our own for the time being.[2] So if you were to move to a different computer and try to run these programs, you may well find that they won't run. Before carrying on, make sure that you have the *Console* class correctly set up, as described in Chapter 3.1

The first example program in this section demonstrates how to make a running program read an integer value.

```
/* ReadDemo.java - Demonstrating a simple form of interactive
                   data entry.

   Author     : GPH
   Date       : 8th June 2003
   Tested on  : Linux (Red Hat 8.0), JDK 1.4.1
*/

import htpuj.*;

public class ReadTest
{
  public static void main (String[] args)
  {
     System.out.print ("Enter an int : ");
     System.out.println ("You entered \"" + Console.readInt ()
                         + "\"");
  }
}
```

It should be clear from this example that the *Console* class is the "black box" we have provided to allow you to read data interactively. This class is part of our very own package, called *htpuj*, which must be `import`ed at the start of any program using it. The *readInt* method of this class, as you would expect, attempts to read an integer value after the user has typed one (and throws a suitable error if the value it encounters is not of the correct format, that is, all digits). *readChar*, *readDouble*, and *readString* methods are also provided, and these work exactly as you would expect.

Continuing with the theme started earlier in the chapter, Bruce's sign can be extended so that the message can be provided interactively. Again, only minor changes are needed. First, we need to import extra functionality (the *Console* class) from an external source:

```
import htpuj.*;
```

Next, we need to add some sort of prompt for the user's benefit:

```
System.out.print ("Enter message: ");
```

Finally, we need to call the *setMessage* method, passing the message we receive from the user as the parameter:

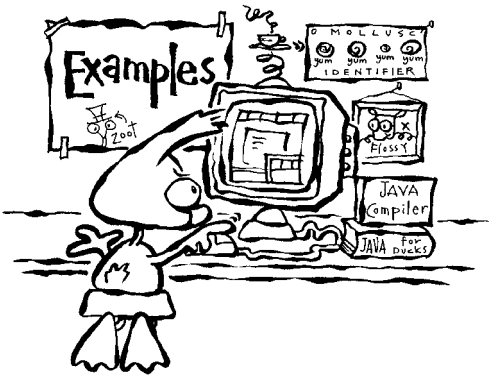
```
brucesSign.setMessage (Console.readString () );
```

2 Don't worry, we will reveal all before the book is out, when you've learnt enough Java to understand how our classes work! If you really want to know, it's all in the back of the book.

The final version is now a little longer:

```
/* InteractiveSign.java

   Bruce's sign, which receives its message from the
   user interactively.
   Author   : GPH
   Date     : 4th July 2003
*/

import htpuj.*;

public class InteractiveSign
{
  // Single Attribute - What to Display
  private String message;

  // Methods


  public InteractiveSign ()
  {
    message = "";
  }


  public void setMessage (String newMessage)
  {
    message = newMessage;
  }


  public void display ()
  {
    System.out.println (message);
  }


  public static void main (String[] args)
  {
    // Create an object for the sign
    InteractiveSign brucesSign = new InteractiveSign ();

    // Prompt the user
    System.out.print ("Enter message: ");

    // Receive and set the message...
    brucesSign.setMessage (Console.readString () );

    //...and display it.
    brucesSign.display ();
  }
}
```

That's about all there is to say about input at the moment. The examples are easily adapted to suit your needs, and we will go on to add more flexibility to your programs in later chapters, allowing you check the number of command line arguments, to prompt for input repeatedly, and to check the input for errors.



Example 1 – Buddy's times table assistant

Buddy has never really had a head for numbers, and has a hard time remembering his times tables. Unfortunately, the other ducks are far too busy waddling about the reserve, quacking and playing cricket to help him most of the time, so he decides to write a program to help him. What is the best way to perform this task?

As anyone over the age of seven knows, times tables involve only whole numbers, so this task can be undertaken using only `int` values. Since this program only needs to receive one value from the user, and the user should know this value when they run the program, it makes sense to pass the value as a command line argument. Notice the extra code to convert the String value to an integer.

The example program below would give the first six terms in a times table, it is hardly rocket science to extend it for more terms!

```
/* TimesTable1.java - Buddy's program to help with times tables

   Author     : GPH
   Date       : 9th June 2003
   Tested on  : Linux (Red Hat 8.0), JDK 1.4.1
*/

public class TimesTable1
{

  public void printLine (int x, int y)
  {
    System.out.println (x + " times " + y + " is " + (x*y) );
  }

  public static void main (String[] args)
  {
    TimesTable1 tt = new TimesTable1 ();
    int num = Integer.parseInt (args[0]);

    tt.printLine (1, num);
    tt.printLine (2, num);
    tt.printLine (3, num);
    tt.printLine (4, num);
    tt.printLine (5, num);
    tt.printLine (6, num);
  }
}
```

Example 2 – Cilla's cricket poser revisited

Cilla decides to change her cricket scheduling program from the last chapter, to incorporate user input. The program works as before, but this time the user provides the data interactively.

The user is asked to enter the number of ducks prepared to play cricket today, and the calculations are carried out in exactly the same way as earlier. The only changes needed therefore are:

```
import htpuj.*;
```

which tells the compiler and interpreter that some of the functions in this program are found in other packages. The number of ducks available to play is now found with a suitable prompt:

```
System.out.print ("How many ducks want to play?: ");
numDucks = Console.readInt();
```

A prompt is shown to the user, and the program takes the data the user inputs and stores it in the *numDucks* variable (which was used in the earlier example). The rest of the program is unchanged, apart from the class being renamed to distinguish it from the earlier example.

```
/* CricketScheduler2.java – Cilla's improved cricket
                            scheduling software
   Author     : GPH
   Date       : 9th June 2003
   Tested on  : Linux (Red Hat 8.0), JDK 1.4.1
*/
import htpuj.*;

public class CricketScheduler2
{
  public CricketScheduler2 () {}

  public int getNumTeams (int num, int size)
  {
    return num / size;
  }

  public int getRemainder (int num, int size)
  {
    return num % size;
  }

  public void printInfo (int num, int size)
  {
    System.out.println ("There are " + num + " ducks available.");
    System.out.println ("This means " + getNumTeams(num, size)
                        + " teams of " + size);
    System.out.println ("This leaves " + getRemainder(num, size)
                        + " substitutes.");
  }

  public static void main (String args[])
  {
    CricketScheduler2 c2 = new CricketScheduler2 ();
    int numDucks;
    final int TEAMSIZE = 11;
```

```
        System.out.print ("How many ducks wish to play?: ");
        numDucks = Console.readInt ();
        c2.printInfo (numDucks, TEAMSIZE);
    }
}
```

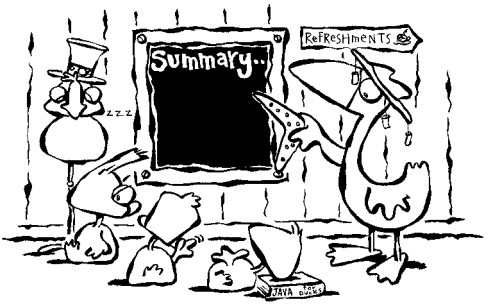
9.1 Identify and explain the error(s) in the following fragment of code:

```
final int numCoots = 11;
System.out.print ("Enter the number of coots : ");
Console.readInt (numCoots);
```

9.2 Modify Mr Martinmere's reserve poser (from the examples in the last chapter) so that it accepts the numbers of ducks and coots on the reserve as command line arguments rather than hard-coded values.

9.3 Change the program you wrote for Exercise 9.2 so that it prompts the user for the numbers and accepts the values interactively, rather than at runtime.

9.4 Modify Elvis's pocket money poser (also from the previous examples), so that the user is prompted for their name (to personalise the program somewhat) and the values used in the calculations.

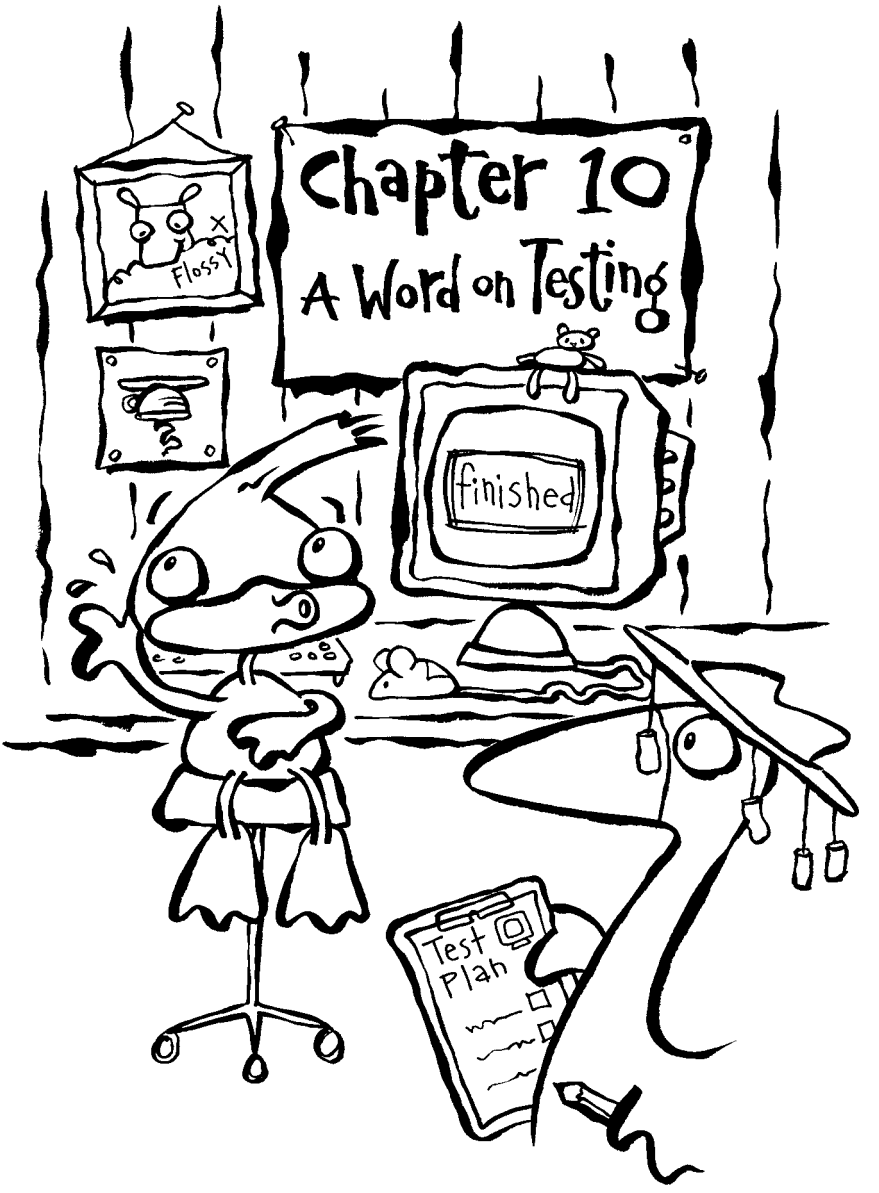


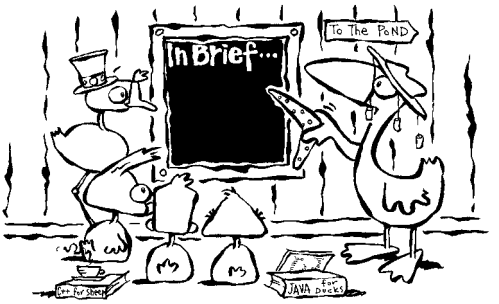
This chapter does not contain a huge amount in the way of new Java, but the concepts it introduces are fundamental to allowing any amount of user interaction with your programs. Hardly any real-world programs run without some sort of input, either from the command line at runtime, from the user during execution, or from some file or network connection. Therefore it is important to know at least a couple of ways of obtaining data from the user.

Command line arguments (CLAs) are used to pass values to a program as it is invoked, while interactive prompts are used to obtain values while a program is executing. Command line arguments are only useful if the user knows the values before the program is run; if this is not the case, interactive input can be used instead.

Processing command line arguments is far easier to code than processing interactive input, but the extra effort is worth it in programs where, for example, values need to be input repeatedly, where there is no set order for the values, or where there is a choice of values to input (say, to change a duck's name, weight or value).

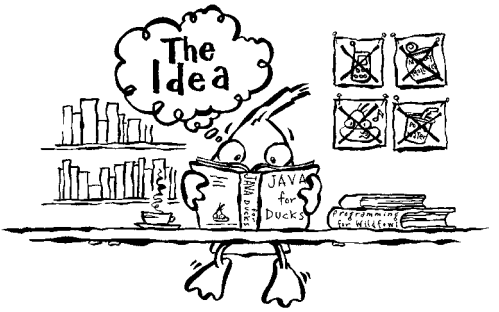
Some of the programs in this chapter are getting a little complicated. It is important that we are sure that the results they produce are correct, so the time has come to take a quick break from Java to look briefly at how programs can and should be tested.


Chapter 10
A Word on Testing
Flossy
finished
Test Plan



You have now written your first complete Java program. Most of the rest of this book will be about Java, but in many ways writing a program is only half of the story. If a program is going to be truly useful the programmer and, more importantly, the program's intended final users must have confidence in the results that it produces. They must be sure that the results they see from the program are correct. Business users are potentially going to base costly business decisions on the results produced by the programs they use, pilots are going to trust navigational systems in their aircraft, and so on. This brings us to a brief pause from looking at Java. This chapter looks briefly at how and why computer programs are tested.

After reading this chapter you should understand something about the ways in which all your programs should be tested. You should be able to write a *test plan* for a program and you should also be able to carry out the plan. Using the ideas from this chapter you should be able to demonstrate that all the Java programs you write in the future are "correct" and you should understand why it is never possible to be completely sure that a program is completely correct!



Writing a computer program is a complicated business but it is far from the only activity involved in producing reliable software for reliable computer systems. We have already seen the importance of thorough analysis and design before a program is written. The process of producing reliable computer systems is usually called *Software Engineering*. The term software engineering includes a wide range of activities – designing software, testing software, writing documentation, and more – of which programming is only quite a small part. Some programmers call themselves *Software Engineers* to reflect the range of tasks involved; in many organisations a programmer would be expected to carry out a range of such tasks. Programming and software engineering are not the same thing.

Computer programs are everywhere in the modern world. Modern cars have systems that rely on extremely complex programs to control everything from the car's heating system to its brakes. Airliners have computerised "fly-by-wire" systems that almost allow them to navigate and fly themselves. Anyone who drives a car or flies on an airliner is placing a lot of trust in these computer programs and by implication in the skills of the programmers who wrote them; if the programs failed by producing the wrong results the consequences could be catastrophic. Remember that all these programs were at some point written by a human programmer and that humans often make mistakes. That is why all these programs were also very thoroughly tested before anyone was allowed to use them in any sort of potentially costly or dangerous situation.

Testing programs is another important part of software engineering. For a computer system to be truly useful its users must be confident that the system functions correctly and that the results it produces are correct and reliable. The process of testing a computer program to ensure its accuracy and reliability is in many ways much more important than the process of writing the program itself.

Testing a simple program

Cilla's most impressive program for working out how many cricket teams the ducks can form will serve as an example of how a simple program can be tested. This program takes the number of ducks who want to play cricket on an afternoon and displays how many teams there should be and how many ducks are left over[1] and will have to be substitutes. The process of testing this program thoroughly is quite complex, even though the program itself is very small and straightforward.

Here is the program again:

```
/* CricketScheduler.java - Cilla's cricket scheduling software

   Author     : GPH
   Date       : 8th June 2003
   Tested on  : Linux (Red Hat 8.0), JDK 1.4.1
*/


public class CricketScheduler
{
  public int getNumTeams(int num, int size)
  {
    return num / size;
  }

  public int getRemainder (int num, int size)
  {
    return num % size;
  }
```

1 Admittedly the results of an error in this program are unlikely to be as catastrophic as, say, an error in a fly-by-wire system but an enraged waterfowl that has lost out when there's a chance to play cricket is not a pleasant thing.

```
    public void printInfo ()
    {
      System.out.println ("There are " + numDucks +
                          " ducks available.");
      System.out.println ("This means " +
                          getNumTeams(numDucks, TEAMSIZE) +
                          " teams of " + TEAMSIZE + ", and " +
                          getRemainder(numDucks, TEAMSIZE) +
                          " substitutes.");
    }

    public static void main (String args[])
    {
      CricketScheduler cs = new CricketScheduler ();
      int numDucks        = 36;
      final int TEAMSIZE  = 11;

      cs.printInfo();
    }

}
```

This is the version of the program that has the numbers hard-coded; we are just using this version to keep the code short here. Obviously in "real life" Cilla would use a version where she was able to enter the number when the program was executed.

The first step in testing this program is to develop a plan; not surprisingly this is called a *test plan*. This plan will simply list some input values (called *test data*) that will be provided to the program together with the results that would be expected if the program were working as expected. These expected results are calculated, in advance, by hand. They are obviously checked very thoroughly as mistakes can be disastrous.

Each set of input values and expected results forms one *test case* for the program. If the program produces the correct results for each test case (and assuming that there are sufficient test cases to cover all possibilities) then it will be safe to assume that the program is correct and so works as intended.

Some sample test data for Cilla's program is:

Input	**Expected Outputs**	
ducks	teams	ducks left over
10	0	10
65	5	10
100	9	1

So, for example, the third test case here states that if there were a hundred ducks the correct results from the program would be nine teams with one duck left over as to act as a substitute. The other two test cases simply test other possible input values.

Testing the program is then just a case of running it for each set of data and recording the actual results. These results are compared to the expected results for the same test case and if they all match the program has passed all the tests and is assumed to work.

The problem is that this doesn't mean that the program actually works! There is only sufficient information here for a programmer to assume that it does. In the example above there are only three sets of test data; if the program passes all three test cases all that is actually known is that it works correctly with these three pairs of input values. It is certainly not possible to be sure that it works with every possible set of input data; with most types of input data[2] there could always be some extra test case that has not been tried for which the program produces an incorrect result.

This is the problem with testing any program. A program will be expected to work with very many input values; a program very similar to the program above could potentially process an infinite number of possible values. For this reason it is rarely possible to test any program with every possible set of test data. Instead, representative values are used to build up test cases in a test plan; if the program passes all the test cases it is reasonable to assume that it works correctly in all other possible cases. This is the case with all programs; a program in an airliner cannot be tested on every route the airliner might fly and in all weather conditions. For any program a subset of the possible input values must be used in the test plan. Picking those values requires thought; this process cannot be allowed to be random.

Programs should also be able to cope with input values that are not as expected; a user may enter a negative number when a positive number is expected,[3] for example. This means that programs should also be tested with unexpected values. In this example an input that indicated that there was a negative number of ducks is obviously nonsense, but it is important that the program deals with such inputs sensibly. There are also some specific cases to look out for. There is a potential error in this program if a negative were entered for the number of ducks. This would give the situation were negative ducks were assigned to teams! Another common error occurs when a value is being used as a divisor in a calculation; such a value cannot be allowed to be 0, since division by 0 is an error. The program must always cope with errors of this kind and must behave sensibly.

Taking this potential error case into account a more complete test plan can be drawn up. In this case 0 is allowed to the number of ducks (there would be no teams, and no ducks left over, presumably), but it is the lowest possible value that is allowed. A more complete test plan for the example would look something like:

Input	**Expected Outputs**		**Pass?**
ducks	teams	ducks left over	
10	0	10	
65	5	10	
100	9	1	
0	0	0	
−1	Error		

Once again there are assumptions in this plan. A new one is that if the program correctly handles one input of a negative number for the input value

2 In this small example it might be possible to test all the possible values, assuming that there was a small number of ducks. But, in more sophisticated programs, there is generally no sensible way to record and test all the possible input values.

3 Users are like that.

it will also correctly handle all others. This assumptions, and there are more even in this simple case, are necessary since it is rarely feasible to test a program with all its possible input values or combinations of input values. A final change in this test plan is the addition of the final column; this is just there to record the result of the tests.

Boundary and typical values

A systematic plan is needed for choosing the values to be used in test cases. One possibility would be to just choose random values[4] but this could lead to carrying out too many or too few tests and could mean that the testing takes far too long or overlooks some important test cases. Something much more systematic is required. A system is needed that will allow testers to choose a small enough number of test cases so that they can actually carry out the test in a reasonable amount of time but which still gives them enough test cases to allow them to be confident that a program is correct.

There are basically two sorts of expected values that can be input to a program. *Typical values* are simply examples of those that the program is expected to process when it is running normally; testing with these is just a case of choosing a collection of typical values and assuming that if the program processes these all correctly it will also correctly process all other similar values.

Boundary values are more complicated. These are those that occur around a limit on the acceptable values. In the example above there is a lower boundary on the possible value for the number of ducks – it cannot be less than 0. The program should be tested with all boundary values and also the two values immediately surrounding each boundary. In the example there is an assumption that there is no upper boundary on the number of ducks, although there might well be in practice.

To help the tester, a test plan should also indicate the purpose of a particular test case – whether the value is typical or boundary. A full test plan for Cilla's Cricket Scheduling program might be:

		Input	Expected Outputs		Pass?
		ducks	teams	ducks left over	
1	Typical	10	0	10	
2	Typical	65	5	10	
3	Typical	100	9	1	
4	Boundary	0	0	0	
5	Boundary	−1	Error	Error	
6	Boundary	1	0	1	

The other addition this time is numbers to index each test case. This is simply for ease of reference so that, for example, someone testing the program could

4 Programs do exist that generate random numbers that can be used as test data to be used as input to other programs. They are useful for carrying out large numbers of tests automatically.

easily and unambiguously give the programmers a list of the tests that the program fails.

In this plan, tests one to three are randomly chosen typical values while tests four to six test the lower boundary on the number of ducks (we have assumed that there might reasonably be no ducks at all and that there is no upper limit). Notice how a lot of the testing is concentrated around the boundary values. In practice we would tend to expect three tests at each boundary value, so this is obviously where most testing will be concentrated. This is expected since it seems reasonable to assume that this is precisely where errors may have been made.

Writing and using the test plan

The test plan can be written directly from the specification of the program and so can and should be written before the program itself is written. It is normally easy to identify boundary and typical values from the specification and the expected outputs should be calculated by hand and carefully verified. The plan should be allowed to be dynamic to some extent; it should be changed if the program's specification is changed. The plan should certainly be complete before starting to write the program. While some testing will take place during the program's development as the programmer checks the code the final test should always be carried out according to the plan. The final test of a program should never be carried out randomly and the final test plan should never be written after the program is complete.

In many commercial software development businesses there will be people whose specific job it is to test programs. These will rarely be the same people who write the programs. This is a good thing; it is sometimes very difficult to be objective when you test your own programs. After all, you wrote it and you know how good it is. You know it works so what is the point in thoroughly testing it?

It's likely, though, that while you learn to program you will be testing your own programs. When you do this try to be as thorough and as ruthless as possible. It's worth remembering that if you don't find an error someone else will. Make sure that your test plan is complete and that it covers all the possible cases. It's also a good idea to ask a friend to test your program while you return the favour and test theirs. You can enjoy yourself finding tiny errors in your friend's program while they derive similar satisfaction from testing yours.

The final version of a program should pass all the tests in the plan. Sometimes, of course, the plan will show up errors that must be corrected and the program will have to be changed as a result. Obviously the program must then be tested again. In this case it is vital that the whole test plan is run again after any change; a change to correct one error may well have caused an error somewhere else in the program and another test case may now produce incorrect results. The final version of a program should always pass every single test case in the plan.

Testing and debugging

As you write your programs you will be running them from time to time to check that they compile and work as expected. More often that not you will look at the outputs, spot mistakes, and then make changes to the program. You will find bugs and you will fix them. This is *debugging*, not testing.

Do not confuse these two activities. Debugging is an important part of the programming process; a programmer finds errors in the program and fixes them. But this happens before testing. Testing is a structured activity used to verify that a program is correct. It takes place only when the program is believed to be correct and when all the known bugs have been found and corrected. Of course, testing may well uncover more bugs but it still remains a separate activity.

User testing

The first true test of any computer program comes not as it is being written but when it is used "in anger" by one of the intended users. This kind of testing takes place in a very different environment to that in which the testing strategy described here is used. Users are very unpredictable things[5] indeed!

Most commercial software will be thoroughly tested by the software company that wrote it. But before it is made widely available the company will always send it to a few selected users for *beta testing*. This testing simulates the environment in which the programs will eventually be used and can often show up errors that could not be found with any test plan.

You can adopt this idea, again by cooperating with a friend. A fine way to test one of your programs is to ask a friend[6] to use it for a while. Don't give them the test plan and don't ask them to test it systematically but just ask them to play around with your program and see what they can find. The idea is to see how your program works when a real user uses it. Challenge your friend to find a mistake in your program; you might even offer them a prize if they find one! You will be amazed at what they will do with your program as they attempt to break it.

Testing is important

Testing a program thoroughly can be a tedious and repetitive job but it is also a vitally important one. A program that produces incorrect results is a very dangerous thing; many people have great faith in the figures generated by a computer and errors in these figures can be costly and dangerous. Whenever we get in a car or fly in an airliner we are placing a great deal of faith in the programmers who wrote the programs and perhaps even more faith in the testers who tested the programs.

Whole books have been written about testing and this chapter has been only a very quick taster of the whole process. You should now be able to write a test plan for each program you write and you should be able to carry it out. Above all you should understand why thorough testing is so important!

You should also realise that it is never possible to be completely certain that any program works as expected in all cases. There will always be some set of input values, some test case, or some other set of circumstances that might cause the program to fail. What is important is that the testing is as thorough as possible so that the users can have as much confidence as possible in the

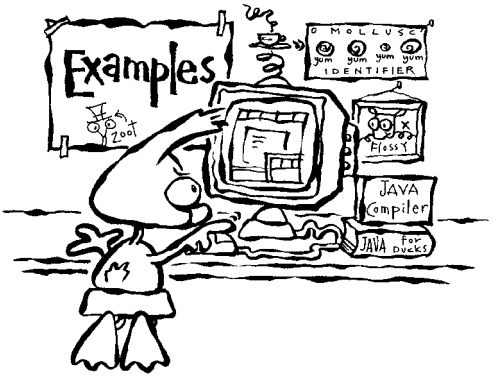
---

5 It has been said that the most used expletive in the software development industry is "Users!". It is probably true.

6 A good friend. Or at least a friend you trust. Friendships can be broken over errors in programs!

program. Obviously the program's application will also determine how thorough the testing must be; a program to sort ducks into cricket teams probably requires less thorough testing than a program to fly an airliner.

In fact testing a program thoroughly is just as important as writing it in the first place. In many cases it is much more important; an untested program is useless and sometimes even a dangerous thing.



Example 1 – Ice skating ducks

Every year the pond freezes and Elvis and his friends seize the opportunity to hold an annual Ice Skating competition. Each competitor receives a mark out of six from each judge. There are five judges. The final mark for the competitor is calculated by discarding the highest and lowest mark received and then taking the average of the three marks that remain.

Write out a test plan for the program that calculates the final marks.

The highest mark from any judge is six and the lowest is 0, so these are the boundary values. The plan will also have to verify that the correct average is calculated and that the correct scores are being disregarded; this can be done with a range of typical values.

The plan would be:

	Purpose	Judge					Mark	Pass
		1	2	3	4	5		
1	Typical	6	6	6	6	6	6.0	
2	Typical	1	6	3	3	3	3.0	
3	Typical	6	6	2	2	4	4.0	
4	Typical	1	6	2	4	4	3.33	
5	Boundary	−1	6	6	6	6	Error	
6	Boundary	0	6	6	6	6	6.0	
7	Boundary	7	6	6	6	6	Error	

The typical values in tests one and two are chosen randomly; in fact they're chosen to make calculating the expected result easier! Test three includes two equal highest and lowest scores to check that only one of each is disregarded and test four has a result that is not an integer. The final three tests check the

boundary conditions; this is only tested with judge 1 so there is an assumption that the other judge's marks are processed in the same way.

Example 2 – Elvis's calculator

Elvis has a simple program that allows him to do some basic mathematics. It allows him to add, subtract, multiply, and divide. He has tested this program thoroughly and believes that it works. To help him with some more complicated mathematics he adds a square root function for integer values to his program. What should he add to his test plan?

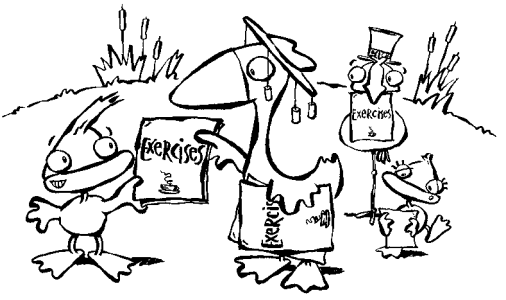
A range of typical values is easy to find; pretty much any positive values will do. Since there is no square root of a negative number[7] all negative values are error cases, so Elvis will have to be sure that his program works correctly with those, presumably by generating some sort of error message. The program should correctly find the square root of 0 (which is 0), so this is a boundary.

The extra test cases in his plan would be:

	Purpose	Input	Output	Pass
1	Typical	4	2.0	
2	Typical	15	3.87	
3	Boundary	0	0	
4	Boundary	−1	Error	
5	Boundary	1	1	

This time there are test cases on both sides of the boundary (tests three to five). There are also assumptions that if the program correctly deals with −1 it will do the same with −2 and that an accuracy of two decimal places is sufficient. This accuracy is determined by the application; if the program were to be used in a more complex application such as airliner navigation it would be necessary to use and check many more decimal places.

Elvis would also have to run all the other test cases for his program before being confident that his new program worked. It is entirely possible that adding this new function to his calculator has introduced errors elsewhere in the program.



10.1 Type Cilla's cricket scheduling program into your computer (or get it from the web site), compile it, and execute it. Use the test plan to see if it is

7 Only an imaginary one, of course.

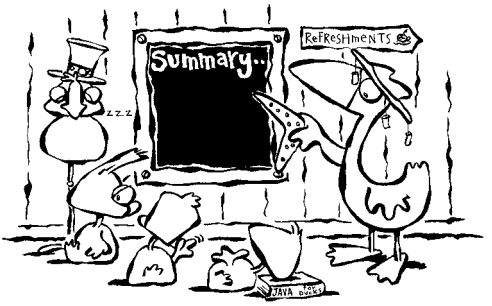
correct in all cases. Identify any cases where the program does not work correctly. You'll find it easier to use the version that takes input from the user.

10.2 The case of dividing a number by 0 has been identified as a potential error in a program. Write a small program to find out what your Java system does when faced with a division by 0. At what stage does your Java system spot this error? You don't know how to stop this happening yet (that's in Chapter 14) but at least you'll recognise it when it happens!

10.3 In the first example in this chapter the assumption was made that if the first judge's marks were handled correctly all the others were also handled correctly. Is this a sensible assumption to have made?

10.4 A program takes the marks achieved by a student on three tests and displays the average. Each test is marked on a scale of 0 to one hundred inclusive. Write a test plan for this program.

10.5 The program from Exercise 10.4 is extended to display the class of the student rather than the average. Three classes are defined; A (for average marks 70 to 100), B (average 40 to 69.9), and C (39.9 and less). Write a test plan for this version of the program.

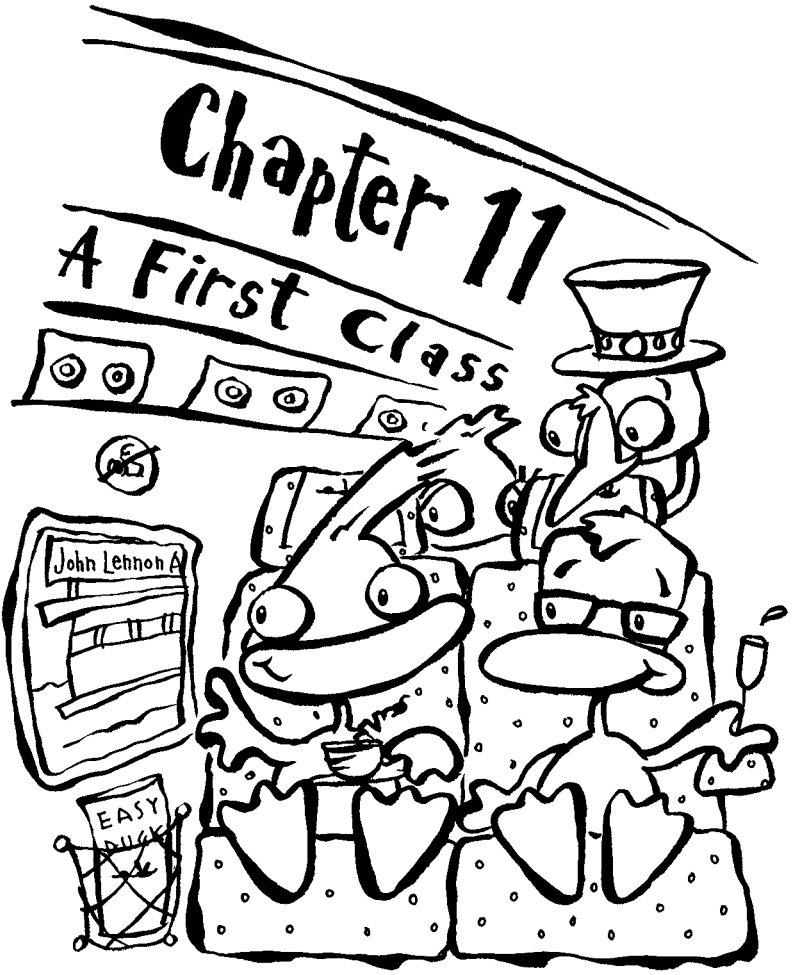


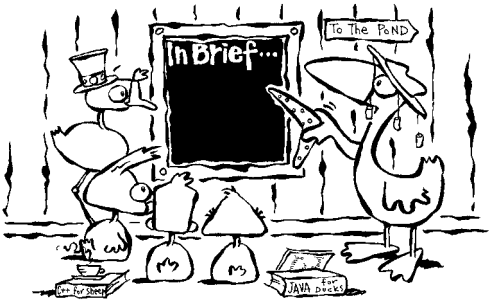
Testing is important. As well as having to learn to program you are now going to have to learn to test your programs. It will be best if you get into the habit of developing and using thorough test plans. Never be tempted to treat testing as a random activity. It can be tedious, but it is very important.

A test plan provides a list of test cases that can be applied to a program. Each test case consists of a set of input values and the corresponding set of output values. A program has passed the test if and only if it passes all the test cases. Only then should a programmer have any confidence in the program. Only then should a programmer be prepared to pass the program on to its intended users.

The test plan should be written before the program. Ideally it should not be written by the programmer or by anyone who has any knowledge of the details of how the program works. The test plan should certainly not be written as an afterthought when the program is believed to work!

Writing a program is only part of the activity of programming. Testing the program and correcting the errors revealed is another, equally important, part.

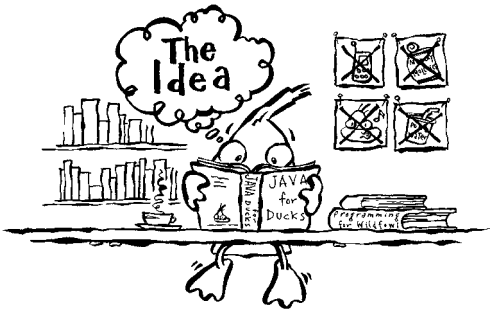

Chapter 11
A First Class
John Lennon A
EASY



Up to now, we have looked at some basic objects, given them a moderate amount of functionality, and learned how to use these objects in simple programs. This book was never intended to be *that* basic, so in the next two chapters we shall revisit much of the earlier material, going into greater depth about classes, objects, methods, and attributes.

There is little new in the way of Java in these chapters; we are concentrating more on writing classes that could conceivably be reused by other programmers in the future, and the methodology of writing such programs, rather than any bells and whistles that would overcomplicate matters at this stage.

In this chapter, you will be building a class yourself. By the end of it, you should understand how to create a `.java` file to define your class, and you should understand a little more about methods and attributes and how to incorporate some into your class.



A `.java` file defines a class. A `.java` file can, of course, also contain a program. As we have seen, a Java program is essentially a Java class with a special method called `main`. So in effect, any Java class can be made into a program, for good or bad, by defining a `main` method within that class.

Before we look at a definition of a class, let's quickly revise some definitions.

- An *object* is something of interest in a problem area. It is something that is going to be used in or somehow affected by a program. An example might be a duck.
- An *object type* is a specification of the characteristics shared by a collection of objects. An example might be "ducks". In Java (and in most other object-oriented languages) an object type is called a *class*.
- An *attribute* is a characteristic of an object type that is of interest. This is a single value of a particular type. The ducks will have names (strings), ages (integers), values (money), and more.

- A *method* describes a behaviour of an object that is of interest. A duck might move to a particular point, might increase its value or might change its name. Programs interact with objects through methods.

So the definition of a class is going to need to define the object type that the class models. This involves specifying the attributes (and their types) and the methods (and how these operate). We'll start with attributes, but first the syntax of the file.

A class definition

The definition of a class is obviously fundamental to the development of any program that uses the class. It is vital that the definition is laid out neatly so that a programmer using the class can see immediately how the class works and is in no doubt about the details. The chapter on analysis showed how it was possible to read a program specification, and deduce from it the object types (or classes) that such a program would use. Analysing each class in turn, it is then possible to settle on a list of attributes and methods that each object should define.

The first thing to do before trying to use a class is obviously to read through the definition. The file should start with a suitable header block of comments that will give you the general idea of what the class is for (and will provide you with the name of the original author in case you have any complaints!). Then, assuming that it follows the usual format, you should find definitions of the attributes and methods.

You should take note of the exact name of the class, the types of any attributes, and all the details of the methods. Later on you'll be able to read the statements inside the methods to see what they do, but you shouldn't have to do this as everything should be clear from comments. If anything isn't totally clear it's probably a good idea to seek suitable clarification before attempting to use the class.

The basic format of a well-written definition looks like this:

```
// Header Comments
public class <class name>
{
  // Attributes defined
  // Methods defined
}
```

From the top, the first important element is a *header block* of comments that provides someone reading the file with all the information they need about the class. This information should include at least the name of the class, some description of its purpose, the name of the author, the date the file was written, and perhaps details of the particular Java platform that was used.

We have already seen this in all the complete Java definitions and programs so far. An example might be:

```
/* Duck.java - A simple Duck class.

   Author    : AMJ
   Date      : 31st December 2002
   Tested on: Red Hat 7.3, JDK 1.4.0
*/
```

Here we have the name of the class and a brief description. The rest of the information identifies the author (in case someone finds a mistake and wants to complain!), the date the class was written, and finally the platform that was used (Red Hat is a version of the Linux operating system).

The line following the comments provides the name of the class that is defined. The convention used here for naming classes is that the identifier is singular rather than plural and has an initial capital letter. The name of the class is also the name of the file containing the definition, of course. So in this example the class is defined in the file `Duck.java` and the first line of the definition must be:

```
public class Duck {
```

The `.java` file can also import other libraries at this point if they are needed – these are in fact other class definition files – an idea you have met when including the library for reading input.

It is also often useful to include clear comments to indicate the meaning of each part of the file. Admittedly, it is not strictly necessary in such a simple example as the one in this chapter, but it is a good habit to get into.

The remainder of the definition, after these initial comments and the rest of the preamble, provides the code that actually implements the class. In this file it is customary to list the attributes first, followed by the methods. It is important to remember that a Java class is split into two parts; rather like people classes have a public and a private side. We have seen this split before, but now is the time to understand exactly what's going on.

Attributes

Once again, this is something that we looked at quickly way back in Chapter 6. This section just summarises that slightly more formally.

An attribute is some value of interest that needs to be stored about an object. This will usually be some value or feature that is relevant to some problem that is being solved. The attributes of a class are simply listed in the first part of the definition; this involves specifying the identifier of the attribute and its type. The syntax is as expected:

<visibility> <type> <identifier>;

Here, *<visibility>* denotes which components of a program can access the attribute; there are three levels of visibility (or, more formally, *scope*), and we will examine two of them (`public` and `private`) in more detail a little later. More often than not most attributes have private scope, and that is the convention that we will use.

The same guidelines apply to the choice of identifiers for attributes as applied to the choice for variables. The identifier should be short and concise but should still convey its meaning. The type of the attribute can be any of the types that are available for variables. In more complicated programs the type can also be another class to allow for the cases that we have seen where an object type has an attribute that is itself another object type.

With the attributes defined, it's on with the methods.

Methods

When analysing a problem, three pieces of information were determined about each method required for an object type. Not surprisingly, the same three pieces

of information are required to specify a method in a *.java* file. These are the name of the method, the type of the value that it determines (if any), and the types of any values that it processes (the parameters).

The format for declaring a method is not unlike the declaration of an attribute:

<visibility> <type determined> <name of method> (<types of values processed>)

Again, *<visibility>* corresponds to the method's *scope* (`public` or `private`), and it is usually (but not always) the case that a method has public scope.

This definition itself provides a complete definition of the method. For example we will soon meet a program that tracks ducks on the pond. It stores their position and often calculates how far they have strayed from a particular point. A method to determine a duck's distance from a point might be declared:

```
public double distanceFrom (int dx, int dy)
```

This should, of course, be extended with a suitable comment if something is not absolutely clear.

The types and identifiers of values processed by the method are specified in a comma-separated list; if there are no values this list is just left blank (but the brackets remain). The values inside the brackets are called the *parameters* of the method, or sometimes the *arguments*. A method may also automatically make use of any of the attributes in the private part of the class (and also any of the other methods); the parameters are used to provide additional values. The type of value that the method determines is called the *return type* of the method.

All this means that the method definition can be dissected. It tells us that the method determines a floating-point value, is called *distanceFrom* and takes two integer values as parameters. Even though there is no comment, with some knowledge of the likely application we can deduce that this method calculates the distance that a duck is from a particular point on the pond, with the distance being determined as a floating-point value.

Some methods do not return a value. This explains the need for a data type, `void`, that indicates this. A method that does not return a value is defined as if it was returning a `void` value. So:

```
public void changeName (String newName)
```

defines a method that changes a duck's name, a process that does not involve returning an interesting or useful value. This is a *void method*.

A definition of a method contains a lot of information. This information is vital for the programmer who will be writing programs that will make use of the class and its methods. A programmer must be able to read a class definition file and understand precisely what it conveys about the interface of the class.

Public and private data

As we have said, a *.java* file defining a class is, by convention at least, split into two parts. These two parts specify the attributes and methods that a class possesses. We have also mentioned a special property known as scope. We shall now discuss *public* and *private* scope (there is a third, `protected`, which we will look at briefly in the very last chapter).

The distinction between these two is straightforward. Anything defined to be public is available to any program that might make use of the class, while anything defined as private is available only to the class itself.

This is a very powerful mechanism. It means that, as long as the public part of the class (the *interface*) remains constant, the private part can be changed if needed. This means that a programmer using the class needs only know about the class's public face; the private face remains hidden and can be changed. At the same time, the programmer working on the class can change the details of the private part, perhaps to implement something more efficiently, without risking any unpleasant effects on programs that use the class.

An analogy would be the controls of a car. The normal controls – the steering wheel, pedals, handbrake, and so on – remain constant and form the public interface to the car's engine. Any changes that, say, a mechanic might make to the engine itself are totally independent of this interface; anything that changes in the engine is private and is something that the driver doesn't need to know anything about. The public interface of the car always works in the same way, and this should not be altered by any changes made to the engine.

The convention used in this chapter (and throughout this book) will be that attributes are always defined with private scope, and methods usually with public scope. This is not required, but it is easily the most common approach. It also allows for a powerful and secure mechanism called *data hiding*, as we will see later in Chapter 13.

Finally, let's define some attributes and methods!

Attribute and method definitions

We have taken a theoretical look at attributes and methods, and discussed why they are usually split between having private and public scope respectively. Now it would make sense to define some attributes and methods and see them in practice.

We shall return to an earlier example, which we first saw in Chapter 6 – a class to represent a duck. Recall that, for Mr Martinmere's purposes, a duck has a name, age, and value. These would be stored as attributes – name as a `String`, age as an `int`, and value as a `double`. Since all the attributes we expect to encounter in this book are private, their declarations would be:

```
private String name;
private int age;
private double value;
```

Let us extend the duck class somewhat; since each duck is permitted to waddle freely about the reserve, we can provide attributes to store the duck's location, should Mr Martinmere ever need to track them. Assuming all the ducks are on the ground,[1] we can achieve this with two attributes, storing the duck's *x* and *y* position as `int` values (`double` would also be acceptable, but it's unlikely we'll even need to get down to that degree of accuracy):

```
private int x, y;
```

1 If we allowed the ducks to go upwards we would soon be embroiled in the three-dimensional version of Pythagoras' theorem. That alone is reason enough to simplify the problem.

Since attributes tend to be found at the top of a source file, so far our class will look something like this:

```
// History block

public class Duck
{
   private String name;
   private int age, x, y;
   private double value;

   // Methods
}
```

Next we move onto the methods. It is usually a good idea to have methods which allow access to at least some of the private attributes (known as *accessor* or *selector* methods). We will look at this in more detail in Chapter 13, but for the moment we'll just add a few likely methods.

The methods that a class would have depends, of course, on the needs of the programs that would use the class. In the current example, the *Duck* class will be extended with a few possibilities:

- a method to do nothing more than return the duck's age (an accessor method);
- a method to display the duck's name and position neatly;
- a method to determine how far away the duck is from a particular position (this position would be provided as two parameters, one for the x coordinate and one for the y coordinate);

Each method can be defined from its description, provided we know the type of value it will return, and the types of any parameters it takes. For example, we know the duck's age is stored as an `int` value, and such a method will take no parameters, so the declaration of a method to access this attribute will be just:

```
public int getAge ()
```

The other methods would be declared as follows:

```
public void printNeatly ()
```

This method, called *printNeatly*, does not require any parameters and does not return a value.[2]

```
public double distanceFrom (int some X, int some Y)
```

This method requires x and y coordinates as parameters, and returns a double value.[3]

---

2 There is a common convention among Java programmers that each class should have a method called *toString ()* that provides much the same information as the *printNeatly ()* method here. Defining such a method allows us to pass an instance of the class to the `System.out.print ()` and `println ()` methods. In fact, a default version of the *toString ()* method is defined for all classes. Why not write a simple program to see what this default version does?

3 There would be a loss of precision if the calculation returned an `int` value – how far is (1,1) from (0,0), for example?

So far we have the definitions of the methods; the implementations themselves will need to be added later. For the moment the class looks like this:

```
public class Duck
{
 // Attributes

 private String name;
 private int age, x, y;
 private double value;

 // Methods

 public int getAge ()
 {
 // implementation here
 }

 public void printNeatly ()
 {
 // implementation here
 }

 public double distanceFrom (int someX, someY)
 {
 // implementation here
 }
}
```

Remember that while the Java is correct in this example this is not a complete definition of the class. The implementations of the methods are still missing but, before adding these, there is one more method to include.

The constructor

There is one last method to add. We have seen that to declare an object the programmer specifies the type of the object to be declared, provides an identifier, and calls the constructor:

<class name> <identifier> = new <constructor>();

The constructor is the special method that takes care of creating an instance of the class, an object. The name of the class and the constructor are, obviously, the same. If the constructor has been defined to take parameters these are also provided in the brackets. The name of the constructor should be written in exactly the same way as the name of the class given at the top of the file; this includes the case of the letters. Any mistake will mean that the method is not recognised as the constructor.

Happily most of what the constructor needs to do is taken care of automatically. All the allocation of memory happens behind the scenes and the programmer does not need to worry about it. The constructor may also assign default values for some of the attributes; comments should be added to the header file to document this (and explain the values that are used) if this is the case. It is very poor if the implementation of the constructor (or any other method for that matter) has to be examined to determine exactly what it does.

In this case the constructor does not take any parameters.[4] The declaration is just:

```
public Duck () // Constructor
```

The addition of a constructor gives an almost complete definition file for the *Duck* class, but now some implementations are needed.

The implementations

The final part of defining a class is to add Java statements that implement the methods. Any Java statements can be used, even ones that we haven't met yet! Let's take each of the example methods in turn.

```
public int getAge ()
```

This method does nothing more complicated that finding the value of an attribute and passing it back (or "returning") it to whatever called the method. The statement to do this in Java is simply `return` so:

```
public int getAge ()
{
   return age;
}
```

The next method:

```
public void printNeatly ()
```

needs to do some output. The exact format of this output would probably be determined by the application, but some sequence of *System.out.print* lines will be needed, such as:

```
public void printNeatly ()
{
   System.out.println ("Name:      " + name);
   System.out.println ("Position:  " + x + ",  " + y);
}
```

This method does not return a value. The statements are simply executed in the order shown.

The next method requires a calculation:

```
public double distanceFrom (int someX, int someY)
```

A small amount of geometry is needed, in fact Pythagoras' theorem. Without boring you with the mathematical details, the required calculation is:

```
public double distanceFrom (int someX, someY)
{
 return Math.sqrt ((double) (((someX - x) * (someX - x))
                           + ((someY - y)  * (someY - y))));
}
```

4 Actually, a constructor can take parameters. An obvious possibility is some initial values for the attributes of the class. If this is required the prototype has the types of the parameters specified in the brackets as you might expect.

This method also makes use of the built-in Java function for finding a square root. There are lots of other useful maths functions to be found in the same place.

The final method that needs to be implemented is the constructor. Most of what the constructor does is done behind the scenes automatically and in many cases like this there is no need for anything other than an empty implementation. Here the constructor is nothing more than that:

```
public Duck ()
{
}
```

It is worth pointing out that Java will provide a constructor just like this one if no constructor is specified in the class. Defining your own constructor is a good habit to get into, but it's not always disaster if you leave it out. If the attributes of the class are all of primitive types (integers, characters, Booleans, and so on), and you are happy with the compiler default values, then not including a constructor (or else including an empty constructor) will not cause any major problems. However, if you need to initialise an object, or a primitive variable to a non-default value, it is important that you include code to achieve this in the constructor.

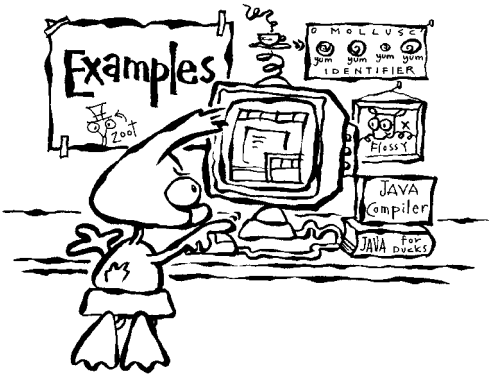
This gives the final definition of a simple *Duck* class that might be used in some sort of duck-tracking application. A small final change is that the integers representing the position have been separated out; it is usually a good plan to group together connected attributes in this way. The class definition is now:

```
/* Duck.java - A simple Duck class.

   Author    : AMJ
   Date      : 31st December 2002
   Tested on: Red Hat 7.3, JDK 1.4.0
*/

public class Duck
{
  // Attributes

  private String name;
  private int age;
  private double value;
  private int x, y;

  // Constructor

  public Duck ()
  {
  }

  // Methods

  public int getAge ()
  {
    return age;
  }
```

```
    public void printNeatly ()
    {
       System.out.println ("Name:      " + name);
       System.out.println ("Position:  " + x + ", " + y);
       return;
    }

    public double distanceFrom (int someX, int someY)
    {
       return Math.sqrt ((double)  (((someX - x) * (someX - x))
                                  +((someY - y) * (someY - y))));
    }
 }
```

Defining a class

That's all there is to it. Once the class has been defined we can start working on programs that use the class, which is what we'll come to in the next chapter. But first, some examples!



Example 1 – Bruce's library

Bruce has set up a small library in his hut. He naturally wants to keep records of the ducks who borrow his valuable books, and he has decided to develop a small Java program to help in this task. As a first stage he has decided to create a simple class to store details of the ducks who borrow. The first attempt will be limited to a single simple method to display the details of one duck borrower.

What does the class implementation file look like?

The borrowers from Bruce's library are all ducks. We have already seen a class for storing the details of ducks in the chapter, so it might seem that this class would form a good starting point. But this is probably not so. The class in this chapter was developed for a particular application; the main aim was to track the ducks as they moved around on the pond. The task of keeping details of their book-borrowing activities is a very different thing indeed.

It is possible to generalise Bruce's requirement, and to develop a class for things that borrow. This is a reasonable step because, as we have seen before, there are many applications where things are borrowed by other things and applications where things are borrowed by ducks are, it must be admitted, rare.

Bruce is obviously interested in the name of his borrowers, so we might start the definition of the attributes with:

```
public class Borrower
{
   private String name;
```

This is fine, but it would not be possible to identify an individual borrower if two had the same name. This is a common problem, and it is quite common to work around it by introducing some other value that is guaranteed to be unique. An obvious addition here is some sort of borrower number:

```
public class Borrower
{
   public String name;
   public int number;
```

That should be enough for the attributes. Two methods are needed in this first version; one simply displays the details of a borrower and the other is, of course, the constructor. In this example the constructor need do nothing very much:

```
public Borrower ()
{
}
```

While the other method is little more that a pair of statements to print the required values:

```
public void print ()
{
   System.out.println ("Borrower: " + name);
   System.out.println ("Number: " + number);
}
```

This gives the final version of Bruce's first attempt at a class:

```
public class Borrower {
  private String name;    // Name of borrower
  private int number;     // Unique borrower number

  public Borrower ()
  {
  }

  public void print ()
  {
    System.out.println ("Borrower: " + name);
    System.out.println ("Number: " + number);
  }
}
```

Example 2 – Cricketing ducks

The ducks continue to enjoy cricket. They naturally want to keep a range of statistics about each game, so that they can spend the long winter months arguing over who is the best duck cricketer.

Buddy has started to develop a program that will process the details of each duck's performance. He plans to store the number of innings each duck has had, the number of

runs that have been scored, the number of times the duck has been "not out", the number of wickets taken and the number of runs conceded.

The final program will obviously be quite complex, so Buddy has decided to start with a much simpler class. This class will simply store each of the required values, and will provide a method to display them and another method to prompt a user to enter them. Implement Buddy's class.

The class will obviously need to store the name of the duck which is a string; the numeric attributes of this class are all integers, so this section of the class is straightforward:

```
private String name;

private int innings;
private int notOuts;
private int runsScored;

private int wickets;
private int runsConceded;
```

Entering the values is also not too complicated. We will assume for the moment that there is no requirement to validate the values that are entered, so:

```
public void enterStats ()
{
   System.out.print ("Enter name : ");
   name = Console.readString ();

   System.out.print ("Enter number of innings: ");
   innings = Console.readInt ();
   System.out.print ("Enter runs scored: ");
   runsScored = Console.readInt ();
   System.out.print ("Enter not-outs: ");
   notOuts = Console.readInt ();
   System.out.print ("Enter wickets taken: ");
   wickets = Console.readInt ();
   System.out.print("Enter runs conceded: ");
   runsConceded = Console.readInt ();
}
```

Obviously we have to `import` the package containing the `Console` class before this program will compile!

A method to display the values would also be straightforward; it need be nothing more than a sequence of suitable print statements. But here is a more "Java-ish" way of achieving this.

A common approach is to include a method called *`toString`* in a class. This is used to return a single string that represents an individual object. This string can be useful in all sorts of ways to programs making use of the class, or can simply be displayed. The requirement here is only the latter, but this approach would be common among experienced Java programmers.

The definition of the method looks not unlike the definition of a method to print the values:

```
public String toString()
{
   String stats = "*** Stats for " + name + " ***";
   stats += "\nInnings       : " + innings;
   stats += "\nRuns scored   : " + runsScored;
```

```
    stats += "\nNot-outs       : " + notOuts + "\n";
    stats += "\nWickets taken : " + wickets;
    stats += "\nRuns conceded : " + runsConceded;
    return stats;
  }
```

This gradually builds up the string *stats* to form a string[5] that contains a summary of all the information. This method can then be used in another that will generate this string and display it:

```
public void printNeatly ()
{
   System.out.println (toString () );
}
```

With the class written, the question arises of how Buddy can test his methods. The solution is simple; he can add a `main` method to the class and run it as a program; this version simply reads in the details for one object (representing one cricketing duck) and then displays them.

```
public static void main (String args[])
{
   CricketingDucks cd = new CricketingDucks ();
   cd.enterStats ();
   cd.printNeatly ();
}
```

The class can now be run as a program:

```
tetley% javac CricketingDucks.java
tetley% java CricketingDucks
Enter name: Buddy
Enter number of innings: 10
Enter runs scored: 200
Enter not-outs: 1
Enter wickets taken: 10
Enter runs conceded: 300

*** Stats for Buddy ***

Innings        : 10
Runs scored    : 200
Not-outs       : 1

Wickets taken  : 10
Runs scored    : 300

**********************
```

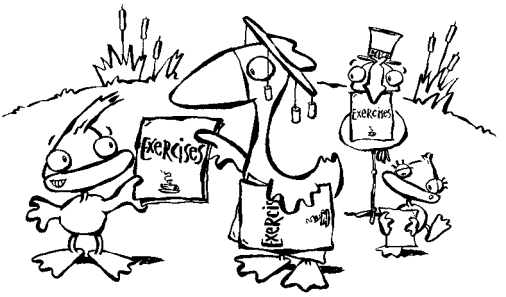
The complete definition of the class is worth looking through in some detail to make sure that you understand what's going on. It illustrates most of the basic Java concepts that we have covered so far, so take some time to go through it.

```
/* CricketingDucks.java - Program to store cricket statistics.

   Author    : AMJ
   Date      : 17th April 2003
   Tested on : Linux (Red Hat 7.3), JDK 1.4.0
*/
```

5 The `\n` inserts a newline into the string.

```
import htpuj.*;

public class CricketingDucks
{
  private String name;

  private int innings;
  private int notOuts;
  private int runsScored;

  private int wickets;
  private int runsConceded;

    public CricketingDucks ()
    {
      innings = 0;
      notOuts = 0;
      runsScored = 0;
      wickets = 0;
      runsConceded = 0;
    }

    public void enterStats ()
    {
      System.out.print ("Enter name: ");
      name = Console.readString ();

      System.out.print ("Enter number of innings: ");
      innings = Console.readInt ();
      System.out.print ("Enter runs scored: ");
      runsScored = Console.readInt ();
      System.out.print ("Enter not-outs: ");
      notOuts = Console.readInt ();
      System.out.print ("Enter wickets taken: ");
      wickets = Console.readInt ();
      System.out.print("Enter runs conceded: ");
      runsConceded = Console.readInt ();
    }

    public String toString ()
      {
      String stats = "*** Stats for " + name + " ***";
      stats += "\nInnings       : " + innings;
      stats += "\nRuns scored   : " + runsScored;
      stats += "\nNot-outs      : " + notOuts + "\n";
      stats += "\nWickets taken : " + wickets;
      stats += "\nRuns conceded : " + runsConceded;
      stats += "\n******************";

      return stats;
    }

    public void printNeatly ()
    {
      System.out.println (toString () );
     }
```

```
    public static void main(String args[])
    {
      CricketingDucks cd = new CricketingDucks ();
      cd.enterStats ();
      cd.printNeatly ();
    }
}
```



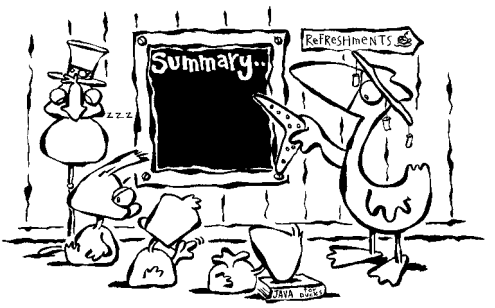
11.1 Bruce's *Borrower* class would be much more usable if it were possible to assign values to the attributes in the first place! Write some methods to do this – you might even consider using the constructor for part of this exercise.

11.2 Now extend the class again so that the user can enter values at the command line. Do this with interactive input, which we saw in Chapter 9.

11.3 Buddy's aim is to create a program that will work out the duck's batting and bowling averages. Extend the class with a method that calculates the bowling average (the number of runs conceded divided by the number of wickets). Remember to extend the *toString* method to contain this new value, and use the `main` method to test your new method.

11.4 Following the same procedure, add a method to calculate the batting average. This is the number of runs divided by the number of completed innings (the number of innings less the number of not out innings).

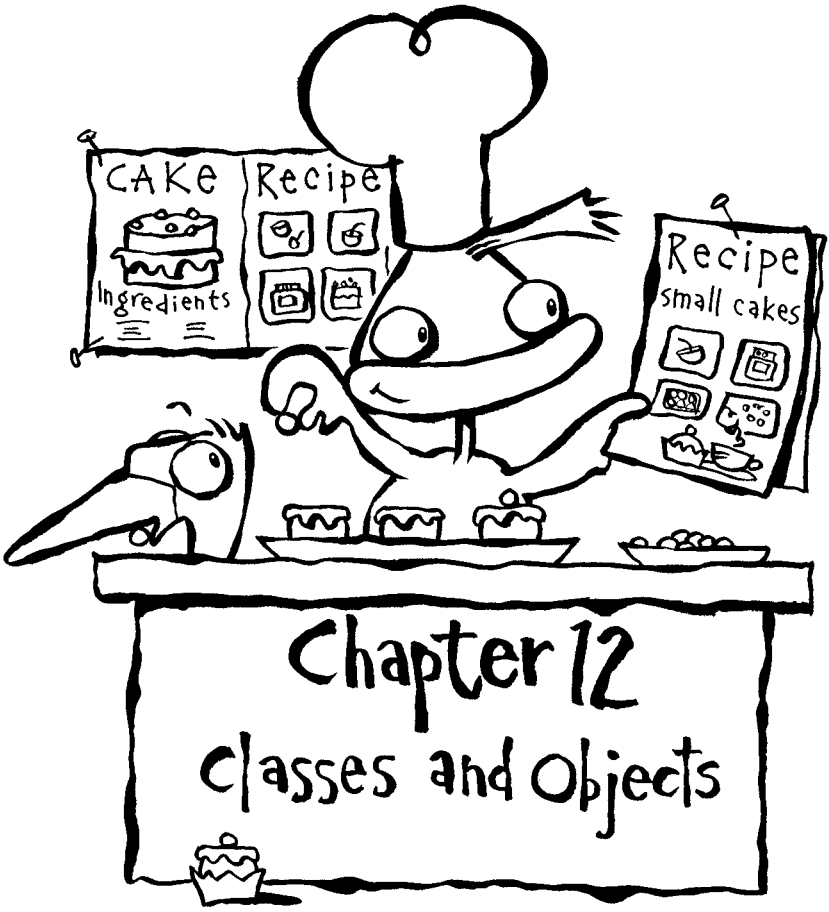
11.5 Now write a program that makes use of the class. The program should read in values for each of the attributes and should display the bowling and batting averages.



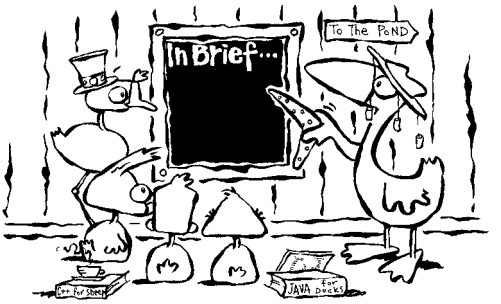
A Java class is defined in a *.java* file. This file contains the descriptions of all the attributes of the class and the descriptions and implementations of all the methods. It is customary to define the attributes as private members of the class and methods as public; we will see why this is important in the next chapter. It also customary to group the private and public parts of the class together; this serves to make the class definition easier to understand.

You should now be able to create and use simple Java classes. You should also understand how to provide a class with a `main` method, a method that allows a programmer to carry out some basic testing.

The idea of *data hiding* has been mentioned in this chapter, and will be examined again in Chapter 13. Before then, however, we will look at reusing objects that we have defined in programs, or even other classes!



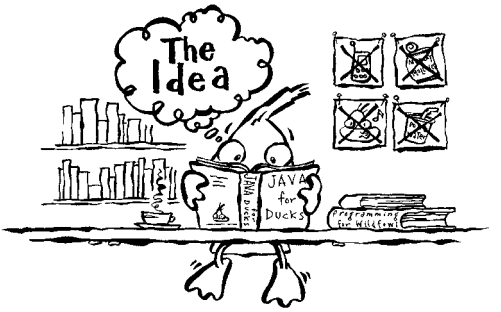
Chapter 12
Classes and Objects



You've now seen quite a lot of Java. You've seen how to create a class and how to create instances of the class (objects), and how to use these objects in very simple programs. This has all been a bit informal, though, so the time has come to have a proper look and to get some practice in writing your own programs. This chapter and the last one start you on the way to doing that.

You might want to go and take a quick look again at Chapter 6 when we had a first look at how to declare and use a class in Java, and at the chapters that have described how to do input and output. We're now going to build on those ideas to create complete object-oriented programs.

This chapter takes the basic Java statements that you have learned up to this point and shows you how the programming ideas that you have used and practised so far can be extended to produce Java programs that make use of classes and objects. After reading this chapter you will understand how to create and use objects in a Java program. You should be able to read a Java class's implementation file, understand it, and make use of the class that the file contains in your own programs.



There are two stages to developing an object-oriented program. The first is to define the class and the second is to develop the program that uses it. This chapter is interested in the second of these stages; we'll concentrate on reading the definition of a class (found in the `.java` file) and writing a program that uses it. We looked at the other stage in the previous chapter, of course.

The ability to split programming into these to fairly distinct pages is, of course, an advantage of object-oriented programming; a programmer can make use of class that some other programmer has written, and these two programmers can be working at the same time. Obviously the programmer using the class must first understand exactly what the class does and how it works.

A Java `.java` file describes amongst other things the *class interface*. This aspect of the file is the important one in this chapter; you will need to be able to examine the `.java` file defining a class and so work out how to make use of the class in your programs. You will normally be able to see all the details of the implementation of the class too, but this is not important if all you need to do is use the class.

It is also possible to use a class if only the `.class` file of the class is available. This can happen when for some reason the programmer who has written the class does not want a programmer using the class to have access to the details of the class's implementation. In this case, of course, the class interface will have to be specified in some accompanying documentation. We will assume here that the `.java` file is available, so we start by seeing how to understand its contents and so extract the class interface.

We saw a simple class defined in Chapter 6, and we defined a slightly more complicated class in the last chapter. Both of these examples defined a simple class to store details of ducks, and included suitable attributes and methods. In general, the basic format of a `.java` file defining a class is:

```
public class <classname> // the name of the class
{
   // definition of attributes
   // definition of methods
}
```

We saw in the last chapter that it is customary, but not compulsory, to define all the attributes of the class before defining the methods that make use of them. Both can be defined to be either *public* or *private*. The distinction is that anything defined as public is available to any program that makes use of the class, while anything defined as private is available only within the class itself. It follows that for the purposes of writing a program that uses a class we will be most interested in the things that are defined as public.

A common practice is to define all attributes as private and all methods as public. This allows the programmer to control access to the values of the attributes and so allows for a powerful mechanism called *data hiding*. We will keep to this convention in all the programs in this chapter, and indeed in all the programs in this book, and you are recommended to do the same. The advantages of data hiding are explored in more detail in Chapter 13.

So, the public part of the class normally specifies the methods that can be used on instances of the class (objects). It may also specify attributes but this is unlikely and is usually bad programming practice and so best avoided. The specification of a method tells a programmer using the class the name of the method, the type of value it returns (if any), the types of any values that it requires as parameters (if any), and finally provides one or more Java statements that carry out the method's task.

The file itself should be sufficient to explain completely and unambiguously what can be done with the class. Anything that is not totally clear in the `.java` file should always be explained by suitable comments.

Preparing to use a class

A program that uses a class is a said to *call* the methods of the class. It might be called the *calling program* of the methods. The program must be in the same

directory as the files defining the class, and will require access to a suitably compiled version of the class – the `.class` file.

This means that the first stage in preparing to use a class is to obtain the `.class` file defining it. If this has not been provided, the `.java` file must obviously be compiled to produce it. This is done in the usual way, for example:

```
tetley% javac someClass.java
```

You have in fact been doing this already if you've been making use of our `Console` class. You had to take the `.java` file of this class and produce the corresponding `.class` file. What we are going to do now is very much the same thing.

It is, of course, possible that there is some error in the definition of the class that will prevent it from compiling. If this is the case then these errors must obviously be traced and corrected before any program using the class can be written. There is clearly no point in spending effort on writing a program that makes use of a class that doesn't work properly!

This means that you now need to be able to compile programs that are stored in more than one file; as a minimum you will have one file containing the definition of the class and one containing a program. While you don't need to know what's in this class definition file you do need a copy of it, probably in the same directory as your program. Everything should be straightforward, assuming that the `.class` file is stored in the same directory as your program but, as you've probably come to expect by now, the way in which you compile programs that use a class can be slightly different on different Java systems. As always I'll have to refer you to your *Local Guide* to confirm all the details and to provide any extra information you need.

Reading the definition

The first thing to do before trying to use a class is obviously to read through the definition. The file should start with a suitable header block of comments that will give you the general idea of what the class is for and so on. Then, assuming that it follows the usual format, you should find definitions of the attributes and methods.

You should take note of the exact name of the class, the types of any attributes, and all the details of the methods. Later on you'll be able to read the statements inside the methods to see what they do, but everything should be clear from comments. If anything isn't totally clear we know by now that it's probably a good idea to seek suitable clarification before attempting to use the class.

Once you understand what the class is and what it can do the first step in writing a program is obviously to declare an object.

Declaring an object

The first stage in using an object (an instance of the object type defined in your class) is obviously to declare one. The format of the declaration is something we've met a few times before; we simply call the class's constructor.

An object declared in this way is effectively the same thing as the variables you have used before. The effect of defining a new class has been to create a new type of variable. The only difference really is in the format of the

declaration; with an object it is necessary to explicitly call the constructor. So we might declare a *Duck* object:

```
Duck elvis = new Duck ();
```

As with any variable more than one class instance can be declared in a single declaration:

```
Duck elvis = new Duck(), buddy = new Duck();
```

It makes sense to check the comments to see whether or not the constructor assigns initial values to each attribute of the object, and it might make sense to add a comment to the declaration to indicate what these are.

Now, with an object declared it is time to start working with the values. The attributes of the class will be private and so programs cannot access them directly. The way to access these values is through methods, and hopefully a collection of methods will have been provided for this purpose.

Calling a method

We've used methods before, for example, when we needed to set the string that was displayed on Bruce's sign, but here is a quick recap of the syntax for calling them. The *.java* file will provide a list of all the methods available, and will include all the details that we will need.

A method is called on an object in a statement of the form:

<object name>.<method name> (<parameters>);

This is not as complicated as it looks! The identifier of the object has the name of the method appended to it with a full stop between. The parameters are then specified in brackets after the method name. If there are several parameters they are separated by commas; this time the brackets contain the identifiers of the parameters and never their types. If there are no parameters the brackets must still be present but there is nothing between them.

An example should make this clear. Yet another simple *Duck* class designed for some duck tracking application similar to the one defined in the last chapter might include this set of methods:

```
public int getAge (); // return the duck's age
public void printNeatly (); // print all details of a duck
public double distanceFrom (int, int); // return distance from
                                       // a point
public boolean atPosition (int, int);  // true if duck is at
                                       // position provided
```

Assuming that the program had declared:

```
Duck elvis = new Duck ();
```

the statement to call the *printNeatly* method to display Elvis's details would be:

```
elvis.printNeatly ();
```

This method has the return type `void` indicating that it does not return a value (it's a void method). There are no parameters, and so the brackets are empty.

The other three methods, on the other hand, do return values, and the final two do have parameters.

The return values are presumably of interest to the programmer using the class and so they should be stored in a suitably declared variable of the appropriate type. This gives a slightly longer form of a call to a method:

<variable> = <object name>.<method name> (<parameters>);

Another example – to use the *getAge* method to retrieve Elvis's age and store it in a suitable program variable the statement is:

```
int age;
age = elvis.getAge ();
```

This method has no parameters, and so the brackets are present but, as usual with methods like this, they contain nothing. This assignment could also have been combined with the declaration of the variable to provide an initial value for that variable:

```
int age = elvis.getAge ();
```

It's worth pausing to mention that calling a method like this that returns a value with nowhere to store the value:

```
elvis.getAge ();
```

is very probably an error. It is almost always a very bad idea to ignore a value that a method is returning.

If a method requires parameters the values are included in the brackets. If there is more than one value all the values must be provided in the correct order (the types of the parameters must match the types specified in the method definition). The *distanceFrom* method returns the distance that a duck is from a particular point. To find out how far Elvis is from the point (5, 3) and store that distance in a program variable called *away* the statement would be:

```
double away = elvis.distanceFrom (5, 3);
```

This is obviously different to:

```
double away = elvis.distanceFrom (3, 5);
```

which refers to the point (3, 5). The correct number of parameters must be provided in the order specified in the method definition.

This call to the method would result in a compilation error:

```
double away = elvis.distanceFrom (4); // Error!
```

The error here is simply that the number of parameters does not match with the number specified in the method definition. Obviously the compiler cannot spot if the values have been provided in the wrong order; all it can check is the number of values provided and their types.

So with the ability to call all the methods of the class it should be possible to start on a program. But first an important warning.

A warning about assignment

Classes are obviously very similar to the standard data types that you have used in your programs so far. They are used in much the same way and much of the syntax used with them is the same as is used with, say, integers. The important difference to remember is that the *.java* file defines absolutely

everything that your programs can do with an instance of the class. The general rule is that if it's not defined in the *.java* file, and specifically in the public part of the *.java* file, there's no way to do it in a program.

There is one case where this is particularly important. It is often very tempting to want to assign the values of all the attributes of one instance of a class to another. This does not work as might be expected! For example, this code is totally legal, but may well not behave as the programmer intended.[1]

```
Duck elvis = new Duck();
Duck buddy = new Duck();
elvis.moveTo(10,10); // change position
buddy = elvis;
```

This would indeed change the second duck's position to (10, 10) but something more subtle has actually happened. The two objects are now effectively the same; by this, we mean that the names *elvis* and *buddy* now both refer to the same location in the computer's memory. This really is rather subtle, and is something that you don't really need to worry about for now. The problem now is that a later statement such as:

```
elvis.moveTo(10,11);
```

would affect both *buddy* and *elvis,* since the attributes of both are stored in the same place. It is important to remember that attributes must be copied using only the methods defined in the class.

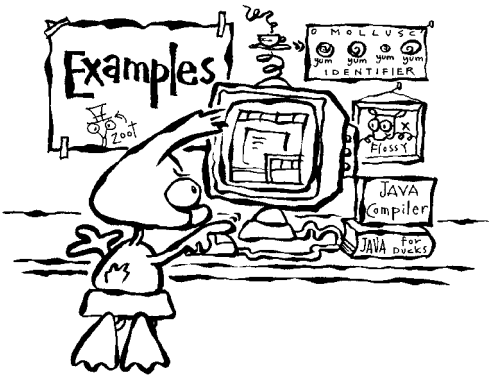
Using a class

A class effectively extends the range of data types that you have at your disposal as you write your programs. An examination of the *.java* file gives you all the information you need in order to use the class. After compiling the class, objects are declared in a similar way to variables.

Methods are called on an object by joining the name of the method and the identifier of the instance with a full stop. If a method requires parameters, the correct number must be provided in the expected order and must match the expected types.

This really is not as complicated as it might seem! Now it's time for some examples of how this all works.

1 A particular irritation is that in C++ using an assignment like this has a rather different effect, copying the values of the attributes.



Example 1 – A simple duck class

Mr Martinmere likes to keep track of the locations of all his valuable ducks. He gets very concerned if they stray too far away from the edge of the pond; if they do he has to think of some way to fetch them back. He has purchased a prototype tracking device, which he has fitted to Elvis for testing purposes.

Mr Martinmere wants a small Java program that interfaces with the tracking device. It would be incredibly useful if the program was easy to extend in future. Bruce has developed a class to store details of the ducks. The `.java` *file is*:

```
/* Duck.java - A simple Duck Class.

   Author     : Bruce, after AMJ
   Date       : 31st December 2002
   Tested on  : Red Hat 7.3, JDK 1.4.0
*/

public class Duck
{

  // Attributes

  private String name;
  private int x, y;

  // Constructor

  public Duck ()
  {
  }

  // Methods

  public void moveTo (int newX, int newY)
  {
    x = newX;
    y = newY;
  }

  public void setName (String newName)
  {
    name = newName;
  }

  public void printLocation ()
  {
    System.out.println ("(" + x + ", " + y + ")");
  }
```

```
  public void printNameAndLocation ()
  {
    System.out.println (name + " is at " + "(" + x + ", " + y + ")");
  }

  public String getStatus ()
  {
    if (distanceFrom(0,0) > 10.0) {
      return name + " is too far away!";
    }
    else return name + " is in range.";
  }

  public double distanceFrom (int atX, int atY)
  {
    return Math.sqrt ( (double) ( (atX - x) * (atX - x) )
           + ( (atY - y) * (atY - y) ));
  }
}
```

Write a program that declares an instance of the class and then prompts the user to enter its position and name. Use the `printNameAndLocation` *method to print the position neatly and the* `getStatus` *method to display a warning if the duck has strayed an alarming distance away.*

You will obviously need the *`.java`* file for the class in order to complete the exercise. It's available on the web site to save you typing it in. Getting hold of a copy in this way is much the best approach since you avoid all sorts of possible problems that might be caused by typing errors.

The first step in developing this program is to compile the class to produce the *`.class`* file. The precise steps required will naturally depend on your particular Java system, but something like:

```
tetley% javac Duck.java
```

is probably required.

This step will also verify that you have copied the file correctly, or at least that your version compiles correctly. Any errors that might be reported should obviously be corrected before carrying on. A missing bracket is much easier to spot and correct at this very early stage and it is quite senseless to start writing a program that makes use of a class that does not compile.

The declaration of the *`Duck`* objects is simply:

```
Duck aDuck = new Duck();
```

A simple dialogue can be used to get the user to enter the name and initial position of the duck. Some temporary variables are needed to store these values before they are saved in the object instance; there is no way to access the attribute values themselves as they are defined as private. If we use the `Console` class we have already encountered, then the code for the dialogue might be:

```
int x, y;
String name;

// get details of duck

System.out.print ("Enter the duck's name: ");
name = Console.readString ();
System.out.print ("Enter " + name + "s x position: ");
x = Console.readInt ();
```

```
System.out.print ("Enter " + name + "\'s y position: ");
y = Console.readInt ();
```

These values are then copied into *aDuck* using the method calls:

```
aDuck.setName (name);
aDuck.moveTo (x, y);
```

Once the values have been entered and saved the location of the duck can be printed:

```
aDuck.printNameAndLocation ();
```

And a similar call to the *getStatus* method will display a status message.

Now, as Mr Martinmere often spends time walking around his reserve, it would of course be incredibly useful for the status messages to appear on Bruce's sign, since Mr Martinmere's desktop computer can be several hundred yards away! This is actually very simple to add to the program; the *printNameAndLocation* method provides a suitable string that can simply be displayed on Bruce's sign.

It's so easy that we might as well do it[2]. The complete program would be:

```
/* DuckTracker.java - Mr Martinmere's program to
                      keep track of his ducks.

   Author    : AMJ
   Date      : 31st December 2002
   Tested on : Red Hat 7.3, JDK 1.4.0
*/

import htpuj.*;

public class DuckTracker
{
  private Duck aDuck;

  public DuckTracker ()
  {
    aDuck = new Duck ();
  }

  public void readDuckInfo ()
  {
    int x, y;
    String name;

    System.out.print ("Enter the duck's name: ");
    name = Console.readString ();
    System.out.print ("Enter " + name + "'s x position: ");
    x = Console.readInt ();
    System.out.print ("Enter " + name + "'s y position: ");
    y = Console.readInt ();

    aDuck.setName (name);
    aDuck.moveTo (x, y);
  }

  public String getStatus ()
  {
    return aDuck.getStatus ();
  }
```

2 We would need the *class* file for the sign to, of course.

```
  public static void main(String args[])
  {
    DuckTracker tracker = new DuckTracker ();
    Sign brucesSign = new Sign ();

    // get ducky details

    tracker.readDuckInfo ();

    // print details

    brucesSign.setMessage (tracker.getStatus () );
    brucesSign.display ();
  }
}
```

This program highlights another advantage of using a class. The program does quite a lot (and behind the scenes some of it is quite complicated[3]) but it is still reasonably short; a lot of the processing is taking place in the implementation of the methods. Also, assuming you understand what they achieve, you can use methods that use programming techniques that you yourself have not learned. The method to print a warning if a duck moves too far away certainly uses Java that you have not seen yet.

If you look again at the definition of the class you will see that this program does not use all of the methods defined in this class; this is quite normal. The class will be developed so that it can hopefully be used in many programs and it is far from likely that any one program will need to use all the methods.

Example 2 – Coots have classes too

Mr Martinmere also uses programs to keep track of the coots on the pond. He is especially interested in the value of each coot, something that is determined by a range of factors combined in incredibly complex formulas. He wants a program that will take the details of two coots and will calculate and display the difference in value between the two.

Bruce has developed a Java class to store details of coots. Its definition is:

```
/* Coot.java - A simple Coot Class.

   Author    : Bruce, after AMJ
   Date      : 31st December 2002
   Tested on : Red Hat 7.3, JDK 1.4.0
*/

public class Coot
{
  // Attributes

  private String name;
  private int x, y;
  private int age;
  private int height;     // in cm
  private double weight; // in oz

  // Constructor

  public Coot()
  {
  }

  // Methods
```

3 Look at the implementationn of the `distanceFrom` method, for example.

```
    public void print()
    {
      System.out.println (name + ": Age: " + age + ", Height: "
                          + height + ", Weight: " + weight);
    }
    public void moveTo (int newX, int newY)
    {
      x = newX;
      y = newY;
    }
    public void setName (String newName)
    {
      name = newName;
    }
    public void setFactors (int newAge, int newHeight,
                            double newWeight)
    {
      age = newAge;
      height = newHeight;
      weight = newWeight;
    }
    public double getValue ()
    {
      if (age > 1) {
        return (height - weight) / age - 1;
      } else
      {
        return 0;
      }
    }
  }
```

Using the information from this file write the program that Mr Martinmere requires. You can find an implementation of this class on the web site.

The program follows a pattern similar to the one in the previous exercise. The mechanical step is much the same; the *.java* file defining the class is compiled and the resulting bytecode (in the *.class* file) is placed in the same directory as the program. The two required object instances (*cootOne* and *cootTwo*) are declared in the usual way.

A similar dialogue would be used to get the values for the three relevant factors from the user and store them in temporary variables. There would then be a method call to *setFactors* to set the attributes and determine the value, all done without our knowing anything about the incredibly complex formulas being used.[4] It seems very likely that these factors correspond to three attributes in the private part of the class but there is no way to know this for certain. In any event this internal detail is quite unimportant to someone using the class; all the information needed is available without looking at the details of the implementation. Care would be needed when using *setFactors* to make sure that the parameters are supplied in the expected order; this order is given in the comment and is age, then height, and then weight.

Similarly it seems likely that the value is stored as an attribute (it isn't, as it happens!) but again there is no way to be sure. What matters is that the

4 If you take a look, the formulas are not in fact *that* complex.

getValue method returns the value and that the *valueDifference* method has access to the correct figures. In fact for this program only the second of these is needed (although we might expect that *valueDifference* makes use of *getValue* (it does!); the difference in value can be printed simply:

```
System.out.println("The difference in value is " +
                   cootOne.valueDifference(cootTwo)
                   + ".");
```

Of course this happens to be identical to:

```
System.out.println("The difference in value is " +
                   + cootTwo.valueDifference(cootOne)
                   + ".");
```

It would not matter which was used.

Again there are many methods and possibly many attributes in this class that are not used. Also there are many details of the implementation of the class that a programmer using it needs to know nothing about (for example, the implementation of the `getValue` method). These things are quite usual.

The final program would be:

```
/* CootComparator.java - Compares relative values of two coots.

   Author     : AMJ
   Date       : 31st December 2002
   Tested on : Red Hat 7.3, JDK 1.4.0
*/

import htpuj.*;

public class CootComparator
{
  private Coot cootOne, cootTwo;
  public void readCoot (Coot c)
  {
    int age, height;
    double weight;

    // get details of first coot

    System.out.print ("Enter the age of the first coot: ");
    age = Console.readInt ();
    System.out.print ("Enter the height of the first coot: ");
    height = Console.readInt ();
    System.out.print("Enter the weight of the first coot: ");
    weight = Console.readDouble ();

    c.setFactors(age, height, weight);
  }

  public double valueDifference ()
  {
    return Math.abs (cootOne.getValue () - cootTwo.getValue () );
  }

  public static void main (String args[])
  {
    CootComparator comp = new CootComparator ();
    comp.readCoot (comp.cootOne);
    comp.readCoot (comp.cootTwo);
    System.out.println("The difference in value is " +
                       comp.valueDifference() + ".");
  }
}
```

12.1 Write a program using the `Coot` class from the second example that will prompt the user to enter the details of a coot and will print the value of that coot. The value should be calculated according to Mr Martinmere's incredibly complicated formulas. Remember that you can find the implementation of the class on the web site.

12.2 Examine the following extracts from a definition of a class to store details of geese. The implementations of all the methods have been removed. Write down a full description of the methods that are available for this class. What do you imagine is in the private part of this class? Does it matter that you don't have access to it?

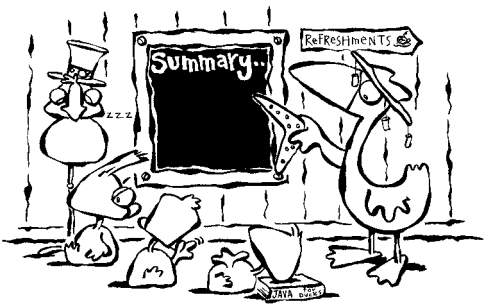
```
public Goose ()
public void setName (String newName)
public void setAge (int newAge)
public void setWeight (double newWeight)
public String getName ()
public int getAge ()
public double getWeight ()
public double getValue ()
public void moveTo (int newX, int newY)
public double distanceFrom (int atX, int atY)
```

12.3 Write a program that uses the `Goose` class above. The program should prompt the user to enter the name, age, and weight of a goose and should print the value of the goose in a message of the form:

<Goose Name> is worth <Value>.

Assume that the `Goose` class is defined in the file `Goose.java`. You can find an implementation of the class on the web site, or you can probably deduce the implementation from the ones you have for ducks and coots.

12.4 Write another program that could be used to keep track of ducks. The program should prompt the user to enter a duck's name and current position. The program should then display the distance the duck is away from the Bruce's shed, which is located at position (2, 5).



This chapter has explained how to make use of Java classes in your programs. The classes used have been quite simple but the basic ideas and syntax remain the same for all more complicated classes. By making use of classes your programs are now able to achieve quite complex results.

The process for using a class is straightforward. All that you need to do is take the `.java` file and compile it to produce the corresponding bytecode in the `.class` file. This done, the class can be used and instances (called objects) can be declared. The syntax for declaring objects is very similar to that for declaring variables of the other basic types such as integers, with the small addition of a call to the constructor.

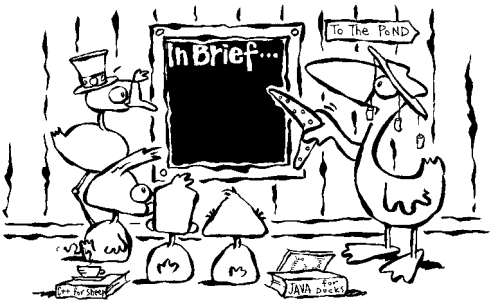
The operations that can be carried out on objects are defined in the `.java` file, and a program may make use of anything defined to be public. A programmer using the class has no need to know anything about the private parts of the class or indeed anything about how the methods are implemented. All the information that is needed is conveyed succinctly, accurately, and unambiguously in the file. Where there is any possible ambiguity this should be resolved with comments; it should never be necessary to examine the statements inside the methods.

A particular advantage that you have seen of this approach in this chapter has been that you have been able to make use of methods that use Java statements and techniques that you have not yet learned.

You should now also have learned how to compile the file containing your program together with a file containing the implementation of the class used in the program. Of course most programs will make use of many classes, but the ideas and the mechanical details are all the same.

We've mentioned the idea of *data hiding* a few times in these chapters. Now it's time to look at exactly what this is …

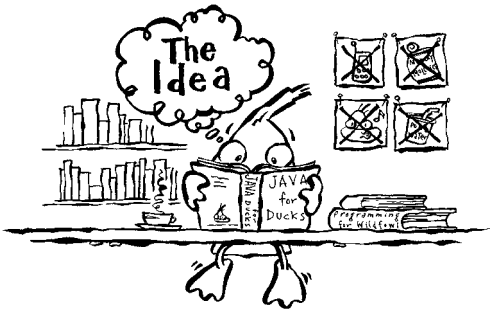

Chapter 13
Get Your Hands Off My Data
Recipe
shuffle
Public Interface
Private Attributes



By now you should have had plenty of practice at writing simple Java programs and classes. Some of these programs will have made use of classes developed by other programmers and you should have become accustomed to the way that Java defines and implements a class. The definition describes the public interface to the class and also includes the implementations of the methods themselves. In the last two chapters you got plenty of practice in developing your own classes, and you should now be able to develop your own classes for other programmers to use.

As you define a class you are specifying a number of things. One of these is the attributes that objects of your class have, and another is the ways in which programs can interact with and manipulate these attributes. It is important that you retain control over the values of the attributes to guard against, for example, incorrect programs setting them to invalid values. This chapter introduces a mechanism to do just that; *data hiding* makes sure that you are always well in control of the values of your attributes.

After reading this chapter you should understand how to implement simple methods for Java classes to support data hiding. You should also understand the concepts underlying data hiding and you should understand why this is very important and how the separation of Java classes into public and private parts allows programmers to implement this powerful mechanism.



In the last chapter we saw some very basic types of method. Some of these types are useful in all classes. Obviously every class needs a constructor, but there are also two other types of method that are generally almost as important. Methods are needed that allow a program to access the value of an attribute; these are commonly called *accessors* (or *selectors*). Similar methods, usually called *mutators*, are needed to allow a program to alter the value of an attribute.

There are, then, three basic types of method that will be found in most classes:

- *constructor* – the method that actually creates an instance of a class when it is declared;
- *selector* – a method that allows access to a private value from within the class by returning its value (sometimes called an *accessor*);
- *mutator* – a method that allows access to a private value of the class so that that value can be changed.

These three types of method will be found in almost every class that is ever written; every class must have a constructor and it is hard to imagine a useful class that did not allow any form of access to its private data. Each of the classes in the previous two chapters had these methods, and if you worked through the implementations of the classes in the exercises and examples you will have seen and written some of each. There is nothing new here.

There are, of course, other methods in any class. These tend to be more specialised than selectors and mutators and can only rarely be used in different classes. What is important here is that selectors and mutators allow for data hiding, a mechanism that keeps the developer of a class in control of the accuracy and integrity of the values in the class's attributes.

Data hiding

The key idea in the chapter is data hiding. This is also one of the key concepts in object-oriented programming, and is another reason why object-oriented programming is so popular.

The idea is quite simple. A programmer using a class needs to have no knowledge of how that class is actually implemented, of how the private attributes are actually stored. In Java terms this means that a programmer needs to only know the public interface to the class and has no need to know anything at all about the private parts. We have seen that it is possible to write a program that uses a class by examining only its methods; there is no need to worry about what attributes are stored.

So a programmer developing a class can specify exactly how that class can be used. If the class is written and tested well it should not be possible to use it in a way that would produce incorrect results. Testing is also made easier because the interface is probably quite small; it is much easier to thoroughly test a small well-defined interface than it is to test a large unstructured program.

There is another potential benefit. If a programmer using a class needs to know nothing of the way in which the class itself is stored then this storage can be changed if needed. It can be changed and programs using the class will still work and will be totally unaware of the change as long as the methods making up the public interface continue to behave as before.

Data hiding is achieved in Java with selectors and mutators. These control access to the private part of the class and mean that the attributes in these private parts can be changed without the change causing any problems to programs using the class. Let's look at each in turn.

Selectors

Since it is part of a class, a selector can access the private parts of the class. A selector's job is to provide the value of a given attribute to the program that

calls the method. The format of the declaration in the *.java* file follows a predictable format:

public <return type> <method name> ()

Selectors must be defined in the public part of the class, or they would be of very little use! They do not have parameters so there is also an empty set of brackets. The return type, which is the type of the attribute that is being returned, is added to the start of the line. A common convention, and one that is used in all the programs here, is that the name of the selector method is made up of the word *get* followed by the identifier of the attribute that is returned.

For example, in a simple *Duck* class, the *getName* method would return the name of the duck, a string:

```
public String getName ()
```

The implementation of a selector is usually only one line. The value required is just returned. So in this example:

```
public String getName ()
{
  return name;
}
```

A similar example could be used to provide the value of a duck:

```
public double getValue ()
{
  return value;
}
```

In both these examples, the selector is returning a value that is presumably stored explicitly in an attribute. But selectors do not necessarily have to return a value that is actually stored in the private part of the class. It can sometimes be useful and very powerful to implement selectors that do not do this; selectors can return a value that is derived from one or more of the other values in the class. You might remember that we suspected that some of the *getValue* methods that we have met before were doing just this when the value was based on a complex formula.

Suppose that a user of a *Duck* class often wanted to use the values of the duck after some tax (calculated as some known percentage of the actual value) had been added. One approach to this would be to add a new private attribute to store the price plus tax and to allow access to it via a suitable pair of a selector and mutator. There is a drawback with this; the value and the value after tax are the same information stored in different ways and so the mutator that changed one would have to change the other. If the value were reduced then the value after tax would have to be reduced too. This is a needlessly complex solution and one that would be best avoided.

Far better would be to simply create a new selector to return the value after tax. This can be added to the header file:

```
public double getValuePlusTax ()
```

and, assuming that the rate of tax is stored in the *final* variable *TAX_RATE*, it can be implemented:

```
public double getValuePlusTax ()
```

```
{
  return value * (1.0 + TAX_RATE);
}
```

A programmer using the class does not need to know whether the value with tax is stored as a private member or not (as we did not when we encountered the complex formulas). As before, the programmer just has to understand and use the public interface to the class.

This solution also makes the job of writing the mutators more straightforward, so let's have a look at how these work.

Mutators

Mutators change values of attributes and are very similar to selectors. The main difference is that they each have the new value for the private attribute provided as a parameter. This parameter is listed together with its type (and is given an identifier) in the brackets after the method's name. The heading line is then complete:

```
public void setName (String newName)
```

It is useful to use a similar convention for naming mutators to that used for selectors. Here we will use *set* followed by the name of the attribute that is affected, so *setName* changes the value of the *name* attribute.

The declaration of the parameter in the brackets looks rather like the declaration of a variable and this is in fact just what it is. The parameter is available inside the method in just the same way as a variable. Again, a convention for naming the parameter is a good thing; here we use the word *new* followed again by the name of the attribute.

The body of the mutator method takes the value of the parameter and assigns it to the private attribute. Finally, the method has no particular need to return a value, so it is declared as `void`. The full method is now:

```
public void setName (String newName)
{
  name = newName;
}
```

This is a very simple example, and one that sets the new value without bothering to check it; the assumption is that the string provided is a valid name for a duck. But suppose that a similar method had been written to set a duck's age:

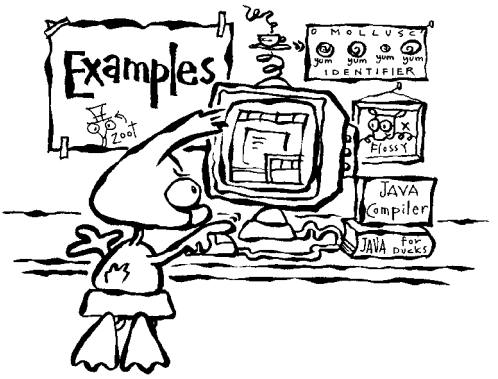
```
public void setAge (int newAge)
{
  age = newAge;
}
```

This is fine, and it would work. The problem would come when a program tried to use it with an invalid value; this method would quite cheerfully set the age of a duck to 400 (which is unlikely) or –1 (which is clearly invalid). A slightly more sophisticated mechanism is needed.

The solution is to check the value before it is used, and to set it only if it is found to be valid. That needs some Java that we've not seen yet, so we'll see how to do that in the next chapter.

Basic methods

Selectors and mutators, together with the constructor, are the most fundamental methods to be implemented for any class. It's time for some more practice.



Example 1 – More from Bruce's library

Bruce's first efforts to produce a program to store details of the borrowers from his library produced this `.java` *file:*

```
public class Borrower {
  private String name; // Name of borrower
  private int number;  // Unique borrower number

  public Borrower ()

  {

  }

  public void print ()
  {
    System.out.println ("Borrower: " + name);
    System.out.println ("Number:   " + number);
  }
}
```

He now wants to extend this class to include suitable selectors and mutators. What should the complete implementation be?

The class definition needs two selectors and two mutators, one of each for each of the private attributes. This also seems to be a good time to add some comments into the file.

```
/* Borrower.java - A class to store borrower details for
                   Bruce's library.
   Author   : AMJ
   Date     : 6th February 2003
   Tested On: Red Hat 7.3, JDK 1.4.0
*/

public class Borrower
{
  private String name; // Name of borrower
  private int number;  // Unique borrower number

  // Constructor
```

```
    public Borrower ()
    {
    }

    // Print out details neatly formatted
    public void print ()
    {
      System.out.println ("Borrower: " + name);
      System.out.println ("Number: " + number);
    }

    // Selectors
    public String getName ()
    {
      return name;
    }

    public int getNumber ()
    {
      return number;
    }

    // Mutators

    public void setName (String newName)
    {
      name = newName;
    }

    public void setNumber (int newNumber)
    {
      number = newNumber;
    }
}
```

There seems to be no particular need for the constructor to set any default values and the selectors and mutators follow the usual patterns from this chapter. Simple.

Example 2 – More wandering ducks

The whereabouts of his ducks on his pond continues to be of great concern to Mr Martinmere. He has a program that uses a class, but the recent failure of his hard disk combined with his poor backup strategies means that he is forced to rewrite it!

Happily, he has found a very early printout of the class. This contains the headers for all the methods but is missing the implementations. Clearly they must be added back in. This is the file he has:

```
public class Duck {
  private String name;
  private int x, y;

  public Duck (String n) // All ducks start at 0,0

  // Selectors
  public String getName ()
  public int getX ()
  public int getY ()

  // Mutators
  public void setName (String n)
```

```
  public void setX (int newX)
  public void setY (int newY)

  // Other Methods

  // Move duck to newX, newY
  public void moveTo (int newX, int newY)
}
```

What does the full implementation look like?
Taking each method in turn, the first is the constructor. The comment in the file requires that each duck is given the initial position (0, 0). The name of the duck, we can assume from the constructor's parameter list, is obtained before the object is created, then passed as the only argument to the constructor. Thus the constructor is simply:

```
public Duck (String n)
{
  x = 0;
  y = 0;
  name = n;
}
```

The selectors and mutators all follow what is by now a very familiar pattern. An example of each:

```
public int getX ()
{
  return x;
}

public void setName (String n)
{
  name = n;
}
```

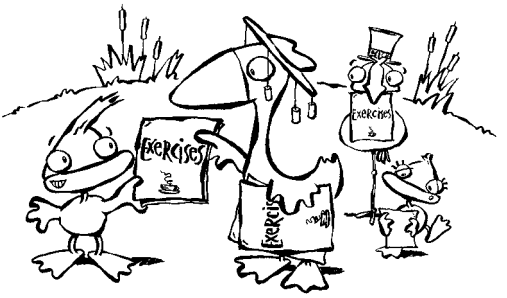
The final method is slightly more interesting:

```
// Move duck to newX, newY
public void moveTo (int newX, int newY)
```

This method moves a duck to a new position so it is effectively nothing more than a mutator that operates on two private attributes at the same time. We actually saw it in the previous chapter, but it's worth looking at again here. It is implemented:

```
public void moveTo (int newX, int newY)
{
  x = newX;
  y = newY;
}
```

A side issue here is that this implementation of the class would work with any existing programs that used the implementation that Mr Martinmere has lost as a result of his unfortunate accident. The public interface hasn't been changed, and it doesn't matter whether the implementation is identical to the lost one just as long as it works correctly.



13.1 A class written to process the details of coots has the following definition:

```
public class Coot
{
  private String name;
  private int age;
  private double value; // in UK Pounds

  public Coot ()        // sets no default values

  public String getName ()
  public int getAge ()
  public double getValue ()

  public void setName (String)
  public void setAge (int)
  public void setValue (double)

}
```

Fill in the implementations.

13.2 Now write a short program that will allow you to test the class. The program should prompt a user to enter some values for the attributes of an instance of the class and should then use and test all of the methods in turn. We have seen programs such as this before; they are called driver programs, and such programs are often written to test a class. The *main* method would be a good place to put it.

13.3 A program that has been developed to test this class contains the following lines of code. What error has been made? How should the error be corrected?

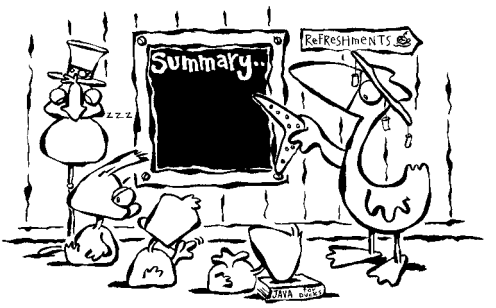
```
Coot zoot;
zoot.setName ("Zoot");

System.out.println ("Name has been sent to " + zoot.name);
```

13.4 There is no error in this skeleton of a class definition but there is some poor design. Explain the poor design and write a corrected skeleton file.

```
public class Coot
{
  private String name;
  private int age;
  private double value;
  private double valuePlusTax; // value plus 15%

  public Coot ()
```

```
    {
    }
    public String getName ()
    {
      return name;
    }
    public int getAge ()
    {
      return age;
    }
    public double getValue ()
    {
      return value;
    }
    public double getValuePlusTax ()
    {
      return valuePlusTax;
    }
    public void setName (String newName)
    {
      name = newName;
    }
    public void setAge (int newAge)
    {
      age = newAge;
    }
    public void setValue (double newValue)
    {
      value = newValue;
    }
    public void setValuePlusTax (double newValuePlusTax)
    {
      valuePlusTax = newValuePlusTax;
    }
  }
```

13.5 In the examples you have seen so far there have been mutators and selectors for each and every private attribute. Is this always sensible? Do you think that mutators and selectors should be added to the design of every class?

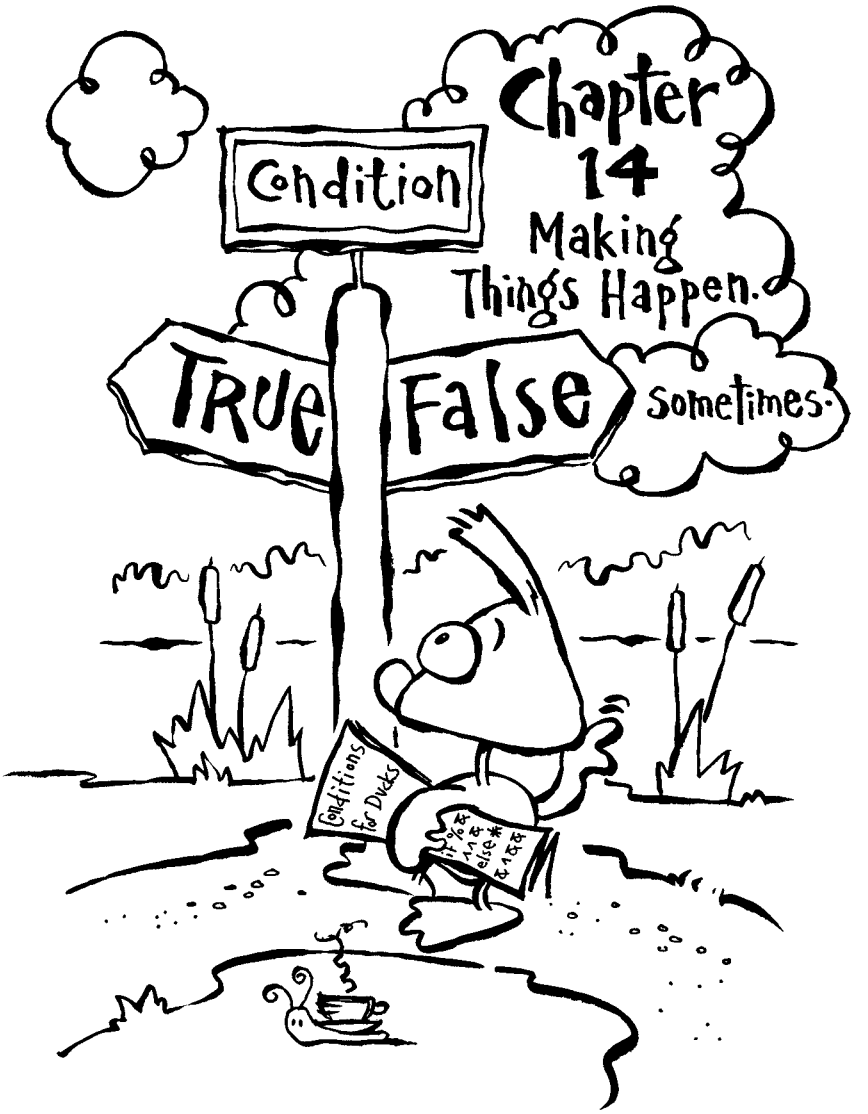


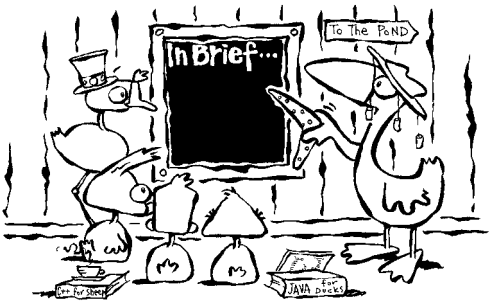
You have now seen and used all aspects of creating a Java class. You should be able to define a class and then develop a suitable implementation, including selectors and mutators. Finally you should also be able to write programs that make use of your class.

You should understand the concept of data hiding. You should know why it is important that access to the private attributes of a class is controlled and why such access should be allowed only through the class's public interface. This interface is implemented in Java with selectors and mutators; these are simple methods that either return or change the value of a private attribute.

Although you have seen only simple examples of methods so far the details you have seen and practised remain the same as methods become more complicated. You have in fact seen all the Java that is used to define, implement, and use very simple classes.

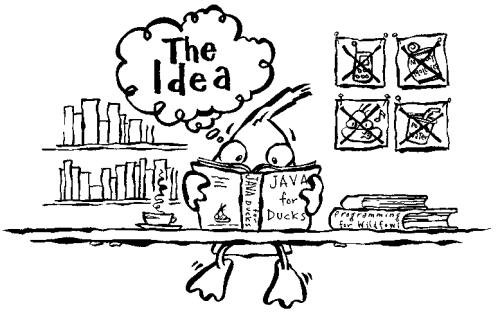
The remainder of this book will show you Java that will allow you to write more and more complicated and useful programs.


Chapter 14
Making Things Happen.
Condition
TRUE
False
Sometimes.
Conditions for Ducks



Up to this point the behaviour of your programs has been very predictable. The programs have been simply a sequence of statements that is executed in a strict order from the top of the program to the bottom. While you have been able to accept input values from your users all you have been able to do is use these values in calculations. You have not been able to use them as the basis of decisions. More complex programs take values from the user and will behave differently depending on the values that are entered. This chapter is about writing programs that do just that; they take values, test *conditions*, and then behave differently depending on the results.

After reading this chapter you should understand how to include conditional statements in your programs. You should understand the concept of a Boolean value more fully and you should be able to use Boolean variables and expressions in your programs. You should be able to combine these values using Boolean operators and you should be able to use these values and operators in conditional statements. You should understand and be able to use the Java `if` and `switch` statements.



A *condition* is a statement that is either true or false. Programs need to be able to test conditions so that they can behave differently in different situations. This is achieved with *conditional statements*; these are statements that test a condition and then choose between two (or more, in fact) different possible behaviours depending on whether the condition was found to be true or false. Conditional statements can be combined to allow programs to make a choice between more than two possible behaviours.

There are a couple of types of conditional statements in Java (and many other languages, for that matter). The most common is the `if` statement; there is also the `switch` statement that provides a shorthand form for a particular type of `if` statement. It is possible to write all `switch` statements as `if` statements,

but the `switch` statement can be more elegant and neater. Before using either of these statements, however, you need to understand the concepts of true and false and how these can be stored and expressed in Boolean variables.

True and false

Some statements are always either true or false. Here are some examples:

Statements that are True

- Elvis is a duck.
- Potatoes grow in the ground.
- Christmas Day is the 25th of December.
- Ducks live in ponds and eat molluscs.

Statements that are False

- Elvis is a gibbon.
- Potatoes grow on trees.
- Christmas Day is the 14th of June.
- Ducks live underground and eat curry.

All of the statements are either true or otherwise they are false. They must be one or the other. Their "true-ness" will never change no matter what happens.

Some statements are more complicated. Some statements are true some of the time and false at other times. They are always one or the other and never, of course, both. Here are some examples:

- Elvis is swimming around on his pond.
- I have some potatoes in my kitchen.
- It is Christmas Day tomorrow.
- There are six ducks on the pond and they are all easting molluscs.

At any point in time these statements will either be true or false. Their "true-ness" depends on some condition:

- Elvis might be swimming around on his pond, or he might not be.
- I may have some potatoes in my kitchen, or I may have run out.
- It may be Christmas Day tomorrow, or it might be the middle of June.
- There may be six ducks on the pond but some more may arrive, or some may go away. The ducks that are there may be eating molluscs, or some or all of them may stop. There are actually two conditions here – the number of ducks on the pond and whether or not they are eating molluscs.

In order to determine the "true-ness" of the statement it is necessary in each case to test this condition. This test could be expressed in pseudocode, for example:

```
IF IT IS DECEMBER 24TH TODAY THEN
  IT IS CHRISTMAS DAY TOMORROW
OTHERWISE
```

```
  IT IS NOT CHRISTMAS DAY TOMORROW
END IF
```

Or perhaps:

```
IF IT IS DECEMBER 24TH TODAY THEN
  IT IS TRUE THAT IT IS CHRISTMAS DAY TOMORROW
OTHERWISE
  IT IS FALSE THAT IT IS CHRISTMAS DAY TOMORROW
END IF
```

Or in more general terms:

```
IF THE CONDITION IS TRUE THEN
  THE STATEMENT IS TRUE
OTHERWISE
  THE STATEMENT IS FALSE
END IF
```

This is totally unambiguous; the condition is either true or it is false and so, therefore, is the statement. Neither can ever be both true and false and neither can ever have some other value.

Boolean variables and operators

In Java, Boolean variables are variables that can store one of two values, `true` or `false`. The data type is `boolean` and they are declared in the same way as all other variables:

```
boolean finished;
```

Boolean variables can only be assigned one of the two possible values:

```
finished = true;
finished = false;
```

On their own Boolean variables are of very limited use, which is why we have not made any use of them so far. It is not possible to read values for them from the user's keyboard and it is not possible to output their values directly. Their power comes when they are used in *Boolean expressions*.

Variables of the more familiar numeric types have a number of operators that can be used with them. You have seen (and hopefully written) many programs where integers and floating-point values have been added together, subtracted, multiplied, or divided. Boolean values also have a set of operators; their meaning is similar to the operators for the numeric types, but the meaning is now set in terms of logical operations. These operators allow Boolean variables and values to be combined in various ways based on the principles of mathematical logic. You may well have met them before in a mathematics course.[1]

There are three basic Boolean operators:

---

1 You might also have met them in a computer hardware or electronics course. Combining Boolean operations is effectively the same as combining currents with "gates" in a circuit.

and

The "and" operator combines two Boolean values to give the result `true` if and only if the two values are both `true`. Boolean expressions are often written out in *truth tables* – here is the truth table for the "and" operator.

First	Second	First *and* Second
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

or

The "or" operator combines two Boolean values to give the result `true` if and only if either (or indeed both) of the two values is `true`:

First	Second	First *or* Second
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

not

The "not" operator takes just one Boolean value and inverts it. It gives the result `true` if the original value is `false` and vice versa:

Original	*not* Original
`false`	`true`
`true`	`false`

These three operators have the following symbols in Java:

and	`&&`
or	`||`
not	`!`

The symbol used in the "or" is the "pipe" symbol, which you may not have met before. This is normally found on the UK keyboard to the left of the *z* key or else just above the *TAB*. It may well be shown on your keyboard as two vertical lines with a gap between them.

The symbol for "not" is the exclamation mark. For reasons that are not especially clear in computing circles it is normally pronounced "bang".

The two operators that take two values ("and" and "or") are used in much the same way as the similar mathematical operators for the numeric data types. The "not" operator is simply added before the identifier of the variable:

```
boolean conditionOne = true;
boolean conditionTwo = false;
```

```
// (conditionOne && conditionTwo) is false
// (condition One || conditionTwo) is true
// !conditionOne is false, !conditionTwo is true
```

Boolean values expressed in this way are the key to using conditional statements.

Comparison operators

The key to using conditional statements is that they give the ability to compare two values. A given value must be compared to some expected value to build an expression which can then be used to determine whether a condition is true or false. This comparison requires some operators.

The following operators are available in Java. The symbols and meanings of most of them are much the same as you are probably familiar with from mathematics.

==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

A common error is to confuse the "equal to" comparison operator == with the assignment operator =. You have been warned! Remember that a good way of remembering the difference is to get into the habit of reading the assignment operator as "becomes" or "is assigned" rather than the potentially misleading "equals". For the same reason you might want to read the comparison operator as "is the same as".

The meaning of each of these operators should be fairly obvious, but they are illustrated in this sample code:

```
int first, second;

first = 10;   // first is assigned 10
second = 20;  // second is assigned 20

// first == 10       this is true, first equals 10
// first == second   this is false, first does not equal second
// second != 10      this is true, second does not equal 10
// first > 0         this is true, first is greater than 0
// second < 0        this is false, second is not less than 0
// second < 20       this is also false, second is not less
                     than 20
// second <= 20      but this is true, second is equal to 20
// first >= 10       this is true, first is equal to 10
```

Each one of these comparisons is itself actually a Boolean value (or, more correctly, a Boolean expression); each one is either true or false. This means that it is possible to assign such a comparison to a Boolean variable. For example:

```
int first = 10;
boolean firstIsTen;

firstIsTen = (first == 10); // assigns true
```

The brackets in this assignment are not essential but they do make the line a lot clearer. Some more examples:

```
int aNumber = 10;
boolean condition;

condition = (aNumber > 0);        // assigns true
condition = (aNumber <= 100);     // assigns true
condition = (aNumber != 10);      // assigns false
condition = (aNumber == 0);       // assigns false
```

Conditions, comparisons, and Boolean variables are only really useful if they can be used in conditional statements. So it's time to look at the first of these.

The `if` statement

In its simplest form the `if` statement evaluates a condition and executes the statements following it only if the condition is found to be true. An `if` statement starts with the word `if` which is then followed with the condition in brackets:

if (<condition>)

The statements affected by the `if` statement are below, enclosed in the customary braces:

```
if (<condition>) {
   // statements executed only if condition is true
}
```

If there is only a single statement inside the braces then they can be omitted, but it is always good practice to include them to make sure that things are clear and to avoid the possibility of an error if more statements are added in the future. The statements inside the braces should also always be indented to clearly show that the conditional statement controls them.

For example, if a program were required to divide two numbers it is important to check that the divisor is not equal to 0 since dividing by 0 gives an error (an error that we have already seen in Chapter 10[2]). This can be achieved simply enough with an `if` statement:

```
double top, bottom;

System.out.print ("Enter the first number: ");
top = Console.readInt ();
System.out.print("Enter the second number: ");
bottom = Console.readInt ();

if (bottom != 0) {
   System.out.print(top + " divided by " + bottom + " is ");
   System.out.println(top / bottom);
}
```

2 Where I promised that here in Chapter 14 we would see how to avoid this error. And here we are!

This makes sure that the calculation takes place only if it is safe to do so. The statements inside the braces are executed only if the condition:

```
bottom != 0
```

is true, so they are executed only if the divisor does not equal 0.

Of course if the divisor was 0 it would be useful to print a message to the user telling them why no output has been produced; as matters stand the user would just be left with no response at all from the program, a very bad example of human–computer interaction. This can be done using the optional `else` clause of an `if` statement.

The `else` clause of the `if` statement

The `else` clause of the `if` statement specifies the statements that will be executed if the condition is found to be false. These statements are enclosed as usual by curly brackets and are also indented:

```
if (<condition>) {
   // statements executed if condition is true
}
else {
   // statements executed if condition is false
}
```

A more complete statement to check that a division by 0 does not happen would be:

```
if (bottom != 0) {
   System.out.print(top + " divided by " + bottom + " is ");
   System.out.println(top / bottom);
}
else {
   System.out.println("Error! Cannot Divide by 0!");
}
```

Since there is a choice involved here, this could also (and equally well) be written:

```
if (bottom == 0) {
   System.out.println("Error! Cannot Divide by 0!");
}
else {
   System.out.print(top + " divided by " + bottom + " is ");
   System.out.println(top / bottom);
}
```

The difference in the second example is that the condition that is tested has been "reversed". There is no functional difference between these two examples.

The condition in the brackets of an `if` statement will often combine two connected tests. Suppose that a program requires that the user enter a value between a certain range such as 0 and 20; the program must test the value and reject it if it is less than 0 or if it is greater than 20.

The description of the problem shows that this will be an application for the Boolean "or" operator. Assuming that the range is inclusive the code is:

```
int aNumber;
```

```
System.out.print("Enter a number between 0 and 20: ");
aNumber = Console.readInt ();

if (aNumber < 0 || aNumber > 20) {
   System.out.println("Error: Invalid Value!");
}
else {
   // Process valid value
}
```

A common error is to write a statement such as this in this way:

```
if (aNumber < 0 || > 20) // THIS IS AN ERROR!!
```

While this may make sense, especially when read aloud, it is always an error. The reason should be obvious. The value to the right of the "or" operator is "> 20"; this is not a properly formed Boolean condition since there is nothing for the 20 to be compared to. It should be obvious that this is an error:

```
if (> 20) // THIS IS ALSO AN ERROR!!
```

and this is the exact condition that is on the right-hand side of the "or" operator in this example.

While validating this input, much the same effect could be achieved by accepting only those values that are greater than or equal to 0 and less than or equal to 20. Once again the description contains the name of the required Boolean operator, in this case "and":

```
if (aNumber >= 0 && aNumber <= 20) {
   // Process valid value
}
else {
   System.out.println("Error: Invalid Value!");
}
```

This example is exactly equivalent to this:

```
if (aNumber > -1 && aNumber < 21) {
   // Process valid value
}
else {
   System.out.println("Error: Invalid Value!");
}
```

The choice of which of these possibilities to use is largely a matter of the programmer's taste and personal programming style. With experience, in most cases one possibility will feel more natural but this does not mean that any of the others are ever in any sense wrong.

More complex `if` statements

Sometimes the choice to be made is more complex. There may be three possible values for some variable and three possible paths for the program to follow. There may be many more. The `if` statement is easily extended to deal with this:

```
if (<condition1>) {
   // statements if condition1 is true
}
```

```
else if (<condition2>){
    // statements if condition1 is false but condition2 is true
}
else {
    // statements if condition1 and condition2 are both false
}
```

Only one of the three possibilities is ever executed. If the first condition is true the first set of statements is used. The second condition is only tested if the first condition is found to be false; the statements that it controls are executed only if the second condition is found to be true. Finally, if both conditions are false, the final set of statements will be used. One of the three sets of statements is therefore always executed.

As an example of using a more complex `if` statement, consider the problem of printing a date. Suppose the date is stored as three integers, one each for the day, the month, and the year and that the required format prints the day part as "1st", "2nd", "3rd", and so on. The code to generate the required two letters to follow the number would be a single long `if` statement. Assuming that the day is held in the integer variable *`day`*, one possibility to output the day part of a date would be:

```
System.out.print(day);

if (day == 1 || day == 21 || day == 31) {
   System.out.println ("st");
}
else if (day == 2 || day == 22) {
   System.out.println ("nd");
}
else if (day == 3 || day == 23) {
   System.out.println ("rd");
}
else {
   System.out.println ("th");
}
```

There can be any number of conditions tested in this way. There does not need to be a final `else` section and there need not be if the tests beforehand cover every relevant possibility. If there is no `else` section it is then possible that none of the possibilities will ever be executed.

This form of the `if` statement is so common that there is a different conditional statement available; the `switch` statement provides a neat mechanism for particular types of comparisons. That said, the `switch` statement does nothing that cannot be expressed with a suitable `if` statement.

The `switch` statement

The `switch` statement provides no control that an `if` statement does not; it simply allows for neater code. It is used in the particular case where there is a set of possible values for some variable (called the *control variable*) and different actions must be taken for each one. The format is:

```
switch (<control variable>) {
   case <value 1>: // actions if variable is value 1
                        break;
```

```
    case <value 2>: // actions if variable is value 2
                                break;
    default: // actions if variable had none of the values listed
}
```

Since it must be possible to list all the possible values of the control variable, if follows that the variable must be a `char`, an `int` or one of the other integer types (`long` and `short`, neither of which we will meet in this text).[3] It is not possible to list all possible strings, or all possible floating-point numbers. The effect of the statement is that the case containing the value is found, and the statements found there are executed. The end of the case is marked with `break`, and execution stops.

A `switch` statement would be suitable for printing the name of the month from a numeric representation of a date. There would be twelve possibilities, one for each month. Assuming that the month is stored in an integer variable called *`month`* the code would be:

```
switch (month) {
   case 1: System.out.print ("January");
           break;
   case 2: System.out.print ("February");
           break;
   case 3: System.out.print ("March");
           break;
   case 4: System.out.print ("April");
           break;
   // and so on for the other months
}
```

There would be no need for a `default` section in this case because all the possible values would presumably be known and listed. If it were possible that an invalid value could be held in *`month`* there are two possibilities. One is to add a `default` section:

```
default: System.out.println ("Error: Invalid Month!");
```

A potentially neater alternative is to keep the `switch` statement as it stands and to use it only if the value is found to be valid:

```
if (month >= 1 && month <= 12) {
   switch (month) {
      // switch cases
   }
}
```

An error message can be added to this example too:

```
if (month >= 1 && month <= 12) {
   switch (month) {
      // switch cases
```

3 Formally, these are called *ordinal* types since their values can be listed and ordered. Booleans are ordinal too, but since there are only two possible values there is little point in using a `boolean` in a `switch` statement.

```
    }
  }
  else
  {
    System.out.println("Error: Invalid Month!");
  }
```

Only variables whose possible values can be listed can be used in `switch` statements; this effectively means only integers and characters. The `default` case means that it is not necessary to list all the possible values, of course.

As a further example here is a `switch` statement that would expand the names of the points of the compass held in the `char` variable *`point`*:

```
switch (point) {
   case 'N': System.out.print ("North");
             break;
   case 'S': System.out.print ("South");
             break;
   case 'E': System.out.print ("East");
             break;
   case 'W': System.out.print ("West");
             break;
}
```

Sometimes the same statement is required for more than one value. If in this compass example the character could be in upper or lower case then the same statement should be executed for 'n' as for 'N', and so on. It would be wasteful to duplicate the code, so more than one value can be attached to a case:

```
switch (point) {
   case 'N':
   case 'n': System.out.print ("North");
             break;
   case 'S':
   case 's': System.out.print ("South");
             break;
   case 'E':
   case 'e': System.out.print ("East");
             break;
   case 'W':
   case 'w': System.out.print ("West");
             break;
}
```

If a `switch` statement like this is used, it is important to make sure that the `break` lines are used correctly to separate the various cases. It is also clear why omitting a `break` in a simpler switch statement can produce all sorts of unexpected results.

The neatness of the `switch` statement is apparent from the `if` statement that is equivalent to this last example:

```
if (point == 'N' || point == 'n') {
   System.out.print ("North");
}
else if (point == 'S' || point == 's') {
   System.out.print ("South");
}
else if (point == 'E' || point == 'e') {
```

```
    System.out.print ("East");
  }
  else if (point == 'W' || point == 'w') {
    System.out.print ("West");
  }
```

At the same time the `switch` statement does nothing that an `if` cannot. Once again the choice of whether or not to use a `switch` statement is often just a matter of taste or preferred programming style.

Ending a program

Conditional statements will allow you to check that the values supplied by the users of your programs are expected. This raises the question of what to do if the values are unexpected; often the error will be fatal and there will be little point in carrying on asking for the rest of the values and making calculations.[4]

If a program requires a single value it is simple to just exit the program if an invalid value is provided – this is precisely what the example to trap a potential division by 0 did. When more than one value is required it is possible to use the same technique to achieve this effect with a sequence of conditional statements:

```
// get value 1
if (<value 1 is not valid>) {
    // display an error
}
else {
    // get value 2
    if (<value 2 is not valid>) {
        // display an error
    }
    else {
    // process the values
    .
    .
    .
    }
}
```

This will work but will quickly become very cumbersome and unwieldy, as more and more conditional statements have to be added for more values. Imagine what it would look like if there were 10 input values, or 100! The indentation would mean that the code would quickly disappear off the right of the page!

An alternative is to use a quick and immediate way to end a program. The `System.exit` command provides just that:

```
System.exit (0);
```

4 We will soon see a more elegant solution to this problem where we will be able to keep asking a user for values until they enter something correct. But for the moment something rather more brutal is required.

This terminates the program immediately without processing any more statements. The parameter in the brackets can be used to indicate why the program has exited abruptly; this value can sometimes be trapped by the system that called the program.

The code for dealing with invalid input values is now much simpler:

```
// get value 1
if (<value 1 is not valid>) {
    System.out.println ("Error: Value 1 not valid");
    System.exit (0);
}

// get value 2
if (<value 2 is not valid>) {
    System.out.println ("Error: Value 2 not valid");
    System.exit (0);
}
```

and so on. We will see how to use this technique in some of the examples.

A warning about comparing strings

Testing strings for equality in Java has a nasty habit of causing headaches. It may be tempting to compare them as you would compare other data types such as `int`, `char` or `boolean`, but would not normally give the correct results.

`String` is a built-in class provided with your Java packages; this is not the same thing as the other basic types (and is the reason why `String` starts with a capital letter). It is tempting to try to use strings in the same way as other types, and this will work some of the time. But using them in comparisons can lead to misleading results. Look at this code:

```
String s1 = "Elvis", s2 = "Elvis";
if (s1 == s2) {
   System.out.println ("s1 and s2 are equal");
}
else {
   System.out.println ("s1 and s2 are different");
}
```

The output from this, rather surprisingly, is:

```
s1 and s2 are different
```

The reason for this is subtle. In the case of the basic data types (`int`, `double`, `boolean`), the virtual machine knows to compare their values with the == operator. With classes, however, the == operator is used to compare the memory locations (or *references*) that the two operands refer to, and returns `true` if the two references are identical. Identical here means that they occupy the same part of the computer's memory. You might remember that we noticed a similar problem with using the assignment operator with objects.

So in the case of two `String` objects, == does not compare their values, but effectively checks whether the variables refer to the same instance of the `String` class. Since *`s1`* and *`s2`* are different instances, the program behaves as

above. This is known as comparing references, and will be examined again in Chapter 17.

Now, you may be wondering how it is possible to compare the values of two String objects. Happily the `String` class has been designed with this in mind, and provides a special method, `equals`, to compare two String objects.[5] It is used as follows:

```
String s1 = "Elvis", s2 = "Elvis";
if (s1.equals (s2) ) {
   System.out.println ("s1 and s2 are identical");
}
else {
   System.out.println ("s1 and s2 are different");
}
```

This time the output is as expected:

```
s1 and s2 are identical
```

There are several other useful methods included in the String class; the Java API lists them all, of course. One similar in use to the `equals` methods we have already met, is the `equalsIgnoreCase` method, which as the name suggests, compares two strings case-insensitively:

```
String s1 = "buddy", s2 = "Buddy";
if (s1.equalsIgnoreCase (s2) ) {
   System.out.println ("s1 and s2 match, ignoring case");
}
else {
   System.out.println ("s1 and s2 do not match");
}
```

It is worth spending a bit of time experimenting with simple programs that use strings to get used to how all this works. The API is a fine place to start.

Conditional statements and conditions

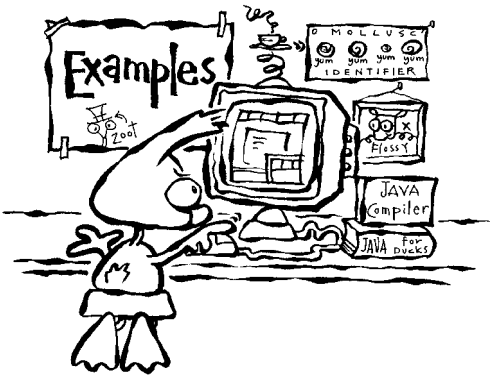
Conditional statements allow a great deal of control over the way in which programs handle the values supplied by users. It is now possible to identify and handle cases of values that would previously have caused serious errors. These errors can now be dealt with neatly useful messages being displayed to the user.

With conditional statements it is now possible to write programs that examine the values provided by users and work in different ways determined by these values.

It's time to practise doing this with the examples and exercises.

---

5 The `equals` method is like the `toString` method in that it is provided by default to all classes. The default action of this method is identical to == , but the programmer can define a different action if necessary, as the default behaviour is not always intuitive or sensible.



The conditional statements described in the chapter can be used to make some of the programs from previous chapters and examples much more robust. They can be made to handle unexpected input values and to provide useful feedback to their users if errors are made.

Example 1 – Cilla's cricket poser revisited

Cilla's cricket poser from Chapter 8 is a fine example of a program where there is a potential divide by 0 error. You might remember that Cilla is in charge of dividing the ducks up into their cricket teams; obviously there is a potential error if she tries to divide some number of ducks up into teams consisting of 0 ducks.

Cilla's program currently looks like this:

```
/* CricketScheduler.java - Cilla's cricket scheduling
   software

  Author     : GPH
  Date       : 9th June 2003
  Tested on  : Linux (Red Hat 8.0), JDK 1.4.1
*/
public class CricketScheduler
{
  public CricketScheduler () {}

  public int getNumTeams (int num, int size)
  {
    return num / size;
  }

  public int getRemainder (int num, int size)
  {
    return num % size;
  }

  public void printInfo (int num, int size)
  {
    System.out.println ("There are " + num
                        + " ducks available.");
    System.out.println ("This means "
                        + getNumTeams(num, size)
                        + " teams of " + size + ".");
    System.out.println ("This leaves "
```

```
                              + getRemainder(num, size)
                              + " substitutes.");
    }

    public static void main (String args[])
    {
      CricketScheduler cs = new CricketScheduler ();
      int numDucks = 36;
      final int TEAMSIZE = 11;

      cs.printInfo (numDucks, TEAMSIZE);
    }
}
```

Where is the potential error and what can Cilla do about it?

The error is fairly obviously in the method that does the division:

```
public int getNumTeams (int num, int size)
{
  return num / size;
}
```

This method (which is intentionally using integer division) will encounter an error if the parameter *size* is 0. This is not an especially likely error at the moment, especially as the size of the teams is currently defined as `final` and so cannot change. Nevertheless the method (or the whole class) might at some point be used in another program, so the possible error should be trapped and dealt with.

The check is a simple conditional statement:

```
if (size == 0) {
  // size is invalid
}
```

But the question now arises of what to return if the error is detected. The method has to return an integer, so presumably some integer value that indicates an error is required. For this example, we'll just make the method return −1, a value that is clearly nonsense. And just to be thorough we'll also make sure that any negative values of *size* (which are clearly also nonsense) also result in an error.

The method now becomes:

```
public int getNumTeams (int num, int size)
{
  if (size == 0 || size < 0) {
    return -1;
  }
  else {
    return num / size;
  }
}
```

Or, and perhaps more neatly:

```
public int getNumTeams (int num, int size)
{
  if (size > 0) {
```

```
      return num / size;
    }
    else {
      return -1;
    }
}
```

Or even:

```
public int getNumTeams (int num, int size)
{
   if (size <= 0) {
     return - 1;
   }
   else {
     return num / size;
   }
}
```

So this method is now avoiding the error and providing a value that indicates that the error has been spotted. It remains to do something with this value.

The *getNumTeams* method is called by the method that prints out the final information:

```
public void printInfo (int num, int size)
{
   System.out.println ("There are " + num
                       + " ducks available.");
   System.out.println ("This means "
                       + getNumTeams(num, size)
                       + " teams of " + size + ".");
   System.out.println ("This leaves "
                       + getRemainder(num, size)
                       + " substitutes.");
}
```

This method must now make use of the value returned by *getNumTeams*; it must examine this value and only display results if it is not an error. In fact it is only sensible to display results at all if the value is not an error so the program might as well exit immediately if there is a problem with the data supplied:

```
public void printInfo (int num, int size)
{
   if (getNumTeams (num, size) !=- 1) {
    System.out.println ("There are " + num
                        + " ducks available.");
    System.out.println ("This means "
                       + getNumTeams(num, size) + " teams of "
                       + size + ".");
    System.out.println ("This leaves "
                       + getRemainder(num, size)
                       + " substitutes.");
   }
   else {
    System.exit (0);
   }
}
```

A final touch is to make this final method display a message if the error case is encountered. This is a simple change to the `else` clause:

```
public void printInfo (int num, int size)
{
   if (getNumTeams (num, size) !=-1) {
    System.out.println ("There are " + num
                      + " ducks available.");
    System.out.println ("This means "
                      + getNumTeams(num, size)
                      + " teams of " + size + ".");
    System.out.println ("This leaves "
                      + getRemainder(num, size)
                      + " substitutes.");
   }
   else {
    System.out.println ("Error! Size of Teams is Invalid!");
   }
}
```

Example 2 – Bruce's sign revisited

Bruce is very pleased with his new electronic sign. He particularly enjoys changing the messages. He has taken to using the version of his program that uses a command line argument to set the message, and this works well.

One day he experiences a problem when he forgets to provide the command line argument and all sorts of unexpected things happen.

How can his program be altered to detect this error?

Bruce's sign program currently looks like this.

```
/* Sign3.java – Bruce's sign, taking a command line argument
                and displaying it as the message.

   Author : GPH
   Date   : 29th June 2003
*/

public class Sign3
{
   // Single Attribute – What to Display
   private String message;

   // Methods

   public Sign3 ()
   {
     message = "";
   }

   public void setMessage (String newMessage)
   {
     message = newMessage;
   }

   public void display ()
   {
     System.out.println (message);
   }

   // Main Method
   public static void main (String args[])
```

```
    {
      // Create an object for the sign
      Sign3 brucesSign = new Sign3 ();

      // Set the Message...
      brucesSign.setMessage (args[0]);

      //...and display it.
      brucesSign.display ();
    }
}
```

The problem here is that the new message to be displayed (stored in *`args[0]`*) is displayed whether or not a sensible value was provided. This is obviously something that can and should be tested for, but the question arises of what to test.

There are two obvious possibilities. Bruce could change his program to test whether or not the message had any characters in it. This is possible, but would only be appropriate if he never wanted to display an empty message. The alternative, probably better, approach is to find out how many command line arguments were supplied, and to complain if the number was not as expected.

The command line arguments are stored in *`args`*, which is a collection of strings. There will be more about collections of various sorts later on, but for the moment you just need to know that there is the public attribute, `length`, that returns the number of items in this particular type of collection. Bruce is expecting one argument, so the expected number is predictably enough 1.

The change now is to alter the program so that the message is only set if one command line argument has been provided. The simplest place to do this is in the main method:

```
if (args.length != 1) {
  // Create an object for the sign
  Sign3 brucesSign = new Sign3 ();

  // Set the Message...
  brucesSign.setMessage (args[0]);

  //...and display it.
  brucesSign.display ();
}
```

This means that the object for the sign is created only if there is a sensible message to display. And obviously the message is only set and displayed if the sign is created.

At the moment, of course, Bruce would be none the wiser if the message was missing. He might be somewhat alarmed by the lack of activity on his sign, so the program should also provide an error message if a problem has occurred. This is a situation where the `else` clause comes in handy:

```
if (args.length != 1) {
  // Create an object for the sign
    Sign3 brucesSign = new Sign3 ();

  // Set the Message...
  brucesSign.setMessage (args[0]);

  //...and display it.
  brucesSign.display ();
```

```
}
else {
  System.out.println ("Error! No Argument Supplied!");
}
```

This should keep Bruce properly informed. An alternative strategy would be to make use of Bruce's other maintenance program for this sign – the version that asked him to enter the message. It would be a simple matter to alter the program again so that it asked for a message if none were supplied as an argument.

The new code once again goes in the main method:

```
String msg = "";

// Check the argument, and get a message if needed
if (args.length != 1) {
    System.out.print("Enter message : ");
    msg = Console.readString();
}
else {
    msg = args[0];
}

// Create an object for the sign
Sign3 brucesSign = new Sign3 ();

// Set the Message...
brucesSign.setMessage (msg);

//...and display it.
brucesSign.display ();
```

14.1 For each of the following conditions write down a value of the variable for which the condition is true. Try to write down at least two values for which the condition is false; is this always possible?

```
aNumber == 1
aNumber >= 3.5
aChar != 'A'
aNumber < 100
aString.equals("elvis")
```

14.2 Examine the following program fragment carefully.

```
int aNumber;
double aDouble = 0.0;
String aString = "ducks";
```

```
aNumber = 10;
aDouble += 3.0;
```

After these statements have been executed what is the value of the following conditions?

```
aNumber == 2
aNumber <= 10
aString.equals("cilla")
aNumber == 10 && aDouble == 4.0
aNumber == 10 || aString.equals("cilla")
!(aNumber == 5)
```

14.3 Write a program that prompts the user to enter two floating-point numbers and enters the result of the first divided by the second. Your program should handle the special case where the second number is 0 and should display a suitable message informing the user that division by 0 is not possible.

14.4 Write a program that prompts the user to enter the current year and the year in which they were born and displays the user's age (or, more accurately, the age that the user will be this year). The program should check that the current year is greater than 2000 and that the user's year of birth is earlier than the current year. An informative error message distinguishing between the two possible errors should be displayed if the user does not provide valid data.

14.5 A class to store dates has the attributes:

```
private int day, month, year;
```

A method is required that will print the date in the format of, for example:

```
14th June 1967
23rd March 1982
10th October 1066
```

The method is of the form :

```
void printFormatted()
```

Implement the class and the method. Write a short program that tests your implementation. There is no need to check that the date supplied is valid, but you are welcome to try.

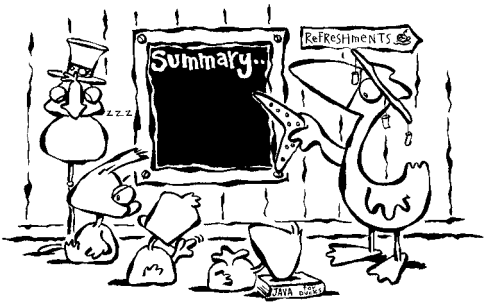
14.6 A simple class has been written to store the position of a duck on a pond. The attributes and methods are as follows:

```
public class Duck
{
   // Attributes
   private String name;
   private int x, y;
   // Constructor
   public Duck()
   {
   }
   // Selectors
   public String getName()
   {
   }
```

```
    public int getX()
    {
    }
    public int getY()
    {
    }
    // Mutators
    public void setName(String newName)
    {
    }
    public void setX(int newX)
    {
    }
    public void setY(int newY)
    {
    }
    public void moveTo(int newX, int newY)
    {
    }
    // Methods to move duck
    // Parameter is number of units
    void moveNorth(int d)
    {
    }
    void moveSouth(int d)
    {
    }
    void moveWest(int d)
    {
    }
    void moveEast(int d)
    {
    }
}
```

Implement this class and then use it in a simple program.

The program should prompt the user to enter the name and initial location of a duck and should then create an instance of the class. Both the x and y values in the initial location should be positive. Finally the user should be asked to enter a direction as a single character (N, S, W, or E – remember to validate it) and a number of units (which should also be positive). The program should end by displaying the position to which the duck has been moved.



Conditional statements allow programs to react and behave differently in response to values provided by users. The key idea behind a conditional statement is a condition, an assertion that can either be true or false. A condition is always true or false, although which it is can change as a program is run.

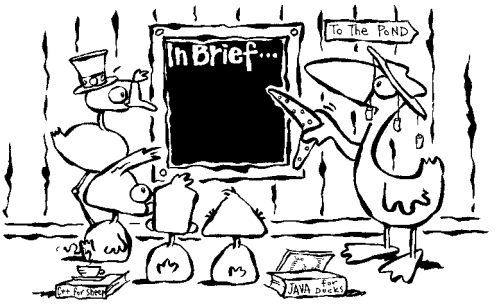
The values `true` and `false` are Boolean and can be stored in variables with the type `boolean`. Boolean values and conditions can be combined using the basic Boolean operators, "and", "or", and "not". Any number of conditions can be combined using these operators.

Conditions often include comparison operators. The simplest operator is "equals", represented in Java as == for basic data types. `String`s use the `equals` method for the same comparison. There is a range of other operators for comparing with values that are less than or greater than some known value.

When an unexpected value is detected in its input a program can be made to exit immediately using the `System.exit` command. If a program does end in this way it is always good practice to display an informative message to tell the user what has happened. A numeric value can also be passed to the computer via the `System.exit` command.

Of course it would be better if the program told the user of the error and gave them the chance to provide another value. A user is likely to get very frustrated at having to continually re-run a program. To allow this sort of control, statements must be executed over and over again; this leads neatly on to the topic of program loops.

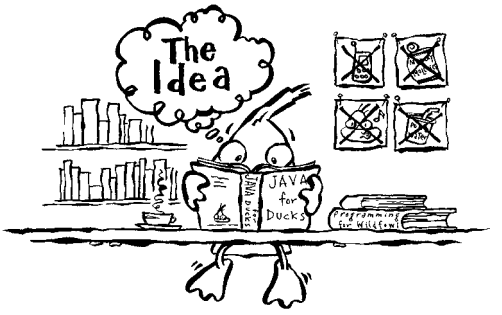

Chapter 15
Making Things Happen.
Again and again.
Laps to Go
04
LAPS
Elvis 3
Buddy 6



Conditional statements control the order in which the other statements in a program are executed. However, even with this level of control the fact remains that the overall flow of the program is still from the top to the bottom; there are usually several routes to the end of the program, but the flow is always in the same prevailing direction. Once a statement has been executed it can never be executed again. It is not possible to go backwards.

There is a second type of statement that can control the order of execution of a Java program. Program *loops* allow statements to be executed a certain number of times, or while some Boolean condition is true, or until some Boolean condition has become false. This chapter explains the different kinds of loops that are available and describes how to use loops in your classes and programs.

After reading this chapter you should be able to write methods and programs that include loops. You should understand the different kinds of loops available in Java and you should be able to decide which to use to solve a particular problem.



Let's illustrate this new idea with an example. The previous chapter explained how to validate the values that users supply to your programs. This is effective as far as it goes, but the dialogue that has resulted is hardly very friendly. The end result is that the users are simply told that they have made an error and the program then finishes; if the user wants to try again with another value (or set of values) the program has to be executed again and all the values have to be entered again. This would be especially annoying for a user who had entered several values correctly only to make a small error with a later value.

In these situations it would be far preferable to have a more complex dialogue where the user is still informed of the error but is then invited to re-enter an acceptable value. This dialogue might resemble:

```
Enter a number between 1 and 10: 20
Error: Number out of range. Try again: 5
Thank you. The number you entered was 5.
```

This is a much more friendly example of interaction than those that we have been able to achieve up to now.

To implement a dialogue such as this, one or more statements in a program must be executed a number of times. Since it is not possible to predict how many times the user will make a mistake, or even if they will make one at all, the statements concerned cannot be repeated in the program; rather the same statements must be executed until the user provides an acceptable value.

This is only one example of where a program *loop* might be used. There are many others situations where this is required, but there are in general only three types of loop:

- Part of a program is executed a given number of times. *Do this six times*.
- Part of a program is executed until some condition becomes true. *Do this until the user has entered an acceptable value*.
- Part of a program is executed while some condition is true. *Do this while the user continues to enter incorrect values*.

In each of these examples "part of a program" means one or more statements enclosed between the usual pair of braces.

The last two possibilities in this list may seem at first sight to be the same thing, but they are not. There is a subtle difference. A loop that is executed until some condition becomes true will always be executed at least once, while a statement that is executed as long as some condition is true may never be executed at all if the condition is initially false.

Some examples of loops are:

- Execute this statement seven times.
- Execute this statement, and carry on executing it until the user enters a negative value at the prompt.
- Execute this statement while the user continues to provide valid input values.

These three types of loop are implemented in Java as `for` loops, `do` … `while` loops and `while` loops, respectively. The names simply refer to the Java keywords that are used to express such a loop.

`for` loops

A `for` loop causes statements to be executed a certain number of times. This number might be a known literal value such as 10 or it might be held in a variable. The first case is used when it is known that the loop will always be executed the same number of times, while the second allows the number to be determined as the program is running.

The loop is controlled by a statement of the form:

for (<initialisation statement>; <continuation condition>; <change statement>)

The initialisation statement is executed the first time the loop is executed and the loop continues to execute as long as the continuation condition is true. Each time the end of the loop is reached the change statement is also executed.

This might all seem a little complicated, so an example illustrates:

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

The initialisation statement is often a declaration. In this example, when the loop is first executed, an integer variable `i` will be created and initialised to 0. The loop will execute as long as `i` is less than 10 (the continuation condition)

and at the end of each execution 1 is added to `i` (the change statement). The practical result of this is that in this case the loop will execute 10 times as `i` moves from 0 to 9 inclusive. It is usual to start loops that count in this way with an initial value of 0.[1]

The statements controlled by the loop are enclosed in the usual curly brackets:

```
for (<initialisation statement>; <continuation condition>; <change statement>) {
  // statements
}
```

As usual the statements are indented to emphasise the fact that they are inside the loop. Any Java statements may appear inside the loop, including conditional statements and other loops.

The variable declared in the initialisation is called the loop's *control variable*. It can be of any data type but it is most usually an integer. It does not necessarily have to be declared inside the first line of the loop, but it most usually is.

The control variable is often used inside the loop. For example, to print the first 10 integers (starting at 0, of course) the loop would be:

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

A slightly more useful loop prints out the 12 times table:

```
for (int i = 1; i < 13; i ++) {
  System.out.print (i);
  System.out.print (" x 12 = ");
  System.out.println (i * 12);
}
```

This loop generates a lot of output for a small amount of code. This is the power of loops. This program might also remind you of Buddy's "Times Table" program from Chapter 9; we'll look at how that program can be rewritten with a loop later on.

While the control variable can be used in any statement inside the loop there are two important rules:

1. The control variable should never be altered inside the loop. It should be altered only when an execution of the statements inside the loop has ended.
2. If the control variable is declared in the header of the loop it cannot be used once the loop has terminated.

If it is necessary to use the value of the control variable after the loop has finished it can be declared before the loop:

```
int i;
for (i = 0; i < 10; i ++) {
  System.out.println (i);
}
// i can be used here
```

1 Remember that computers start to count at 0, not 1!

Otherwise, attempting to use the control variable after the loop has finished will result in an error:

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

// using i here is an error
```

By far the most common use of a `for` loop is just to execute a set of statements a given number of times; the easiest way of doing this is always to count up to this number. This gives a loop of a general form that will cause statements to be executed *n* times:

```
for (int i = 0; i < n; i ++ ) {
    // statements executed n times
}
```

Other less common possibilities include printing out just even numbers:

```
for (int e = 0; e < 10; e += 2) {
  System.out.println (e);
}
```

or counting backwards from a number:

```
for (int b = 10; b >= 0; b --) {
  System.out.println (b);
}
```

You might want to type these loops into small programs to see what they do.

`for` loops are called *determinate* loops because the number of times that the loop will be executed can always be predicted (hence determined) before the loop begins. This does not mean that the number of times a certain loop executes is always the same; the number is often controlled by, or based on, the values of other variables in the program.

For example the loop to print the 12 times table:

```
for (int i = 1; i <= 12; i ++ ) {
  System.out.print (i);
  System.out.print(" x 12 = ");
  System.out.println(i * 12);
}
```

might be more useful if the user could choose how many terms of the table to see:

```
int terms;

/* get the number of terms - in reality this number
   would also be validated with a conditional statement */

System.out.print ("Enter the number of terms: ");
terms = Console.readInt ();

// display the table

for (int i = 0; i < terms; i ++ ) {
  System.out.print (i);
  System.out.print (" x 12 = ");
  System.out.println (i * 12);
}
```

It is not possible to forecast how many times this loop will be executed before the program is run, but the number will always be known immediately before the loop starts.

The loop could easily be extended to allow the user to pick the first value in addition to the number of terms:

```
int start;
int terms;

System.out.print ("Enter the first value: ");
start = Console.readInt ();
System.out.print ("Enter the number of terms: ");
terms = Console.readInt ();

/* these values should be validated here, and the
   loop executed only if they are sensible! */

for (int i = start; i <= start + terms; i ++ ) {
  System.out.print (i);
  System.out.print (" * 12 = ");
  System.out.println (i * 12);
}
```

Finally, some specialised users might require only alternate terms in the table:

```
for (int i = start; i <= start + terms; i += 2) {
  System.out.print (i);
  System.out.print (" * 12 = ");
  System.out.println (i * 12);
}
```

`for` loops are used when the number of times that a loop will be executed can be determined for certain before the loop is executed for the first time. If this number cannot be determined then the alternative, an *indeterminate loop*, must be used instead.

`while` loops

A `while` loop is indeterminate; this means that it causes statements to be executed while some Boolean condition is true. The condition can be anything that produces a Boolean value, much like the condition of an `if` statement. A `while` loop is started with a line of the form:

(while (<condition>)

and the program statement or statements affected by the loop appear immediately below enclosed with the familiar braces:

(while (<condition>) {
* // statements*
}

The statements are of course indented to make it clear that they are inside the loop. As with a `for` loop, any valid statements may appear inside the loop, including other loops.

The statements will continue to be executed as long as the condition in the loop is true. When the final statement in the sequence is reached control returns to the first, and so on. This means that:

- the condition must at some point become false or the loop will never terminate;

- if the condition is not true the first time the loop is to be executed the statements will never be executed at all;
- the programmer must be confident of the value of the condition every time the loop is reached, including the first.

`while` loops are often used to make sure that the values entered by users are as expected. If the user enters an invalid value a `while` loop can be used to provide an error message and repeat the prompt until an acceptable value is provided:

```
int aNumber;

System.out.print ("Enter a number between 1 and 10: ");
aNumber = Console.readInt ();

while (aNumber < 1 || aNumber > 10) {
  System.out.println ("Error: Number out of range");
  System.out.print ("Enter a number between 1 and 10: ");
  aNumber = Console.readInt ();
}
```

The condition

```
aNumber < 1 || aNumber > 10
```

defines the state when the number entered is out of the required range and therefore invalid. The loop is executed only if the first attempt to enter the number is invalid and it will repeat until a valid number is entered:

```
Enter a number between 1 and 10: 15
Error: Number out of range
Enter a number between 1 and 10: 20
Error: Number out of range
Enter a number between 1 and 10: 7
```

This produces a much neater dialogue between the user and the program. If the user makes a mistake they have the chance to correct the error immediately rather than having to run the whole program again.

Care must be taken that the value of the condition changes at some point in the loop. For example, this loop will never terminate:

```
int aNumber = 10;
while (aNumber >= 10) {
  System.out.print("The number is " + aNumber);
  aNumber ++ ;
}
```

Here the value of *aNumber* will keep increasing and the value of the Boolean expression controlling the loop (that *aNumber* must be greater than or equal to 10) will always be true. The loop is an *infinite loop*. In some very unusual situations an infinite loop is what is required, but such situations are very rare. If you find yourself with a program that never seems to end, it's always a good idea to check for infinite loops.

Sometimes the statements inside a `while` loop will not be executed at all. If the condition is initially false, then control skips past the loop and the statements are never executed. There are times, however, when it is clear beforehand that the statements inside the loop will always need to be executed at least once. In this case there is an alternative form of indeterminate loop that can handle this situation more elegantly.

`do…while` loops

`do…while` loops are a specialised form of `while` loop. In fact any `do … while` loop can always be written as a `while` loop.[2] The only advantage of this type of loop is that the resulting code can be neater. The relationship between these two types of indeterminate loop is not unlike that between the `if` statement and the `switch` statement – it is possible to get by with only one, but using both makes for neater programs.

The statements inside a `do … while` loop are always executed at least once. The loop is once again controlled by a condition but this time the condition is tested only after the statements inside the loop have been executed; if the condition is true then the loop is executed once again. The format of the loop makes the location of the test clear:

```
do {
   // statements
} while (<condition>);
```

As is now very familiar, the statements inside the loop are enclosed with braces and are indented. Also as usual, the statements may include any valid Java statements, including other loops.

`do … while` loops may also be used to validate values entered by a user. Their use for this is reasonable since the user is always going to have to enter the value at least once and therefore the statements in the loop will always be executed at least once.

A `do … while` loop can be used to validate user input:

```
do {
  System.out.print ("Enter a number between 1 and 10: ");
  aNumber = Console.readInt ();
  if (aNumber < 1 || aNumber > 10) {
     System.out.println("Error: Number out of range");
  }
} while (aNumber < 1 || aNumber > 10);
```

This is obviously very similar to the example using a `while` loop and there is little to choose between the two. Both include some duplication of code; the `while` loop example has two `System.out.print` statements and this version has the condition tested twice. The two are equivalent; the choice of which to use in a particular application would come down to the usual matter of style and taste.[3]

This observation applies to most cases where an indeterminate loop is used. If it is obvious that the loop will always be executed at least once then a `do … while` loop is the correct choice, but many programs could equally be written with either form of loop.

A final warning about == and =

Finally, a word about a very common error. Very often an indeterminate loop executes as long as some variable has a particular value. This is most commonly

---

2 And indeed, vice versa.

3 It's interesting that I tend to prefer the `do … while` version but everyone I work with seems to use the `while` version. It really is just down to style and preference – programming style can be a very personal thing.

the value of some Boolean variable:

```
boolean finished = false;
while (finished == false) {
  // statements
}
```

or in general:

```
while (<variable> == <value>) {
  // statements
}
```

A very common error and careless error is to write loops of this form as:

```
while (<variable> = <value>) {
  // statements
}
```

or

```
do {
  // statements
} while (<variable> = <value>);
```

Neither of these loops will behave as expected. If the variable is any type other than boolean the program will fail to compile. The compiler spots that = has been used instead of == and points this out. On the other hand, if a `boolean` has been used the program will compile correctly, but will then fail to work as expected when run.[4]

The problem is that the = (is assigned) operator has been used instead of the == (equals) operator. If you find that one of your programs has a loop that is not behaving as expected this is always something to check. The correct loops are of course:

```
while (<variable> == <value>) {
  // statements
}
```

and

```
do {
  // statements
} while (<variable> == <value>);
```

The same error can, of course, also crop up in any condition, and particularly in conditional statements. The trick to avoid this is to get into the habit of never writing:

```
finished == true
```

4 The reason for this distinction is very subtle indeed. The assignment operation actually returns the value assigned so a statement like "`finished = true;`" returns true, which is the correct Boolean type to use in this place. A statement `someNumber =0;` returns an integer, which is not the correct type, and so the compiler spots the error. Java is actually very good at spotting these errors; C or C++ would allow these errors with any type.

but always using:

```
finished
```

which amounts to exactly the same thing (and saves typing). The same applies to:

```
finished == false
```

and:

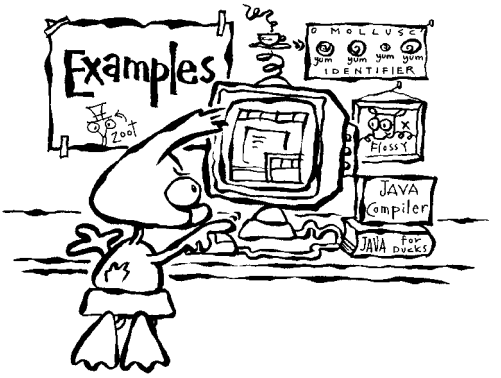
```
!finished
```

The shorter form is probably always to be preferred.

Flow of control

Armed with loops and conditional statements you can now write programs that have a complex *flow of control*. While your programs still execute in a general sense from top to bottom you now have much more control over precisely what is going on.

This is a very basic skill of programming, so it's time to get some practice in with some examples and exercises.



Example 1 – Buddy's tables revisited

Buddy is very happy with his "Times Table" program, but the contents of his main method are starting to worry him. There seems to be a lot of repetition, and he is wondering if a loop could make his program neater.

The question arises of whether he is correct …

Buddy's program calculated the first six terms of a times table; the number of the table required was provided as a command line argument. The body of the main method looked like this:

```
TimesTable tt = new TimesTable ();
int num = Integer.parseInt (args[0]);

tt.printLine (1, num);
tt.printLine (2, num);
tt.printLine (3, num);
tt.printLine (4, num);
```

```
tt.printLine (5, num);
tt.printLine (6, num);
```

This is indeed an obvious case where a loop can be used. The first parameter of the *printLine* method is clearly varying from 1 to 6, and this can easily be replaced with the control variable of a suitable loop. The loop is determinate since we know that it will always execute exactly 6 times, so this is a case where a `for` loop is appropriate.

The 6 statements printing the tables are simply replaced with:

```
for (int i = 1; i <= 6; i ++ ) {
  tt.printLine (i, num);
}
```

This solution is shorter and neater and, as we will see in a moment, is much easier to extend.

Example 2 – Elvis's harder tables

Elvis is impressed with Buddy's program but wants a program that will do more. He wants a program that will print 13 terms of each table (starting at "0 times" and finishing with "12 times". He also wants to print out complete versions of the first 13 tables so that he can get some revision in.

He decides to use Buddy's program as the basis for his own. What does he need to change?

Two loops are actually needed in this program. One will count the number of tables and one will count through the terms of the tables. Since both loops are always required to execute exactly 13 times this is a clear case where a `for` loop should be used for both.

There are also two changes needed to Buddy's program. The first will make the program count up to "12 times" rather than the current 6, and the second will make the program repeat for a number of tables. When more than one change is to be made to a program it is always a good idea to make changes to a program one at a time. This means that any error that occurs will always have been caused by the most recent change; it should therefore be easier to spot, isolate, and correct.

The first stage is therefore to alter Buddy's program so that 12 terms are printed for each table. With the changes just made, this is trivial as the only change needed is in the loop which becomes:

```
for (int i = 0; i <= 12; i ++ ) {
  tt.printLine (i, num);
}
```

This could equally well be written like this:

```
for (int i = 0; i < 13; i ++ ) {
  tt.printLine (i, num);
}
```

which is the style that many Java programmers (particularly those brought up on C or C + +) might prefer; it's just the usual matter of style and preference.

Now we come on to add Elvis's requirement for many tables. Once again we can forecast in advance how many tables will be printed; this is obviously another determinate loop. Another `for` loop is needed.

The code looks a little complicated, but it can be built up gradually. The first step might be to include some neat headings for each table, so this loop would print out some headings for each of the tables:

```
for (int table = 0; table < 13; table ++ ) {
  System.out.print("The ");
  System.out.print(table);
  System.out.println(" Times Table";
}
```

It's worth noting in passing here that there is no need for three separate `System.out.print` lines here; it's just done like this to keep the lines on the page! The *< 13* here is another matter of preference. Since the loop is intended to count up to 12 you might prefer:

```
for (int table = 0; table <= 12; table ++ )
```

which amounts to exactly the same thing. In this case the control variable has been given a meaningful name; this becomes a very good idea when there is more than one loop in a program as there is here.

We already have a loop to print out a particular times table. In the current program the number of the table required is stored in the variable *num* which is obtained via a command line argument:

```
for (int i = 0; i < 13; i ++ ) {
  tt.printLine (i, num);
}
```

The trick in writing Elvis's program to print out a collection of tables is to put the second loop inside the first. As a first step to doing this, this pair of loops will print out the required times table 13 times:

```
for (int table = 0; table < 13; table ++ ) {
  System.out.print("The ");
  System.out.print(table);
  System.out.println(" Times Table");
  for (int i = 0; i < 13; i ++ ) {
     tt.printLine (i, num);
  }
}
```

Again the indentation clearly shows that the inner loop is inside the other. The final fix is to make the number of the table that is to be generated change for each execution of the outer loop. This is easily done since the number of the required table is actually the control variable of the outer loop, *table*. The inner loop will use this value rather than the command line argument. The command line argument is now in fact not used at all.

This gives the final loops that would be used in a complete program:

```
for (int table = 0; table < 13; table ++ ) {
  System.out.print("The ");
  System.out.print(table);
  System.out.println(" Times Table");
for (int i = 0; i < 13; i + + ) {
```

```
      tt.printLine (i, table);
    }
  }
```

Example 3 – Elvis's calculator

Elvis has many demands on his time. He helps with organising the ducks into their cricket teams and also has to spend a lot of time keeping track of his pocket money. He has decided that what he needs is a simple calculator to help in these important tasks. Unfortunately his pocket money does not stretch to buying a calculator, so he has decided to write one in Java.

He would like to be able to add, subtract, multiply, and divide. Since money is involved he needs to be able to cope with floating-point numbers. The calculator should prompt him for two numbers and the required operation and then display the result. Elvis would prefer not to have to keep rerunning the program for each calculation.

The code to carry out the arithmetic in this example is quite simple. For example, to carry out an addition calculation:

```
double firstNumber;
double secondNumber;

System.out.print ("Enter the first number: ");
firstNumber = Console.readDouble ();

System.out.print ("Enter the second number: ");
secondNumber = Console.readDouble ();

System.out.print (firstNumber);
System.out.print (" + ");
System.out.print (secondNumber);
System.out.print (" = ");
System.out.println (firstNumber + secondNumber);
```

This is fine for a program that is just going to handle addition, but a more general approach is needed for this program. At the start of the program it's quite impossible to forecast whether the calculation required will be an addition, subtraction, multiplication, or division. A conditional statement is needed; `switch` is the obvious choice as there are four distinct possible operations that might be carried out. The user must be prompted to enter the symbol representing the required calculation and a `switch` statement will choose the appropriate action and carry it out. A new variable to store the result will also be needed.

The code for a basic calculation now becomes:

```
double firstNumber;
double secondNumber;
double result;
char operation;

System.out.print ("Enter the first number: ");
firstNumber = Console.readDouble ();

System.out.print ("Enter the second number: ");
secondNumber = Console.readDouble ();
System.out.print ("Enter the operation (+, -, *, /): ");
operation = Console.readChar ();
```

```
switch (operation) {
   case '+': result = firstNumber + secondNumber;
             break;
   case '-': result = firstNumber - secondNumber;
             break;
   case '*' : result = firstNumber * secondNumber;
             break;
   case '/' : result = firstNumber / secondNumber;
             break;
}

System.out.print (firstNumber);
System.out.print (" ");
System.out.print (operation);
System.out.print (" ");
System.out.print (secondNumber);
System.out.print (" = ");
System.out.println (result);
```

As it stands this `switch` statement will fail and incorrect results will be produced if an invalid character (one that does not represent an operation) is entered. This means that the operation character (stored in the variable called *`operation`*) must be validated as it is entered. This involves using a suitable loop to warn the user of the error and to ask them to re-enter. The code to do this follows a familiar pattern:[5]

```
System.out.print ("Enter the operation (+, -, *, /): ");
operation = Console.readChar ();

while (operation != '+' && operation != '-'
       && operation != '*' && operation != '/') {
  System.out.println ("Error: Invalid Operation");
  System.out.print ("Enter the operation: ");
  operation = Console.readChar ();
}
```

This ensures that the `switch` statement has a valid character to process.

Elvis wants the program to execute over and over again so obviously another loop, controlling the whole program, is required. This loop will always execute at least once and so this time a `do … while` loop is the appropriate choice. The loop is controlled by a Boolean variable *`finished`*, which is initially `false` and is set to `true` only when the user indicates that they do not want to run the program again.

A prompt is needed to ask the user if they want to run the program again and a simple dialogue gets the user's response and stores it in a `char` variable *`answer`*:

```
System.out.print ("Run Again? (y/n): ");
answer = Console.readChar ();
```

and a conditional statement sets the *`finished`* variable if appropriate:

```
if (answer == 'y') {
```

---

5 You have now seen a lot of familiar patterns as you have read these chapters. This is a good thing. One of the "tricks" of programming is spotting these patterns in a problem, and realising tht a solution to some other problem can be adapted to the new problem. This is why it's a good plan to make sure that you never throw away any programs that you write.

```
    finished = true;
  }
```

or, equally:

```
  finished = (answer == 'y');
```

Slightly better programming style (and dialogue design) is to consider the fact that the user might enter either an upper case or lower case letter and to use:

```
  if (answer == 'y' || answer == 'Y')
```

as the conditional statement.

If a program is to present the user with a good dialogue it should also enforce the entry of *y* or *n* as an answer to this prompt. This extra feature has been added to the complete program that follows. This program also handles the case where the operation is division and the second number is 0 it is necessary to avoid an error caused by a division by this 0.

Here is the final program. While the statements that carry out the steps to solve the program are very few in number, there are many more that control the program flow, produce the dialogue, and deal with possible error cases. It is quite usual for a great deal of any program to be carrying out tasks such as these.

Take a close look at this program and make sure that you understand what's going on.

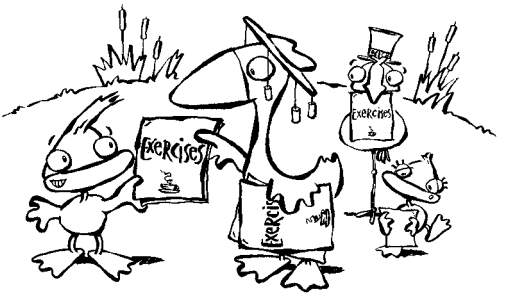
```
/* Calculator.java - A simple calculator for all Elvis's needs.

   Author    : AMJ
   Date      : 20th December 2002
   Platform  : Java, Linux
               Red Hat 7.3, JDK 1.4
*/

import htpuj.*;

public class Calculator
{
  private double firstNumber, secondNumber;
  private char operation;

  public Calculator () {}

  // Receive user input
  public void getInput ()
  {

    System.out.print ("Enter the first number: ");
    firstNumber = Console.readDouble ();

    System.out.print ("Enter the second number: ");
    secondNumber = Console.readDouble ();

    System.out.print ("Enter the operation ( + , - , *, /): ");
    operation = Console.readChar ();

    while (operation != ' + ' && operation != ' - '
         && operation != '*' && operation != '/') {
      System.out.println ("Error: Invalid operation");
      System.out.print ("Enter the operation ( + , - , *, /): ");
      operation = Console.readChar ();
  }
}
```

```
// Perform calculation
public void calculate ()
{
    if (operation == '/' && secondNumber == 0.0) {
    System.out.println ("Error: attempted to divide by 0");
   }
   else {
     double result = 0.0;
   switch (operation) {
     case ' + ': result = firstNumber + secondNumber;
                 break;

     case ' - ': result = firstNumber - secondNumber;
                 break;

     case ' * ': result = firstNumber * secondNumber;
                 break;

     case ' / ': result = firstNumber / secondNumber;
                 break;
   }
System.out.print (firstNumber + "  " + operation + "  "
                  + secondNumber + " = " + result + "\n");
  }
}

// Loop repeatedly while user wishes to perform more calculations
public void run ()
{
  char answer;
  boolean finished = false;

  while (!finished) {
    getInput ();
    calculate ();

    System.out.print ("Run again (y/n)? : ");
    answer = Console.readChar ();

  do {
      if (answer == 'n' || answer == 'N') {
        finished = true;
      }

      else if (answer != 'Y' && answer != 'y') {
        System.out.println ("Error: Please enter \'y\' or \'n\'");
        System.out.print ("Run again (y/n)? : ");
        answer = Console.readChar ();
      }
    } while (answer != 'y' && answer != 'Y'
         && answer != 'n' && answer != 'N');
   }  // while (!finished)
  }

  public static void main(String args[])
  {
    Calculator calc = new Calculator ();

    calc.run();
  }
}
```

This is the longest program that you have seen so far and there are a few things to notice:

- The indentation shows at all times which statements are controlled by which control statements.
- There are often loops within loops and conditional statements within other conditional statements and loops. The indentation helps to keep track of which statements are controlled by which other statements.
- There are sometimes many statements enclosed between one set of braces. Where the closing brace is a long way away from the corresponding opening brace, a brief comment has been added to emphasise what the brace is ending.
- The calculator itself has been modeled as an object, so the `main` method is only two lines long! The calculator is first created, and then told to run. This is quite a common form for Java programs.

This is a lengthy program containing most of the Java that you have met so far. Take some time to go through it and make sure you understand what's going on.



15.1 Write a program that will print a triangle made up of stars, as follows:

```
*
**
***
****
*****
```

The number of stars in the final line should be provided by the user in answer to a suitable prompt. The program should ensure that the provided value is at least one and should allow the user to re-enter the value if an error is made.

Your program should use a simple class, *`Triangle`*, that has one integer attribute to represent the height and a simple method to display the triangle.

15.2 Write a similar program (using a similar class) that prints a square made up of stars:

```
********
********
********
********
********
```

```
********
********
********
```

As before, the number of stars making up each side of the square should be provided by the user. This number should be positive.

15.3 Using your two programs as a starting point write a program that presents the user with a small menu:

```
Shape Menu
~~~~~~~~~~

1. Draw a Triangle
2. Draw a Square
Q. Quit
```

The user should select an option from the menu and should be prompted to enter the required size of the shape. The program should continue to execute until the user chooses the Quit option. If the option chosen is not valid then an error message should be displayed and the menu should be shown again.

Use a class called *`ShapeDraw`* or similar.

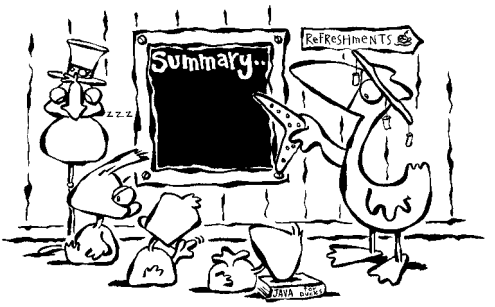
15.4 Add a further shape (class) to your program. A rectangle (use a class called *`Rectangle`*) requires a height and a width to be entered:

```
******
******
******
```

15.5 Finally, change your program so that the user can choose the character that is used to draw the shapes. This should be done by adding an option to the menu:

```
C. Change Character
```

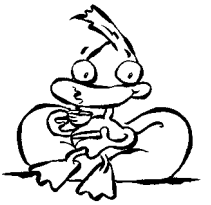
The first time the program is executed the character should still be *. The character used must be one of $, %, * or ^. The simplest way to do this is probably to add an attribute to each class.

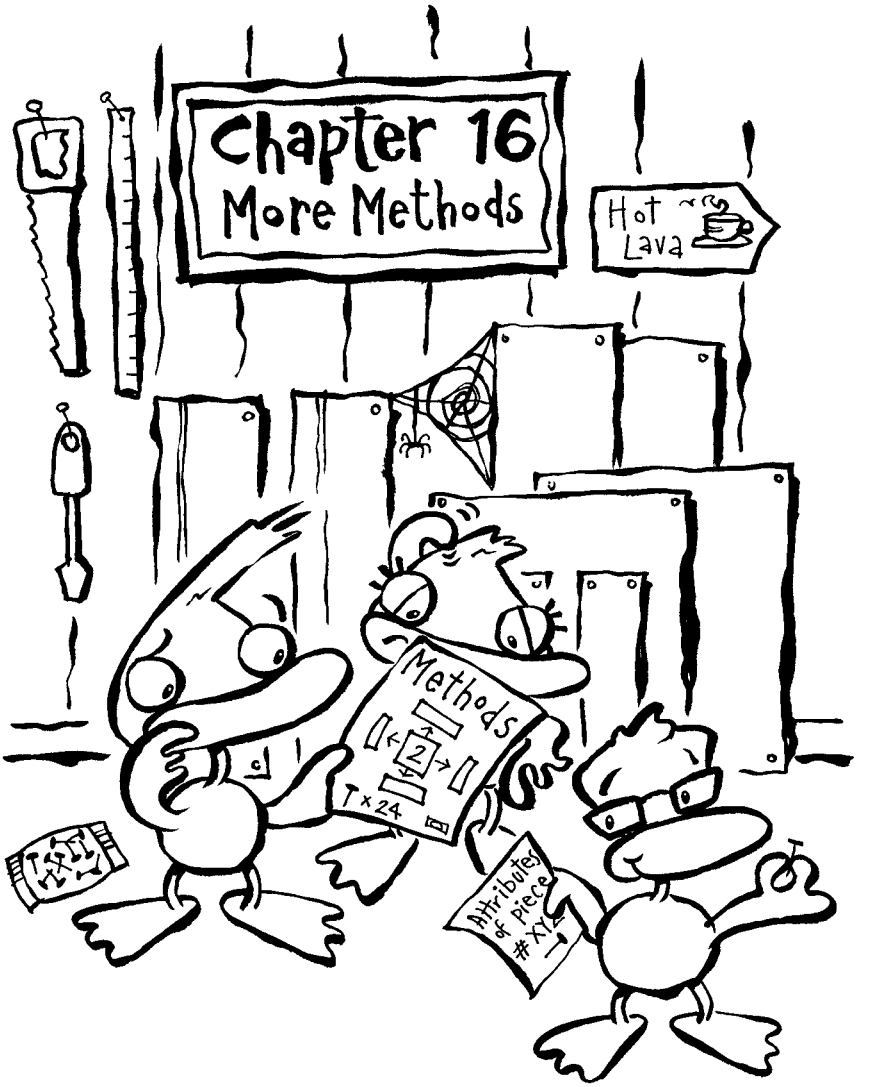


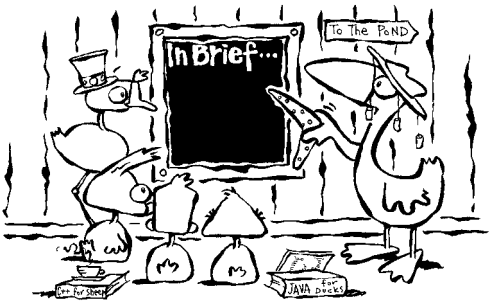
Program loops allow a great deal of flexibility in managing the way that control flows in a program. They allow for the development of much more sophisticated dialogues and mean that programs may continue to execute until the user has finished with them. Together with conditional statements loops give a programmer full control over the order in which program statements are executed.

There are two general types of loop. The first, called determinate loops, is used when the number of times that a loop will execute is known before it started. The alternative, called indeterminate loops, gives the extra flexibility of executing a loop under the control of some Boolean condition. There are two forms of indeterminate loop, one providing for the special case where it is known that a loop will be executed at least once.

Loops and other control statements can be used in any program and in methods. It is now time to look at some more examples of this and to fill in some final details of methods.



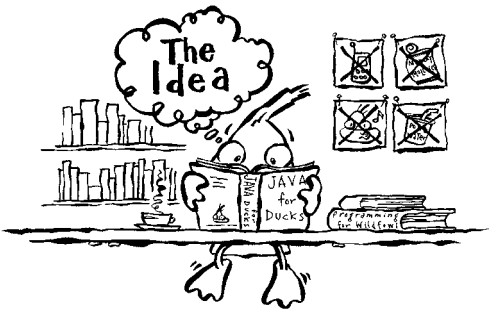

Chapter 16
More Methods
Hot
Lava
Methods
2
T x 24
Attributes of piece
#XY4



The techniques for controlling the flow of execution of statements that you have seen and practised in the previous two chapters can also be used in methods. Now you should be able to use the techniques from the last two chapters to write methods that carry out more complicated processing.

This chapter explains some other final features of methods. There are actually two sorts of parameters that can be used with a method – parameters can be used as "call-by-reference" or "call-by-value" – and this chapter will explain the difference. It will also look at how methods themselves can sometimes be used as parameters to other methods. Finally, there will be a quick look at *static methods*, methods that can be called without an object.

After reading this chapter you should be able to implement most if not all of the methods needed for your classes. You should be able to write methods that make use of conditional statements and loops, and also other methods! Of course you should also be able to use these methods from your Java programs.



There is not much new Java in this chapter, but some of it can seem a little complicated. Before we start, let's recap for one last time some of the details of methods that we have covered so far. A method in Java is defined by a line of the form:

public <return type> <method name> (<parameter types>)

Our convention is that methods are public (althoug that is only a convention). The description itself starts with the type of value that is returned by the method. This is followed by the name of the method, which is in turn followed by the types and identifiers of any parameters, in brackets and separated by commas. This corresponds to the first line of a method declaration.

Such a description is sometimes called a *prototype*. The description has in fact much the same function as a prototype in more familiar engineering

applications where, for example, a model of a car is made before the car itself. A method's prototype shows a potential user how to call the method, and usually gives a good indication as to its behaviour. Armed with the information gleaned from the prototype a programmer can write a program that will make use of the method without having any knowledge at all of how the method achieves its result. Also, a compiler can compile the part of the program that uses the method with no knowledge of how the method works; the code that implements the method itself can be linked in later.

Many methods determine and return exactly one value of the type given as the first part of the prototype as the return type; if a method does not return a useful value at all it is defined to return `void` and is called a void method. Methods sometimes require values to process in order to produce their result; these are the method's parameters, the types and identifiers of which are listed in brackets after the name of the method. These brackets are always included to make it clear that we are talking about a method, but left empty if there are no parameters.
For example:

```
public double distanceFrom (int x, int y)
```

defines a public method called *distanceFrom* that returns a single floating-point value and requires two integer parameters. Let's look in a little more detail at precisely how these parameters are passed into the method.

Call-by-value and call-by-reference

Many of the methods we have seen so far have processed one or more parameters. Specifically they have processed the values of one or more parameters. The *distanceFrom* method, for example, could be called like this:

```
Duck elvis = new Duck ();
double distance;
distance = elvis.distanceFrom (5, 3);
```

where the parameters are the two literal values 5 and 3. It could also be called like this:

```
Duck elvis = new Duck ();
int x = 5;
int y = 3;
double distance;
distance = elvis.distanceFrom (x, y);
```

This would have much the same effect. The point here is that it is the values of the variables *x* and *y* that are used by the method. Using *x* and *y* in this way is exactly the same as passing the literal values 5 and 3.

We would probably be very surprised if calling this method changed either value, and we would certainly expect the values of `x` and `y` to be unchanged immediately after the call to the method. But sometimes an effect like this is what is required and when this is required a subtly different kind of parameter is needed.

This section sounds daunting, and to be fair it can be quite a tricky concept to understand at first, but it can help you in writing elegant programs. There are two ways to pass values to methods; these are call-by-value and

call-by-reference. The names refer to the different ways the parameters are treated within the calling method.

We'll look at call-by-value first, as this is what we have been dealing with so far, and is indeed what has just been described. As another example, let's define a potentially useful method that might be found in some class used in a simple calculator program:

```
public int square (int num)
```

This method would take its argument, square its value, and return the result. So we could create a value to square, and print the square:

```
int number = 8;
System.out.println ("The square of " + number + " is "
                    + square(number) );
```

Now, you might think that the value stored in the variable number is now 64 since that is the value that will be displayed. However, this is not the case:

```
int number = 8;
System.out.println ("The square of " + number + " is "
                    + square(number) );
System.out.println ("The value of number is now " + number);
```

The output from this code would be:

```
The square of 8 is 64.
The value of number is now 8.
```

This might seem surprising, but it can be easily explained (and if you thought the value would still be 8 all along, then well done!).

It works like this. Whenever a variable of a primitive type (integers, doubles, and so on) is passed as a parameter of a method, the value of this variable is copied, and the method works with this copy. So in the example above, the *square* method works with a temporary copy of the value stored in *number*. Inside the method, of course, this value might have any identifier:

```
public int square (int num)
{
  num *= 2;
  return num;
}
```

or just:

```
public int square (int num)
{
  return num *= 2;
}
```

When the method terminates, the value of *number* in the calling method is unchanged. The value used in the method (here we called it *num*) is lost forever. The two values have in fact always been separate and have occupied different parts of the computer's memory.

Things are different when more complex values are passed to methods. When objects are passed as parameters to methods, the mechanism is subtly different, and it is important that you understand how it works and its effects on your programs so that you don't have problems later on.

Consider the following, admittedly rather unusual, method:

```
public static void renameDuck (Duck d, String s)
{
  d.setName(s);
}
```

This method takes two parameters, a *Duck* object and a `String` (also an object, of course). It sets the name of the *Duck* object to the value of the `String` parameter, and does this using the usual mutator method. Notice the extra word in the method definition – `static`. In essence this means we do not need to create an object in order to call the method, but we will discuss static methods more fully a little later on.

Now, let's create a *Duck* object to try it out on. We'll assume that the class has a *getName* selector method that works in the obvious way.

```
Duck myDuck = new Duck ();
myDuck.setName ("Elvis");
System.out.println ("My duck is called " + myDuck.getName() );
renameDuck (myDuck, "Buddy");
System.out.println ("My duck is now called " + myDuck.getName() );
```

The output from this snippet would be:

```
My duck is called Elvis
My duck is now called Buddy
```

Notice that while we have not explicitly called the *setName* method on our *Duck* object, the name attribute of the *Duck* object has changed. This can mean only one thing; the object manipulated within the *renameDuck* method is not a copy of the object passed as an argument, but the object itself. In fact, this is the way objects are passed to methods in Java, and as you have seen, this is fundamentally different to the way primitive types are passed.

This is a powerful mechanism, but one that can cause unexplained errors if not understood properly. You should probably reread this section until you are sure you understand something about this mechanism. The example programs should also help.

Using methods as parameters

We have not introduced any new Java in this chapter, but we have highlighted some important concepts. Many new programmers find reference parameters a difficult concept to grasp. It is important that you get some practice and that you think about the difference between the two types of parameters. This advice is especially timely when we come to look at collections in the next chapter.

While on the subject of methods, it is worth mentioning that you can actually pass a method call as a parameter of another method. This may seem strange, but it is perfectly reasonable when you consider why. Some methods return useful values – `int`, `String`, `char`, and so on – and some methods take such values as parameters. Often a method will return a value that would be useful for another method to take as a parameter. Rather than having to store the return value, and then pass this stored value to another method, we are allowed to simply put one method call inside the other. We have actually been

doing this already with one of the *printNeatly* methods earlier on:

```
public void printNeatly ()
{
  System.out.println (toString () );
}
```

`System.out.println` is a method that prints to the console, and `toString` is a method that returns a `String` object containing information about this particular duck. This is neater than the equivalent:

```
public void printNeatly ()
{
  String tmp = toString ();
  System.out.println (tmp);
}
```

As another example, suppose that we wanted to use the now familiar *distanceFrom* method to display how far a Duck was from some hazardous object. We could write:

```
Duck elvis = new Duck ();
double distance;

distance = elvis.distanceFrom(5, 3);
System.out.println ("Elvis is " + distance + " away!");
```

This is fine, but it is neater to do away with the temporary variable distance altogether and to write:

```
Duck elvis = new Duck ();
System.out.println ("Elvis is " + elvis.distanceFrom (5, 3)
                    + " away!");
```

While this form of method call can make your code neater, it is not without potential pitfalls. It is vital that you carefully check that the return type of the method to be called as a parameter is the same as the type that the calling method requires as a parameter. So a method that takes an `int` as a parameter can only call a method that returns an `int`, and so on.

Static methods

And finally, another detail of methods. Once again there is nothing new here because we've been using static methods since the very first Java program; the `main` method is an example:

```
public static void main (String args[])
```

The `main` method is special in that it is called without an object being created. It is in fact called automatically when the JVM is asked to execute a program.

It is possible to define other static methods. These are simply methods defined within a class that do not need an object of that class in order to be called. Some Java programmers will, with some justification, frown on the use of static methods, and we wouldn't recommend that you start using them very much, but it would be good to recognise one if you see one!

The declaration of a static method is in the same format as the definition of other methods except that the word `static` is added:

public static <return type> <method name> (<parameter types>)

As an example, suppose that we wanted to write a "method" that might be used to display some menu. The method would need to be defined in a class, but as there is no need to associate this method with an object, so it could be static. Assuming the class was called *Menu*, and the method declaration was:

```
public static void displayMenu ()
{
  System.out.println ("Options");
  System.out.println ("1. Option 1");
  System.out.println ("2. Option 2");
}
```

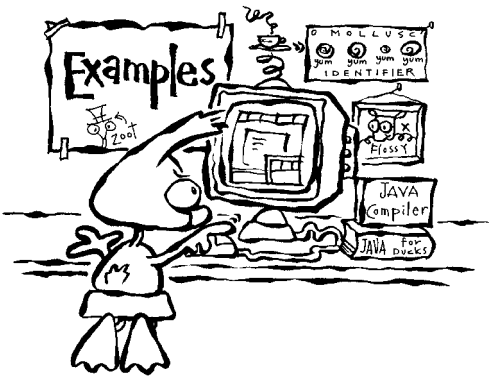
This method could be called with just:

```
Menu.displayMenu ();
```

Using methods like this can make programs easier to understand, and it can be useful especially if there is a need to duplicate blocks of code in a program. The message is to use methods like this occassionally and with care!

Methods

There have been three new ideas in this chapter, although you have been using most of them for some time. All you need is some practice, so it's on to the examples.



Example 1 – Locating ducks

The example `Duck` *class in the second example of Chapter 13 included methods intended to change the position of the duck. It would be useful to have another method to determine whether a duck was at a certain position. Its prototype would be:*

```
boolean atPosition (int xPos, int yPos)
```

What would be the implementation of this method?

The position of the duck is stored in two private attributes, called just *x* and *y*. The first value provided to this method is the *x* value to be checked and the second is the *y* value. These parameters might be called *xPos* and *yPos* in the method to make their meaning clear. If *x* is the same as *xPos* and *y* the same as *yPos*, then the duck is at the location and the method should return `true`; otherwise the method should return `false`.

A simple conditional statement is needed to check whether or not the duck is at the particular point. The implementation is:

```
public boolean atPosition (int xPos, int yPos)
{
  if (x == xPos && y == yPos) {
    return true;
  }
  else {
    return false;
  }
}
```

This structure in a method is very common. It states that if a certain condition is true the method returns `true` and otherwise it returns `false`. It is worth remembering that such a method can also be implemented by simply returning the condition; this example could equally well be written as:

```
public boolean atPosition (int xPos, int yPos)
{
  return x == xPos && y == yPos;
}
```

This second form is obviously much shorter but some might find it too terse and the longer form clearer. The choice between the two is a question of style and preference.

To illustrate this point about style one last time, this function could also be written:[1]

```
public boolean atPosition (int xPos, int yPos)
{
  if (x == xPos && y == yPos) {
    return true;
  }
  return false;
}
```

This version works because the second `return` will never be reached if the expression is true.

While all these three versions are equivalent, it would be good practice to choose the one that you prefer and make sure that you always use the same one; good style is consistent.

Example 2 – Duck summoning

Mr Martinmere is becoming increasingly concerned about the locations of his ducks. Some of them are very valuable, and it is annoying if they move too far away (and out of sight of Bruce's sign). Some preliminary work has been done on tracking ducks (in Chapters 11 and 12), and now there will be more.

---

1 My friend Karim gave me this version when he was checking a version of my C++ book. He would always write methods like this in this way. I never would; I'd use the version that just returns the condition. Style is like that; both approaches are fine as long as we're both consistent.

The starting point is the existing class for ducks that we have met before. After a lengthy design process, the ducks have shared out the methods to be written. Buddy has been given a method with the specification:

```
Name:                summon
Object type:         Duck
Purpose:             Moves one duck to the position of another.
Parameters:          a Duck object
Values Changed:      The location of the Duck parameter.
Side Effects:        Display on the screen.
Description:         The Duck supplied as the parameter is moved
                     to the same position as the Duck calling
                     the method.
```

How does he write this method?

First, let's clarify what this method is supposed to achieve. Imagine that *elvis* and buddy are two *Duck* objects. This method would move the two to the same position; the idea is that one *Duck* object "summons" the other. So, to move both to the position of Elvis (although only Buddy would move):

```
elvis.summon (buddy);
```

Obviously this:

```
buddy.summon (elvis);
```

would move Elvis to Buddy's position, which is rather different to the first version.

This short line contains the identifiers of two *Duck* objects, which are being used in different ways. The first is being used in the familiar way from all the method calls we have seen before. The second is different; it is a separate object, and it is being used as a parameter.

Now, some attribute values are going to change when the method is called, but this is not going to happen in the ways we have seen before. The method is going to change the values of the attributes of the object passed as a parameter; the values of the attributes of the other object will remain unchanged. This means that the parameter is a reference parameter – the values of its attributes will be changed.

The heading for the method is:

```
public void summon (Duck d)
```

The method to move a *Duck* is *moveTo*, which has the new position passed as two parameters. We can use this on the parameter:

```
d.moveTo (newX, newY);
```

where *newX* and *newY* are two values representing the new position. These values are extracted from the other object:

```
int newX = getX ();
int newY = getY ();
```

and the complete method is:

```
public void summon (Duck d)
{
  int newX = getX ();
  int newY = getY ();
  d.moveTo (newX, newY);
}
```

An important thing to notice in passing is the different syntax for calling the methods of the parameter as opposed to the current object. Only the first requires a prefix.

This method is all well and good, and would work, but there is a potentially neater way of writing it. There is no need to store the two values in variables; the methods that generate them could just be used as parameters to the method:

```
public void summon (Duck d)
{
  d.moveTo (getX (), getY() );
}
```

This is certainly neater, not to mention shorter.

Example 3 – The rusty trolley

Mr Martinmere is concerned about a dangerous-looking supermarket trolley that has appeared in the pond where Elvis and his friends live. He is especially worried that one of the (valuable) birds may swim into the trolley and so sustain an injury that would reduce their value.

He would like a program that will track each of his ducks and will tell him how far away they are from the trolley. The program should also warn him if a duck hits the trolley.

In order to test the program Elvis has once again been fitted with a tracking device. The pond is marked out in units of one metre, and the trolley is in the water at coordinates 5, 3.

The program should prompt Mr Martinmere to enter the direction in which the tracking device indicates that Elvis has moved. It should then display his distance from the trolley and should print a warning if he is closer than two metres from it. It would provide another warning if he actually swims into it.

What class would be needed for this program? What does the program look like?

A `Duck` class similar to that in Chapter 13 will be needed for this program. There will need to be two private attributes, one for the x position and one for the y position. It will also be handy to store the name of the duck, so the program can be used to track several ducks without confusion. The following methods will also be needed:

```
public void moveNorth ()
public void moveSouth ()
public void moveEast ()
public void moveWest ()

public void setX (int newX)
public void setY (int newY)
public int getX ()
public int getY ()

public double distanceFrom (int xPos, int yPos)
public boolean atPosition (int xPos, int yPos)
```

The implementation of these methods is straightforward. The `distanceFrom` method is the same as in the previous example, and `atPosition` is as in the first example in this chapter.

The program is on the whole straightforward, but it's worth going through it since it contains almost all the Java we've seen so far:

- Conditional statements are used to check the input values;
- Conditional statements are used to check the duck's distance from the Rusty Supermarket Trolley;
- Loops are used to run the whole program until the user has finished;
- Loops are used to request that the user re-enters incorrect values.

The complete classes and program are below. It is certainly worth spending some time going through them all to make sure that you understand what's going on. If you can, get the program off the web site and play around with it.

```
/* Duck.java - modified duck class for tracking.

    Author    : GPH
    Date      : 17th June 2003
    Tested on : Linux (Red Hat 9), JDK 1.4.2
*/

import htpuj.*;

public class Duck
{
  private int x;
  private int y;
  private String name;

  public Duck ()
  {
    x = 0;
    y = 0;
  }

  public void setName (String n)
  {
    name = n;
  }

  public String getName ()
  {
    return name;
  }

  public void setX (int newX)
  {
    x = newX;
  }

  public void setY (int newY)
  {
    y = newY;
  }

  public int getX ()
  {
    return x;
  }

  public int getY ()
  {
    return y;
  }
```

```
  public void moveNorth ()
  {
    y++;
  }
  public void moveSouth ()
  {
    y--;
  }
  public void moveEast ()
  {
    x++;
  }
  public void moveWest ()
  {
    x--;
  }
  public double distanceFrom(int xPos, int yPos)
  {
    int dx = xPos - x;
    int dy = yPos - y;
    return Math.sqrt( (double)(dx*dx + dy*dy) );
  }
  public boolean atPosition (int xPos, int yPos)
  {
    return x == xPos && y == yPos;
  }
}

/* DuckTrafficControl.java - Improved duck tracking program.
   Author     : GPH
   Date       : 19th June 2003
   Tested on : Linux (Red Hat 9), JDK 1.4.2
*/
import htpuj.*;
public class DuckTrafficControl
{
  private Duck duck;
  private final int TROLLEY_X = 5;
  private final int TROLLEY_Y = 3;
  public DuckTrafficControl ()
  {
     duck = new Duck ();
  }
  public void run ()
  {
    char choice = ' ';
    System.out.println ("**** Duck Traffic Control Program ****");
    System.out.println ("");
    System.out.print ("Enter duck's name : ");
    duck.setName (Console.readString () );
    System.out.print ("Enter X coordinate for "
                    + duck.getName () +" : ");
    duck.setX (Console.readInt () );
    System.out.print ("Enter Y coordinate for "
                     + duck.getName () + " : ");
```

```
    duck.setY (Console.readInt () );
    do {
      printMenu ();
      choice = Console.readChar ();
      switch (choice) {
      case 'n':
      case 'N': duck.moveNorth ();
                break;
      case 's':
      case 'S': duck.moveSouth ();
                break;
      case 'e':
      case 'E': duck.moveEast ();
                break;
      case 'w':
      case 'W': duck.moveWest ();
                break;
      case 'q':
      case 'Q': break;
      default: System.out.println (choice
                                   + " is not a valid choice!");
               break;
      }
    } while (choice != 'q' && choice != 'Q');
  }
  public void printMenu ()
  {
    System.out.println (duck.getName () + " 's current position is "
                        + duck.getX () + ", " + duck.getY () + ".");
    System.out.println (duck.getName () + " 's distance from
                         trolley" + duck.distanceFrom
                         (TROLLEY_X, TROLLEY_Y) );
     if (duck.distanceFrom (TROLLEY_X, TROLLEY_Y) == 0.0) {
        System.out.println ("**** " + duck.getName ()
                            + " has hit the trolley!");
     }
    else if (duck.distanceFrom (TROLLEY_X, TROLLEY_Y) <= 2.0) {
      System.out.println ("**** " + duck.getName ()
                          + " is dangerously close to the trolley!");
    }
    System.out.println (" N - Move North");
    System.out.println (" S - Move South");
    System.out.println (" E - Move East");
    System.out.println (" W - Move West");
    System.out.println (" Q - Quit");
    System.out.println ("");
    System.out.print (" Enter choice (N,S,E,W,Q) : ");
  }
  public static void main (String args[])
  {
     DuckTrafficControl dtc = new DuckTrafficControl ();
     dtc.run ();
    }
  }
```

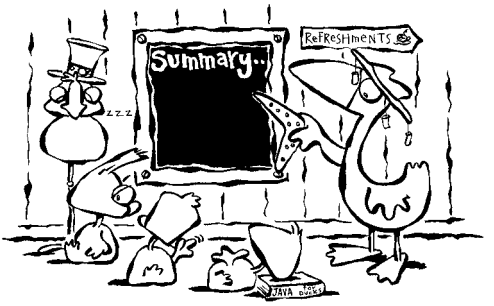
This is a long program, but it does contain examples of just about everything that we've looked at so far. Read it!

16.1 Find and explain the error (or errors) in the following code:

```
public void setLength (int len)
{
  if(len < 0)
  {
    return "ERROR!";
  }
  else {
        length = len;
  }
}
...
System.out.println("Setting length to " + setLength(4) );
```

16.2 Modify the *DuckTrafficControl* class used earlier to incorporate some further error checking. For example, it should be impossible for a duck to move to a position with a negative x or y coordinate. You should also check that the user does not enter a blank string for the duck's name, or else the program output could get confusing.

16.3 Further modifications can be made to this program to make it slightly more user-friendly. For example, it is currently possible to move a duck to the position occupied by the trolley, and for that duck to swim happily on. There is also no option to track another duck, other than terminating the program and starting again. Add this functionality to your code from Exercise 16.2.



Methods can treat parameters in two different ways, depending on what type the parameter is. The simplest case is the value parameter that simply provides a value for the method to process; the value may come from a variable or may

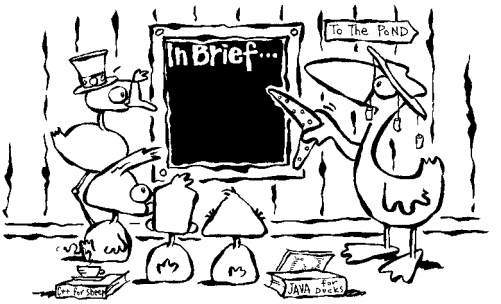
be explicitly included. The slightly more complicated situation is a reference parameter. A reference parameter is always a reference to an object; this reference is passed into the method, which can make use of the object's initial attributes. The same object in the computer's memory is used in the method as in the main program and so any changes made to the object in the method also affect the corresponding object in the main program.

Value parameters allow values to be passed into a method. Reference parameters extend this somewhat in that they can allow values to be passed out of a method, or into and out of a method. However, it must be remembered that this will only work with objects, not primitive data types such as `int` or `char`.

Any valid Java may be used in a method, including conditional statements and loops. You should now be able to use all the Java that you have learned so far and you should be able to write complex methods and classes.

As you write more complicated classes your programs will also become more complex and longer. You may well find that a program of several hundred lines is quite difficult to edit and manage; it can be awkward to keep track of the ever-increasing number of variables in your code. Thankfully, Java, as with most languages, provides various mechanisms for grouping similar variables together into collections, allowing you to refer to any one of a group of variables via a common name. This is what we will look at in the next chapter.

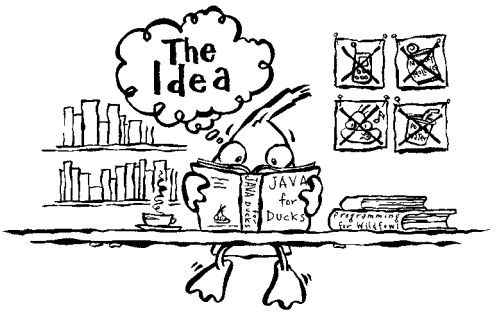

Chapter 17
Collections
0
2
5
DUCKS
ARRAY LIST
Array



By now you should understand what a variable is, how you can use it, and what it is used for. Variables are very good at what they do, but they can only store a single value. What if you need to process and store two integers? More often than not you'll use two integer variables. How about 10 variables, or 100? Before long your program will become unwieldy, repetitive, and extremely large. This is a bad thing and so a neater mechanism is required. It is possible to define variables that can store collections of values.

Java has support for many different ways of storing collections of similar data, and this chapter introduces two of them – arrays and array lists. Array lists are more powerful and flexible than arrays, but initially at least, you can treat them as virtually the same thing.

By the time you complete this chapter, you should be able to write programs that manipulate collections of data rather than many individual variables. You should also be capable of identifying where to use arrays, and where to use array lists, and should understand the fundamental differences between the two.



Java supports many kinds of collection. We'll look at just two in this chapter. The first is the *array*. This is in a sense the simplest collection, and also the least sophisticated. It's a good place to start because you'll find something that looks and works very much like a Java array in almost all programming languages. After arrays, we'll look at another type of collection, the *list*. Finally, we will look at a particular type of list provided by Java – the `ArrayList`. This is rather more sophisticated, mostly in terms of the facilities that Java provides to handle it. It is, however, specific to Java although you will find similar things in other languages (*vectors* in C++ being an obvious example). But first, arrays.

Arrays

An array is essentially a block of memory locations of the same type, referred to by a common name. Each location is referred to as an *element*. As an analogy, we think of the word, "duck", rather than the letters "d", "u", "c", and "k". We have a word (an array) consisting of a number of letters (elements). Taken as a whole the word does something useful, and sometimes we treat the whole word as a single unit. There are times, though, when we are interested in what the individual letters actually are and we are usually very interested in the order in which the letters appear.

The individual locations within an array are known as array elements (corresponding to the letters in our example), and each element can be referred to by its position within the array. So rather than using, say, 30 `String` variables to store the names of students in a class, we can use an array variable containing 30 `String`s. Also, rather than having to refer to the 23rd `String` variable, we can access the 23rd element of the array.

The first stage in using an array is obviously to declare one and to get hold of some suitable memory. Generally, an array is declared as follows:

<variable type> <identifier>[];

And initialised as follows:

<identifier>= new <variable type> [<number of elements>];

It is probably best to introduce an example here. Suppose that we want to create an array of four integers. This is a process of two stages and is achieved by:

```
int numbers[];
numbers = new int[4];
```

The first line[1] tells the compiler to allocate a block of memory large enough to store four integer values. This block can then be referred to by the name *`numbers`*. The square brackets after the identifier tell the compiler that this identifier refers to an array. The `new` keyword on the second line allocates memory to objects (you have seen it before when creating objects of your own), and the 4 in the square brackets gives the number of elements in the array. The variable types must match between these lines, or else the compiler will complain.

As usual it is possible to combine these two steps. These two lines could of course be written as a single statement:

```
int numbers[] = new int[4];
```

Now we have an array, we need to access the individual elements. Values are assigned to the elements in an array using the element's position in the array (this position is also known as the *index*). For example:

```
numbers[1] = 5; // assigns the value 5 to the element with index 1
```

To refer to an element of an array, we use the square brackets seen earlier, with the relevant array index between them. We have in fact seen this syntax before when we have been interested in extracting a command-line argument. It now

1 Some programmers would argue that it would be better style to write `int[]` numbers; This is perfectly legitimate Java.

emerges that the variable `args` in the header of the `main` function:

```
public static void main (Strings args[])
```

is in fact an array of strings.

Now we also know that the first command line argument is stored in *`args[0]`*, which shows us that the index value starts at 0. This means that the last element in an array might not be quite where it is expected to be!

An unwary programmer wanting to assign a value to the fourth element of the *`numbers`* array might write:

```
numbers[4] = 1878; // Error!
```

This illustrates an important property to remember when dealing with arrays. In Java (as in most programming languages), arrays are indexed from 0 upwards. So the first element of the array is referred to using index 0. Similarly, the eleventh element has index 10. More generally, an array with n elements has the first element at index 0 and the last element at index $n-1$.

The value used in the array declaration is the size of the array, not the index of the last element. This is a frequent cause of confusion, and new programmers often find themselves trying to access the fifth element of a 4-element array, for example. And even experienced programmers aren't immune from this (if they're truthful!).

It follows that the number of elements in the array is a number that is both useful and interesting. Java lets us extract this value, as we will see in a moment.

Frequently, if this "out-by-one" error exists in your code, your program will crash when you try to run it and will provide an error message complaining about something called an `ArrayIndexOutOfBoundsException`. This error tells you that the Java interpreter has tried to access an index larger than the size of the array. The compiler will not stop you from trying to go past the end of the array (it isn't possible to spot that this is going to happen when the program is compiled), so this is something the programmer needs to take care of, as many a good program has been flawed by this kind of error. The effects of this error range from simply crashing at inopportune moments to computing the wrong result and continuing blindly on.

Assuming you follow the advice given in the chapter on testing, these errors should be ironed out fairly easy, but it is worth pointing out anyway.

Now for a larger, more useful example. Here's a simple program that makes use of an array.

```
/* ArrayDemo.java - demonstrating a simple array
   Author     : GPH
   Date       : 6th February 2003
   Tested on : Linux (Red Hat 8.0), JDK 1.4.1
*/

import htpuj.*;

public class ArrayDemo
{
   private String names[];
   public ArrayDemo (int numElements)
    {
      names = new String[numElements];
    }
```

```
  public void populate ()
  {
    // Using a for loop, assign a value to each element in turn
    for (int i = 0; i<names.length; i++) {
       System.out.print ("Enter name " + (i+1) + ": ");
       names[i] = new String (Console.readString () );
    }
  }

  public void printContents ()
  {
    System.out.println ("");

    // Using another for loop, print out the values in turn
    for (int j = 0; j<names.length; j++) {
      System.out.println ("names[" + j + "] = \"" + names[j]
                          + "\".");
    }
  }

public static void main (String args[])
{
    if(args.length != 1) {
      System.out.println ("Usage: java ArrayDemo <num of names>");
      System.exit (0);
     }

    int numNames = Integer.parseInt (args[0]);

    ArrayDemo demo = new ArrayDemo (numNames);
    demo.populate ();
    demo.printContents ();
  }
}
```

This small class and driver program demonstrates almost everything you need to know about arrays. The first thing the program does is check that a single command line argument has been passed at runtime:

```
if(args.length != 1) {
  System.out.println ("Usage: java ArrayDemo <num of names>");
  System.exit (0);
}
```

This demonstrates how to find the length of an array:

<arrayname>.length

This is very useful, and it is well worth remembering. It is also worth remembering that there are no brackets after `length`, which is perhaps a little confusing.

Now back to the program. If the required argument is not present, an error string is printed and execution halts. If the argument is present, this is converted into an `int` value, and used in the creation of the program object:

```
int numNames = Integer.parseInt (args[0]);
ArrayDemo demo = new ArrayDemo (numNames);
```

It is worthwhile noting here that the array is initialised within the constructor. This sort of action is commonplace – the constructor is frequently used to initialise variables at the time the object is created, and can remove the need for calling the constructor together with several mutator methods.

The program does nothing more than prompt for the required number of names, add these to the array, and then print out the array's contents. This is not the most useful program in the world, but it does demonstrate the fundamentals of arrays – creation, addition of elements, and subsequently accessing these elements. This structure, especially with some sort of processing before the array is output, is very common indeed in all sorts of tasks.

Creating an array using an initialiser list

So far, we have created arrays in a three-step process; the first is to declare the array by name and type, it is then initialised to the required size, and finally each element is given an initial value. This is a bit long-winded, especially when the values required are known in advance, and so it is possible to perform all three steps at once when we do know the values and types of the elements beforehand.

The syntax is similar to the usual declaration, but includes a list of the initial values between curly brackets:

```
int digits[] = { 0,1,2,3,4,5,6,7,8,9 };
char vowels[] = { 'a', 'e', 'i', 'o', 'u' };
String ducks[] = { "Elvis", "Buddy", "Cilla" };
```

Notice we don't need to explicitly tell the compiler how large each array is; it can work it out for itself by counting the number of elements in the *initialiser list*. This means that `digits` is automatically created as an array of ten integers, for example.

Arrays declared in this way can then be treated like any other so `digits[2]` is 2, `vowels[0]` is 'a', and `ducks[ducks.length-1]` is `"Cilla"`.

Arrays of objects

We have seen that arrays can contain values of any type. In object-oriented programming it is very common indeed for this type to be an object type. Collections of objects can be sorted, searched, and displayed in all sorts of ways. The third example above is in fact an array of objects; `ducks` is an array of `String` objects.

Here's another simple example assuming a simple `Duck` class. The class stores the duck's name and their cricket average. There is a constructor that takes initial values for these two, and a method `toString` that prints them out in some neat format. Creating ducks is simple:

```
Duck tempDuck1 = new Duck ("Elvis", 100.0);
Duck tempDuck2 = new Duck ("Buddy", 50.0);
Duck tempDuck3 = new Duck ("Cilla", 150.0);
```

These ducks can be put into an array. The syntax is just the same as with any other array and, since we know the values in advance, we can use an initialiser list:

```
Duck theDucks[] = {tempDuck1, tempDuck2, tempDuck3};
```

The three Duck objects are now elements in the `theDucks` array. The array itself does indeed also look rather like an object, a concept we'll see more of when we come to array lists in a moment.

Each element of the array is an object, so we can use any available methods of the class on an element. So to find out the details of the duck in the first element:

```
System.out.println (theDucks[0].toString () );
```

or the last:

```
System.out.println (theDucks[2].toString () );
```

or even:

```
System.out.println (theDucks[theDucks.length – 1].toString () );
```

which would work no matter how many elements there were in the array. In these cases, and any others involving objects, it is possible to omit the call to the *toString* method, as the default version of this is called automatically inside a call to `System.out.print` or `println`.

Finally, a loop that would print out the details of all the ducks would probably be useful:

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

As we will see, processing an array using a loop in this way is a very common programming task indeed.

Array checklist

As a last word on arrays (for now at least), some things to remember when using arrays:

1. All elements in an array must be of the same type. The type can be an object type.
2. The array must be declared with the square bracket notation.
3. The array must be initialised with its length in square brackets.
4. Array elements are indexed from 0 to "*array.length−1*".
5. Individual elements are accessed with the notation "*array[index]*".

Arrays are a fine first collection to take a look at. Most, if not all, things that we might want to do with collections can be done with arrays, but we would soon find that some were rather complicated. Operations on collections are very common and so most programming languages provide more complex types to store them; the time has come to look at one provided by Java.

Variable length collections

The description of arrays has hopefully given you some idea of the sort of operations that are often carried out on collections. A complete collection can be displayed, searched, or sorted, for example, and this sort of operation is reasonably easy to program using an array. Things get more complicated with an array when the collection has to be manipulated – when new elements have to be added or deleted. This is possible with an array, but it can be rather complicated and error-prone. At the same time these operations are quite common, so some sort of structure that automatically supports them is often provided.

The problem is that in most programming languages, including Java, arrays have a fixed size; once they are initialised, they have the same number of elements throughout. To add more elements it is necessary to create a new, larger array and to copy the elements between the two. If elements are removed, the array will still occupy the same amount of memory space, which can be wasteful, especially in large and complex programs. Of course, a smaller array can be created and the elements copied but this can be time consuming. Something a little more sophisticated is needed if the size of a collection is going to change. In general terms, most programming languages provide some such collection, known as a *list*.

A Java `ArrayList` is just such a structure. It allows the programmer to dynamically allocate more storage should the number of elements grow, and reduce the allocation should fewer elements be needed for some reason. There are also plenty of handy functions available for searching and so on.

It shouldn't come as a surprise that an array list is defined as a class, and the useful functions are methods. One effect of this is that this section is also an introduction to using the Java built-in classes, and a reminder of the importance of delving in the API for the required documentation.

The API documentation can be found a few clicks away from *`http://java.sun.com/`*. The documentation first provides a list of the standard Java packages. `ArrayList` is found in the `java.util` package, so a bit of scrolling is needed. Clicking through to the `java.util` page gives (among other things) a list of all the classes in the package, and clicking `ArrayList` provides all we could possibly want to know about array lists. Much of this is intended more for programmers planning to implement a Java system; the important bit for our purposes is the list of methods a little way down.

You can read all this at your leisure. You could also have a look round for other classes that might be useful; `Vector`, for example, is another class for storing collections.

We have seen that `ArrayList` is part of the `java.util` package, so if a program is planning to use an array list it is necessary to add the import statement:

```
import java.util.ArrayList;
```

to the start of any program. Now let's use one.

Creating an array list

The documentation also reveals that the constructor for this class is overloaded. We'll make do with just two of these constructors; an array list can be created with an initial default capacity of 10, or it can be created with some other capacity.

The format for using the first version is as you should have come to expect by now:

ArrayList <identifier> = new ArrayList();

and the second is only slightly different:

ArrayList <identifier> = new ArrayList(<capacity>);

As with any other object, an array list is referred to by an identifier. This declaration calls the appropriate constructor which takes care of allocating

the correct amount of memory. So:

```
ArrayList myArrayList = new ArrayList (4);
```

creates an empty `ArrayList` called *`myArrayList`*, which can initially hold four elements. While this:

```
ArrayList myArrayList = new ArrayList ();
```

might intuitively appear to create an array list with no capacity, but actually (according to the API documentation) creates one that can initially hold 10 elements.

Adding elements

Array lists (which we will now simply call *lists*) can store elements of any type, provided that they are objects. This means that it is complicated (but possible) to use lists for the base types, so we'll use the simplest object type we have – `String` – in these examples. We'll use a simple list of names, declared:

```
ArrayList names = new ArrayList ();
```

Elements are added to a list using the `add` method, and once again this is overloaded. The default is for the newly added element to appear on the end of the list. For example,

```
names.add ("Elvis");
```

will add a `String` object with the value `"Elvis"` to the end of *`names`*. It is also possible to specify where in the list you wish the new element to be inserted. For example:

```
names.add (0, "Buddy");
```

will add the `String` object, `"Buddy"`, to index 0 of *`names`*, right at the start.

If there is no space in the list for a new element it will grow automatically. If an element is added at a particular place all the other elements will automatically shuffle along to make room, and the list will grow if needed. It really is rather neat.

You can actually add objects of any type to an `ArrayList` – they are treated internally as being different instances of the same class, even though to the program they may be wildly different. Unfortunately, this means that `int`, `char`, `double`, and so on, cannot be added directly to an `ArrayList`, as they are not objects. We will discuss how they can be indirectly added later on.

But first we should take a look at what is in our list.

Displaying the array list

There is no handy method to print out a list, so we have to resort to the strategy used for arrays and use a loop. The `size` method will provide the number of elements in the list, so the loop is determinate. We also need a method for extracting a particular element; the API documentation reveals that this is called `get`.

The loop to display the list is then very similar to the ones we used to print out an array:

```
System.out.println ("The list now contains: ");
for (int i = 0; i < names.size (); i ++) {
  System.out.println (i + ": " + names.get (i) );
}
```

And running this would reveal that the names list currently contains:

```
0 Buddy
1 Elvis
```

Deleting elements

A list can change size during the running of a program. We've discussed how to make it larger, so it would make sense to talk about how to remove elements. There are two possibilities here:

- We want to remove a particular element, no matter where it happens to be in the list.
- We want to remove the element at a particular location, no matter what element is stored there.

There is a method, imaginatively named `remove`, which can be used in the second of these two ways. This:

```
names.remove (0);
```

will remove the `String` object that is stored in the "0-th" (that is the first) element of `names` if it exists.

Removing a particular element no matter where it is in the list is a little more complicated,[2] and is a two-stage process. The first stage is to find the index required using the `indexOf` method:

```
int position = names.indexOf ("Elvis");
```

and then this value is used to remove the element:

```
names.remove (position);
```

Of course these two steps can be combined:

```
names.remove (names.indexOf ("Elvis") );
```

In either case it is only the first element with this value that is removed.

Finally, if we want to be drastic, there is a method to remove all the elements in a list. This:

```
names.clear ();
```

will erase every element currently belonging to *`names`*.

Changing elements

If we want to simply change an existing element, we can use the `set` method:

```
names.set(0, "Cilla");
```

will change the first element of the `ArrayList` to match the `String` `"Cilla"`. This means that we only require one operation to change an element, rather than the two that would otherwise be needed (remove an element, insert its replacement at the corresponding location).

---

2 If you check the documentation for the `Vector` class you will see that it has an overloaded version of `remove` that does this in just one step.

Other methods

The documentation also describes a few other methods that can come in useful:

- `contains` – returns true if an element is found in the list.
- `isEmpty` – returns true if the list is empty.
- `lastIndexOf` – returns the last index of a value (compare this with `indexOf` used above).

There are a few more – check the documentation for details.

Using elements

Using an element of a list is a little more complicated than using an element of an array; in a way this is the trade-off for all the neat functions. Elements of a list cannot be referenced in the same way as those of an array, as they do not use the same notation in Java. The syntax that is required can look a little odd at first.

The element at a given position is found as follows:

```
String aString = (String)names.get (2);
```

which will cause the `String` *`aString`* to refer to the object stored in the third element of the list. This notation may seem a little strange at first, but it is actually an example of casting (which you met earlier – casting from a `double` to an `int`, for example).

The notation tells the compiler that we wish to treat the object accessed in this way as a `String`. Internally, an `ArrayList` treats all its elements as if they belonged to a single class, called `Object`. This means that objects of different types (*`Duck`* and `String`, for example) can be stored in the same `ArrayList`. This is very neat and can be very powerful but it also means that we have to explicitly tell the compiler how we want to treat each object we access within the list. So if the element were a *`Duck`* object and was in a list called *`ducks`*, we would use the statement:

```
Duck aDuck = (Duck)ducks.get (2);
```

We need to be careful here, as if we attempt to cast to the wrong object type, we can end up with all sorts of weird happenings, ranging from malfunctioning code to compiler errors. With this in mind, it is usually safer to restrict a `ArrayList` to storing a single object type, thereby avoiding such errors. In any case storing several object types in an `ArrayList` brings other problems. For example, it is not easy to search for a particular object – usually you are required to use methods of a class to determine whether this is the object you are looking for. The lists also become harder to sort, and any objects in the list, which do not override the default `toString` method, can cause problems with any output required.

As a final word on `ArrayLists`, we have seen that base types – `int`, `char`, and so on – cannot be added directly to a list, as they are not objects. The designers of Java saw that this sort of situation may well arise and cause problems, so they have provided a solution, known as a wrapper class. This is a way of encapsulating a primitive value in an object, so it can then be manipulated in the same way as any other object.

The wrapper class for `int` is `Integer`, that for `char` is `Character`, that for `double` is `Double`, and that for `boolean` is `Boolean` (we are unlikely to use

this last one however). These classes are created by taking the corresponding primitive value as a parameter, and can then be manipulated via methods. It works like this:

```
ArrayList numbers = new ArrayList ();
Integer myInteger = new Integer (2);
numbers.add (myInteger);
```

Compare this with the following, which is not allowed:

```
ArrayList numbers = new ArrayList ();
int myInteger = 2;
numbers.add (myInteger); // Not Allowed!
```

To extract the `int` value from this Integer object, the `intValue` method is provided:

```
int iVal = myInteger.intValue (); // iVal becomes 2
```

So an `int` value can be encapsulated (or wrapped) in an object, and that `int` value can later be extracted from that object. Now you should be able to use array lists to store lists of numbers (but you might well decide that arrays are less bother!). The same approach works with the corresponding methods for the `Character` and `Double` classes:

```
Double myDouble = new Double (3.14159265);
double dVal = myDouble.doubleValue (); // dVal == 3.14159265

Character myCharacter = new Character ('x');
char cVal = myCharacter.charValue (); // cVal == 'x'
```

Collections as parameters

It is quite common for collections to be passed to methods as parameters. Array lists are objects and are therefore passed by reference, as explained in the previous chapter. Arrays, which are not really objects or base classes are a little more complicated; this quick section explains why.

This is probably best illustrated with an example, so here is a description for a simple method. The method (which is admittedly not very useful, but it's concepts we're after here) just trebles the integer value passed to it; but the value is not returned. It looks like this:

```
public void treble (int val)
{
  val *= 3;
}
```

Suppose for some reason a program is needed to treble all the elements of an integer array, an attempt to do this might be:

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

for(int i = 0; i < numbers.length; i ++)
{
  treble (numbers[i]);
}
```

Now, this code will compile and run, but it won't achieve very much; it would turn out that the array remains unchanged. To understand why this is,

remember that primitive types are always passed by value, while object types are passed by reference. In this example, every time the *treble* method is called, a single primitive value is passed as a parameter. So it is the value of this primitive that is copied and the method trebles this copy. There is no change to the original.

An alternative approach would be to rewrite the method so that it took the whole array as a parameter. This method might be written:

```
public void trebleArray (int array[])
{
  for(int i = 0; i < array.length; i ++) {
     array[i] *= 3;
  }
}
```

The code to call the method would also require a quick rewrite:

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

trebleArray (numbers);
```

This would work as required, and the array would contain the expected values

```
{ 3, 6, 9, 12, 15 }.
```

This works because, for the purposes of method calls, an array is passed as if it were an object – by reference. This means that in the example above, the array which is manipulated by the *trebleArray* method is actually the array passed to the method initially. As no copy is made (and so the same memory locations are used), the calculations performed by the method actually affect the array directly.

An array in Java is a strange beast. It isn't a primitive data type as such – you can't declare something as an array type (as you can just declare an `ArrayList`), it must be declared as an array of values of the same type. However, it isn't quite an object in that you don't initialise an array in the same way as a *Duck*, for example, and there are no array methods (the *array.length* notation is more like an attribute of an array and has no brackets).

This is a subtle but important concept, which can be used to write elegant programs, but can also cause elusive logic errors within your code if you're not careful.

Now we can move on to look at a very common operation on collections – sorting.

Sorting

While collections of objects are often useful, it is a frequent requirement of a program to sort the elements of a collection into some meaningful order. Once the values are stored in some suitable collection, it is fairly easy to sort them into whichever order you choose. Some of Java's built-in classes provide a method to sort the values automatically, but `ArrayList` doesn't and so they, like arrays, have to be sorted "by hand".[3]

---

3 This is not strictly true – the `Collections` class (in package `java.util`) provides a static method called `sort`, which will sort an `ArrayList`, and other types of collection. Read about it in the API documentation or elsewhere on the web, and try it out for yourself!

Collections are sorted by using what are known as sorting algorithms. There are many different algorithms, some simple, some complex, some efficient, and some slow. Entire books have been written on the subject, so we won't be blazing any trails in this section. Suffice to say that the examples you will be using and the programs you will write later are simple, and do not contain huge amounts of data, so simple algorithms will be more than satisfactory.

In most books on learning to program, in whatever language, the first sorting algorithm to be introduced is one called *Bubblesort*, and this is the algorithm we will be using. It is not the most efficient algorithm, but the code is easy to write and understand. In any case, describing Bubblesort in a book about programming is such a fine tradition that it should obviously be included here.

First, we'll explain how the algorithm works, and then we'll illustrate this with an attempt to use it to sort an array. Bubblesort will work the same way with any sort of data proved that, obviously, the values can be compared in some way. For the moment, we'll use integers to keep things simple.

The basic idea of Bubblesort is that we step through a list of values (usually numbers or strings), and compare each pair of adjacent elements with one another, according to some comparison rule. If the elements in the pair are in the wrong order, they are swapped over, and we move on to compare the next two items. So the higher values are shifted towards the end of the list, and the lower values towards the bottom.

It may be best to visualise this process with a list of numbers. Let's suppose that we want to sort this list of integers into ascending order. The comparison required is simply "less then", so if the number of the left is less than the number on the right they are in the correct order. The list initially is:

```
8 9 6 4 3
```

In the first step, the first two items (8 and 9) are compared, and found to be in the correct order. So the list remains unchanged. Next the 9 and 6 are compared, found to be in the wrong order, so they are swapped, making the list look like this:

```
8 6 9 4 3
```

Next the third pair of items – 9 and 4 – are compared, found to be in the wrong order, and swapped:

```
8 6 4 9 3
```

and finally, the 9 and 3 will be swapped after the final comparison:

```
8 6 4 3 9
```

So we say that the highest value has "bubbled" its way to the end of the list, hence the name of the algorithm. In fact, the highest value is guaranteed to find its way to the end of the list after the first set of comparisons. Try it and see!

Of course the list is not sorted yet. The algorithm must be applied again; in fact, the algorithm must be applied until the list is correctly sorted.

Continuing, the second set of comparisons would proceed as follows:

```
6 8 4 3 9 (8 and 6 compared and swapped)
6 4 8 3 9 (8 and 4 compared and swapped)
6 4 3 8 9 (8 and 3 compared and swapped)
6 4 3 8 9 (8 and 9 compared, no swap required)
```

So now the second highest number, 8, has bubbled to the end of the list.

In this case it takes another three passes to fully sort this list, making 5 passes for a list of 5 values. In fact, an unsorted list of *n* values will be guaranteed to be in the right order after *n* passes. This is obvious once you appreciate the pattern of the highest elements bubbling to the end.

Bubblesort is not as efficient as some alternative algorithms (others would take fewer passes to sort the list above, for example, or would need fewer comparisons), but it is easy to visualise, and in practice it is fine for sorting small lists of values.

Now, armed with this quick and dirty introduction, let's implement this algorithm in Java, and sort an array of integers. The implementation involves two nested `for` loops – one counting the number of passes through the algorithm (which is determinate because we know how many elements there are), the other counting the comparisons during each pass.

Obviously, before we can even think about sorting an array, we need to create it:

```
int numbers[] = { 8, 9, 6, 4, 3 }; // unsorted array
```

Now, assuming again that we plan to sort these values into ascending order, the basic logic of the Bubblesort algorithm is:

```
for every pass through the list:
    for every pair of elements (x,y):
        if x is larger than y, swap them over
```

So a first attempt at coding our `for` loops would be:

```
for (int passes = 0; passes < numbers.length; passes ++) {
  for (int index = 0; index < numbers.length; index ++) {
   /* If in the wrong order, swap elements round */
   if (numbers[index] > numbers[index+1]) {
     numbers[index+1] = numbers[index];
     numbers[index]   = numbers[index+1];
   }
  }
}
```

This does not work! This code would compile, but you wouldn't get far in using it, as there are at least two problems with it. First, the mechanism for swapping the elements is flawed. Let's take the first pass as an example:

```
index     0 1 2 3 4
elements  8 9 6 4 3
```

When the loop comes to swapping the 9 and 6 over, here's what happens:

```
numbers[2] = numbers[1];
```

So at this point, the values in the array are:

```
index     0 1 2 3 4
elements  8 9 9 4 3
```

You might have spotted the problem already – in the assignment statement above, the value 6 has been lost forever and we have two 9s. So when the second statement in the swapping section gets executed, we are trying to copy the same value back to where it came from; the actual effect of this step is that the array is unchanged since the two are the same.

The way to work around this is to use a temporary (or buffer) variable to store the value that would otherwise be lost:

```
int tmp; // Buffer variable
if (numbers[index] > numbers[index+1]) {
     tmp = numbers[index]; // store the 1st variable
     numbers[index] = numbers[index+1]; // move the 2nd variable
     numbers[index+1] = tmp; // Copy the 1st variable back
}
```

This only solves one of the two problems with the first attempt to program the algorithm. The other problem is also subtle, but one we have hinted at earlier in the chapter. Let's take a close look at the inner loop – the one that compares successive pairs of elements. As matters stand it is executed 5 times, once for each element of the array. The problem comes at the end of the array when the comparison would be:

```
if(numbers[4] > numbers[5])
```

Again, you might be able to see the problem now. Arrays are indexed from 0, and so our 5-element array is indexed from 0 to 4. Hence in the statement above, we are attempting to step past the end of the array. This causes an `ArrayOutOfBoundsException` run-time error, which stops your program in its tracks before it has a chance to do anything useful.

For an array of 5 elements, there are only 4 pairs of elements to compare on each pass (0 and 1, 1 and 2, 2 and 3, 3 and 4), so the second of the `for` loops should only execute n−1 times for an array of n elements.

A much better version of the algorithm, with these problems ironed out, looks something like this:

```
int numbers[] = { 8, 9, 6, 4, 3 };
int tmp;

for (int passes = 0; passes < numbers.length; passes ++) {
   for (int index = 0; index < numbers.length-1; index ++) {
      if (numbers[index] > numbers[index+1]) {
          tmp = numbers[index];
          numbers[index] = numbers[index+1];
          numbers[index+1] = tmp;
      }
   }
}
```

Notice that in this version the declaration of the *tmp* variable has been moved outside of the `for` loops. This does not affect the running of the program, but it is more efficient, as this variable is now declared only once, and used only when a swap is needed. If it were declared inside the `if` statement, there would be a variable called *tmp* created every time a swap was needed, and destroyed when the next closing brace is reached. In a program this small, this would not make a significant difference, but thinking about efficiency issues like this is a good habit to get into, as when you begin to write more complex programs, this sort of efficiency gain can make programs run significantly faster.

Now, the only thing left to do is to put this code into a reusable form, so let's write a method to sort an array.

```
public void sortArray (int array[])
{
    int tmp; // temporary buffer used in swap
```

```
    for (int passes = 0; passes < array.length; passes ++) {
        for (int index = 0; index < array.length-1 ; index ++) {
            if (array[element] > array[element+1]) {
                tmp = array[element];
                array[element] = array[element+1];
                array[element+1] = tmp;
            }
        }
    }
}
```

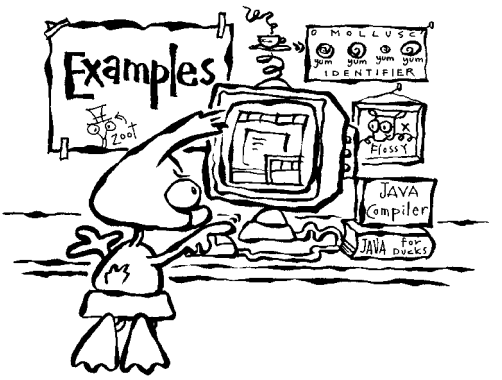
This is a fine and useful piece of code that you are welcome to use in your own programs. It will work, pretty much unchanged, with any collection of values of any type. It's not very difficult to change it to work with array lists provided that the elements of the list can be compared in some way. If we assume that the elements are strings:

```
public void sortArrayList (ArrayList a[])
{
   String tmp; // temporary buffer used in swap
   for (int passes = 0; passes < a.size (); passes ++) {
     for (int index = 0; index < a.size () -1 ; index ++) {
       if (a[element] > a[element+1]) {
         tmp = a[element];
         a[element] = a[element+1];
         a[element+1] = tmp;
       }
     }
   }
}
```

Arrays or array lists

The question now arises of when to use an array and when to use an array list. For the moment we'll suggest a simple rule. If the collection is never going to change in size, use an array. If it is going to change in size (or if you're not sure), use an array list.

Now it's time to practise all this with some examples.



Example 1 – A busy duck

You will have gathered by now that Elvis is a busy duck. As well as playing cricket he also has to meet the deadlines of his Java programming course and all the other demands

faced by a duck today. He has decided that he needs to be better organised; sadly his pocket money does not stretch to a diary so he has decided to write a program in Java.

Being a properly object-oriented duck, Elvis has decided to implement the class required for his program first. He needs to store details of his many tasks, and will need to be able to add new ones and remove ones he has done. He also reckons that sometimes he will just need to know how many tasks he has to do.

Elvis's list of jobs is clearly a collection. There is no forecast of how many jobs he might have at any point in time, so a dynamic array list will be needed. The class itself will have just this list as an attribute; its methods could include:

- A method to add a new job;
- A method to delete a job once completed;
- A method to list the current jobs;
- A method to display the number of jobs currently in the list.

We'll also need a constructor, of course. Since the requirements of Elvis's eventual program are still unclear we'll also include another method that might come in useful; this one will warn him if he has jobs to do.

The class might be called `Reminders`. Since it's going to use an array list it will need to start:

```
import java.util.ArrayList;

public class Reminders
{
}
```

The first thing to add is the attribute to store Elvis's list of jobs. This is an array list:

```
private ArrayList jobs;
```

This can store any sort of object, of course. It doesn't matter at this stage what sort of object it is. In this example, we'll assume that strings are enough for Elvis's purposes, but it would be easy enough to change the class later.

Now let's work through the methods. Array lists do a great deal for us automatically, so most of these are quite short. The constructor comes first; it just needs to create the empty list which is done by calling the constructor of the `ArrayList` class:

```
public Reminders ()
{
  jobs = new ArrayList ();
}
```

Adding a new job is easy, thanks to the add method of the `ArrayList` class. This will need a string containing the description of the new job as a parameter, and we'll need to create a new separate `String` object. It's a void method and the body is just one line:

```
public void addJob (String newJob)
{
  jobs.add (new String (newJob) );
}
```

Deleting a job is a little more complicated. It would be possible to define a method very similar to *addJob*:

```
public void removeJob (String jobDone)
```

and this would work. But it is possible that because of some error there is no job with the appropriate name to delete. This is surely going to be of interest to someone writing a program using this method, and it makes sense to tell them if it has happened. The simplest solution is to define a method that removes the job and returns a Boolean value – `true` if the job was found and `false` if not:

```
public boolean removeJob (String jobDone)
```

This method can use the `contains` method of `ArrayList` to see if the job is there to be removed. If it is not, `false` is returned. Otherwise the `indexOf` and `remove` methods do the job, as explained earlier in the chapter, and `true` is returned.

```
public boolean removeJob (String jobDone)
{
  if (jobs.contains (jobDone) ) {
     jobs.remove (jobs.indexOf (jobDone) );
     return true;
  }
  else {
     return false;
  }
}
```

Now to keep Elvis up to date with the number of jobs in his list. The method to display the number of jobs is nothing more than a call to the `size` method of `ArrayList` since the size is simply the number of jobs in the list:

```
public int countJobs ()
{
  return jobs.size ();
}
```

Similarly a call to the `isEmpty` method will warn Elvis if there are jobs to do:

```
public boolean jobsToDo ()
{
  return !jobs.isEmpty ();
}
```

This last method is worth a quick closer look. The logic of the method is:

```
IF the list is not empty
  THEN Elvis has jobs to do, return true
OTHERWISE IF the list is empty
  then Elvis has no jobs to do, return false
ENDIF
```

This could be written in Java as:

```
if (jobs.isEmpty () == false) {
  return true;
}
else {
  return false;
}
```

This is correct but is needlessly long. In general, any attempt to use the == operator with Boolean values can be written in a much shorter form. This is also a handy way to avoid possible errors; remember the warning about using = in this situation by mistake.

There is now one method to go; Elvis wants to be able to print out a list of all his jobs. There is currently no way to output anything at all, so we'll implement this by first adding a `toString` method to print out the details of a single job. Since we know that all the jobs are `String` objects, this is quite easy:

```
public String toString ()
{
  String rem = "Current Jobs:\n";

  for (int i = 0; i < jobs.size (); i ++) {
   rem += "Job #" + (i + 1) = ": "
        + jobs.get (i) + ".\n";
  }
  return rem;
}
```

The method to print all the details then just needs to print this string returned from this method:

```
public void listJobs ()
{
  System.out.print (toString () );
}
```

Finally, a better version of this method would deal with the possibility that Elvis might be at total leisure with no jobs to do. The *`jobsToDo`* method can detect that, and a suitable message can be displayed instead of the list of jobs:

```
public void listJobs ()
{
  if (jobsToDo () ) {
    System.out.print (toString () );
  }
  else {
    System.out.println ("There are no current jobs.");
  }
}
```

The class is complete!

Example 2 – Testing jobs

Elvis is keen to launch straight into writing his program using his new class. Buddy is somewhat alarmed at this plan, and points out that Elvis should really test his class properly first. After all, he could end up with a program that doesn't work properly because of some error in the class.

Buddy is, of course, quite right. Elvis needs to write a program so that he can test the class. This program (such programs are usually called "driver" programs) doesn't need to do anything especially complicated; it just needs to allow Elvis to use all the methods of his class so that he can test them.

This is a case where the `main` method of the class can come in useful. Elvis's class is intended for use by other programs; it doesn't "do" anything itself. The plan is to write a quick program in the `main` method to help with the testing.

This will mean that the class can be "executed" and tested; it also means that the driver program will always be with the class and so will always be available for testing.

There are two approaches to testing a class in this way. The simplest is simply to call all the methods in some sensible way and to examine the effects. The programmer writes a quick method that uses the class in a sensible way and reports what is going on; if all looks well from this then the class is assumed to work. This is admittedly not a particularly structured way of testing, but it can be sufficient for simple classes.

The first stage is obviously to create an object, so in this program:

```
public static void main (String args[])
{
  Reminders elvisJobs = new Reminders ();
}
```

The methods can be tested in any order, so we might as well start with the add method. Adding a couple of objects to the list and then displaying its contents would be a good start:

```
System.out.println ("Adding Some Jobs...");
elvisJobs.addJob ("Quacking");
elvisJobs.addJob ("Cricket Practice");

elvisJobs.listJobs ();
```

Now to see if the method to remove an element works. Let's remove the first element and then observe results:

```
System.out.println ("Deleting \"Quacking\"...;");
if (elvisJobs.removeJob ("Quacking") ) {
  System.out.println ("Removed!");
}
else {
  System.out.println ("Not Found!");
}
elvisJobs.listJobs ();
```

This call also needs to handle, of course, the possibility that the element is not found. We know that it should be, but if there was an error it might possibly not be found. The next logical step is to see what happens if we try to remove an element that is not on the list:

```
System.out.println ("Deleting \"Reading Poetry\" ...");
if (elvisJobs.removeJob ("Reading Poetry") ) {
  System.out.println ("Removed!");
}
else {
  System.out.println ("Not Found!");
}
elvisJobs.listJobs ();
```

Finally we need to make sure that the method to count the number of jobs works correctly:

```
System.out.println ("There are currently " +
                    elvisJobs.countJobs ()
                    + " jobs to do.");
```

The method for listing all the jobs has been tested already, so all that remains now is to check the method that tells Elvis if he has any jobs to do:

```
if (elvisJobs.jobsToDo ()) {
  System.out.println ("There are jobs to do.");
}
else {
  System.out.println ("There are no jobs to do.");
}
```

Sometimes errors can be made in a program when the collection is empty, so it would be a good idea to repeat these last two tests with any empty list. The list is emptied:

```
System.out.println ("Deleting \"Cricket Practice\"...");
if (elvisJobs.removeJob ("Cricket Practice")) {
  System.out.println ("Removed!");
}
else {
  System.out.println ("Not Found!");
}
```

and then the final two tests are repeated.

This simple method would then be run, and the programmer could examine the results and see if they were as expected. Hopefully they would resemble:

```
Adding Some Jobs...
Current Jobs:
Job #1: Quacking.
Job #2: Cricket Practice.
Deleting "Quacking"...
Removed!
Current Jobs:
Job #1: Cricket Practice.
Deleting "Reading Poetry"...
Not Found!
Current Jobs:
Job #1: Cricket Practice.
There are currently 1 jobs to do.
There are jobs to do.
Deleting "Cricket Practice"...
Removed!
There are no current jobs.
There are currently 0 jobs to do.
There are no jobs to do.
```

This is a very "rough and ready" approach to testing, and the grammar when there is only one job is inexcusable. Sometimes something a little more structured is needed.

Example 3 – Testing jobs more thoroughly

Buddy tells Elvis that his method for testing is a step in the right direction, but is still far from convinced. He shows Elvis the test plan that he has drawn up for the class, and asks how that can be run.

One approach that Elvis could use to carry out Buddy's test plan is obviously to rewrite his current `main` method. This would work, but then Buddy might

change the plan and Elvis would have to go through all the upheaval of changing his program again. Elvis is going to have to think of a better method.

There is indeed a better way to write this method. The approach is to write a simple menu program that allows each of the methods to be called on an object. This can then be used to carry out whatever test plan is required. The program does not have to produce a dialogue that is especially neat since it will only be used by a tester (who may well also be the programmer). It might show a menu like this:

```
1. Add a Job
2. Delete a Job
3. List Current Jobs
4. Count Current Jobs
5. Jobs to do?
0. Exit
```

with each menu option corresponding to one method.

Programs like this take more time to write than the simple approach from the last example, but all such programs for any class follow the same basic pattern. In this case the program must first create an object in the same way as the last attempt. This is manipulated in a menu controlled by a loop and a `boolean`:

```
Reminders elvisJobs = new Reminders ();

boolean finished = false;

do {
   System.out.println ("1. Add a Job");
   System.out.println ("2. Delete a Job");
   System.out.println ("3. List Current Jobs");
   System.out.println ("4. Count Current Jobs");
   System.out.println ("5. Jobs to do?");
   System.out.println ("0. Exit");
   System.out.println ();
   System.out.print ("Choice: ");
   choice = Console.readInt ();
} while (!finished);
```

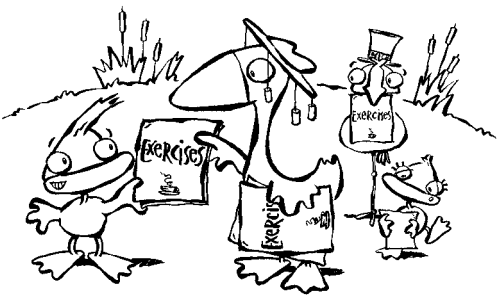
The choice entered by the user[4] is then processed in a simple `switch` statement. An `if` statement could be used, of course, but the `switch` is rather neater in this case. Each part of the `switch` is straightforward even if the statement itself is quite long.

```
switch (choice) {
  case 1: System.out.print ("Enter new job: ");
          elvisJobs.addJob (Console.readString () );
          System.out.println ("Job Added!");
          break;
  case 2: System.out.print ("Enter job to delete: ");
          if (elvisJobs.removeJob (Console.readString ())) {
```

4 It would, of course, be an extremely good idea to check this version of the method before carrying on to worry about what the user had entered. There is very little point in trying to process some input from the user without first checking that it's as expected!

```
            System.out.println ("Job Removed!");
          }
          else {
            System.out.println ("No Such Job!");
          }
          break;
  case 3: elvisJobs.listJobs ();
          break;
  case 4: if (elvisJobs.countJobs () == 1) {
           System.out.println ("There is 1 job to do.");
          }
          else {
           System.out.println ("There are " +
                     elvisJobs.countJobs ()+ " jobs to do.");
          }
          break;
  case 5: if (elvisJobs.jobsToDo () ) {
            System.out.println ("There are jobs to do!");
          }
          else {
            System.out.println ("There are no jobs to do!");
          }
          break;
  case 0: finished = true;
          break;
  default: System.out.println ("Invalid Option!");
          break;
}
```

Either of the two approaches described here can be used to test a class. The choice between the two will usually come down to preference, or perhaps to the details of the test plan. At least the grammar has been repaired in the second version.



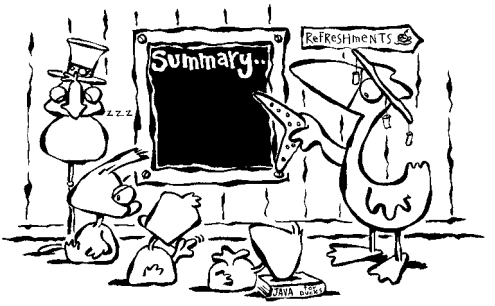
17.1 Explain the error(s) in the following code:

```
ArrayList a = new ArrayList (4);
Integer num = 2;
a.add (num);
System.out.println ("Value is " + a.get(0) );
```

17.2 Write a *SortArray.java* program which receives numbers from the user via the keyboard, rather than hardwired into the code.

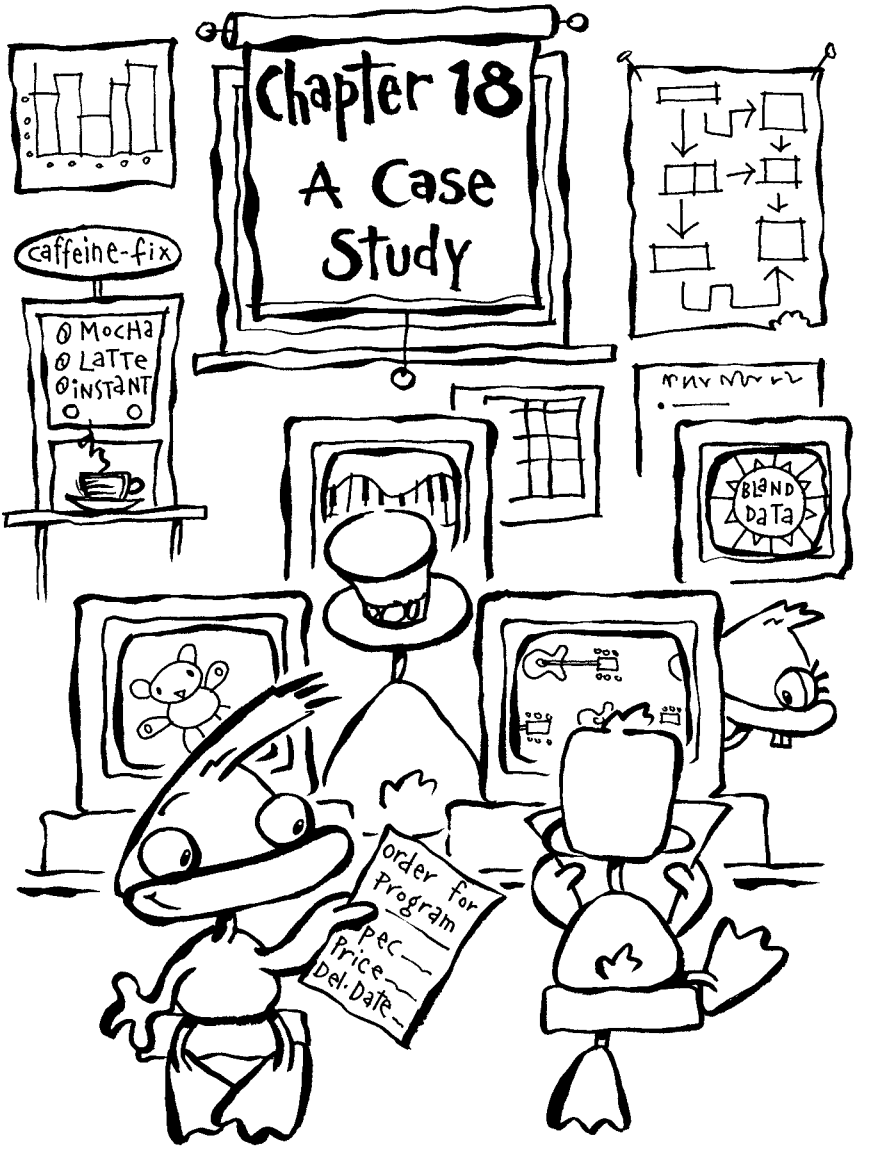
17.3 Write a *SortArrayList.java* program which works in much the same way.

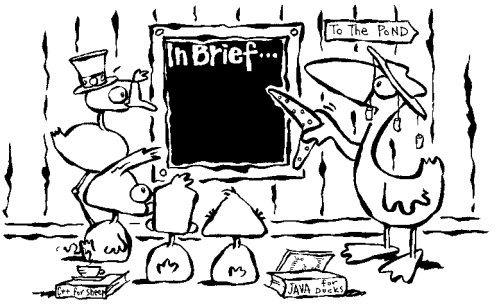
17.4 Add a priority attribute to Elvis's class for his jobs (use an integer). Write a program that displays Elvis's current most urgent job, defined as the job with the highest priority. What happens if more than one job has the highest priority?



By now, you should understand the differences between variables and collections of variables. You should understand why collections are useful, how and why we use them, and should be able to choose arrays or `ArrayLists` depending on the situation. You should be able to apply the Bubblesort algorithm to sort arrays and `ArrayLists`. You should also know why primitive variable types cannot be added directly to an `ArrayList`, but that they can be wrapped inside objects to do so.

And that is very nearly all there is to it. To finish off, the next chapter provides a case study that ties everything we've done together …


Chapter 18
A Case Study
caffeine-fix
MOCHA
LATTE
INSTANT
BLAND DATA
Order for Program
Pec
Price
Del. Date

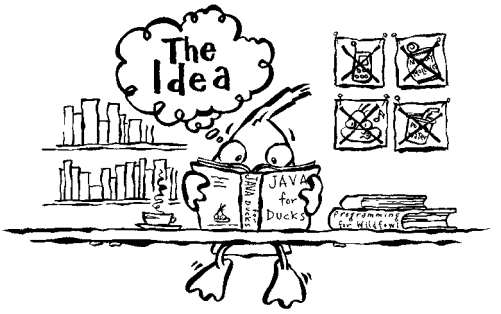


This is nearly the end. You have now seen, and had the chance to practise, all the Java that there is in this book. This chapter presents some reasonably long programs that make use of all of this Java. It goes through the whole process of developing a complete object-oriented program; it starts with the description of a problem and then goes through the process of designing and developing suitable classes and then the programs that use them.

There have been many example programs in this book so far. All of them have been intended to illustrate the particular new part of Java that was being described in the current chapter. The programs in this chapter draw from all of the Java in the book. You should certainly be able to understand all these programs, and you should be able to write similar programs yourself.

After reading this chapter you should understand how a Java program is written. You should know why the classes are designed and developed first and how the programs that make use of the classes are then designed and developed. You will also have seen once again some of the advantages of object-oriented programming and code reuse.

This chapter will also in passing highlight some of the areas that have not been covered. These are for you to find out about later!



It is well known that all the ducks on Mr Martinmere's reserve have become keen cricketers. As time has gone on there have been more and more matches and the matches have been more and more keenly contested. Buddy (whose spectacles have hindered his progress as a cricketer) has been put in charge of keeping various statistics; he has been doing this to date by keeping paper-based records.

A challenge has now been received from the ducks of a neighbouring reserve. The ducks have decided to use Buddy's statistics to help them select the best team. It soon becomes apparent that the paper system is not storing all the

information required, and so it is decided that a computerised system will be needed.

At the moment Buddy is simply storing all the scores from all the matches. When the ducks are interested in more complex statistics, such as which duck is averaging the most runs each match, he is working this out by hand.[1] This is complex and tedious so Buddy is more than willing to agree to a computerised system.

The approach

The immediate need here is to calculate various statistics, but it seems more than likely that the information used may also be of wider interest. There is more than one problem here, even if the first one is the most pressing and the others have not been properly specified yet. The object-oriented approach to a set of problems such as this is obviously to develop the classes that the programs will need first and then to develop the programs themselves. The classes will be used in many programs, a fine example of code reuse.

Some further analysis of the requirements is needed. The first step is to identify the object types that are needed and the attributes and methods of each. It should be reasonably easy to design and implement some suitable classes from the results of this analysis.

It would also be possible, of course, to leap straight in and develop a single program that would solve the current problem. This is possible, and it is quite likely that a working program could be developed sooner, but in the long term there would be more effort required when more programs were needed. In any case, we all know that developing any part of a program without thoroughly understanding the requirements is a sure recipe for problems in the future.

The requirements

The ducks need to identify their best players. Buddy can currently supply the following information for each match they have played:

- The names of the two teams.
- The names of the ducks in each team.
- The number of runs scored by each duck.
- The number of overs bowled by each duck.
- The number of runs conceded by each duck when bowling.
- The number of wickets taken by each duck.

There are many possible measures of cricketing prowess that can be used. Some are incredibly complicated, but the ducks are quite happy to make do with something simpler. They will be content with being able to calculate three values for each duck:

- The batting average – the number of runs scored on average in each match.

---

1 Er, wing?

- The bowling average – the number of runs conceded for each wicket taken.
- The run rate – the average number of runs conceded in each over.

Buddy decides that some sort of database system is needed. He wants to be able to enter the name and score of each duck and would then like the system to calculate the various statistics. He can then collect these and pass them to the ducks' selection committee.

Armed with all this information the first step is to identify and design the classes that will be needed.

The classes

This is not as simple as it might seem. The obvious candidate would seem to be to have a class where each object would represent one match. This would correspond closely with the information that is currently collected, and Buddy would enter the details of each match. A program could then be written to process a collection of these objects to produce the statistics required.

This is possible, and it would work. But if we examine the requirements more closely it may well not be the best way. The task of finding the total number of runs scored by a duck, for example, would involve examining the scores from each match to see whether or not the duck played and keeping some sort of running total. Again, this is possible and it would work, but it is starting to seem complicated.

A closer look at the requirements shows that all the results that have been asked for are details (attributes, even) of the duck. There is no need to display anything that might be an attribute of a match, such as the name of the winning team, for example. The total number of runs scored could simply be kept as an attribute of a duck, something that seems much neater and simpler.

The program that follows could have been written using either of these approaches. We have chosen one particular route, but the alternative is in many ways equally valid. Object-oriented design is sometimes like this; if there was a class already available to implement one possibility we might well have chosen it!

The class will represent ducks, and we will call it `Duck`. It could, of course, equally well been called `Cricketer` or even `CricketingDuck`; it all comes down to preference as usual, and we tend to prefer short names to save on typing! This is the only class in this problem.

The next stage is to identify the attributes. There are four that we can see immediately from the requirements:

- Name
- Batting average
- Bowling average
- Run rate

We obviously need to store each duck's name so that they can be identified (and we will assume that each duck has a different name). It would be possible to store each of the last three of these values as attributes, but again there is a decision to be made. These are final values, based on others. Suppose that we knew a duck's batting average and had a new score; to update the average

we need to know the duck's total number of runs, a number that we do not currently have.

These three numbers are actually derived from other attributes, and it is these values that we must store. To recap how these values are calculated:

- Batting average is the total number of runs scored divided by the number of innings.[2]
- Bowling average is the total number of runs conceded while bowling divided by the number of wickets taken.[3]
- Run rate is the number of runs conceded divided by the number of overs bowled.

So we need to store the values that are needed to calculate the averages; these are all integers. There is no need to store the averages themselves, of course. They can simply be calculated as and when they are needed. Finally, we need to store the duck's name (a string). Buddy also stores the name of the duck's team. While there is no immediate need for this information, we may as well add it in since it could come in useful in the future. We will assume that ducks do not change teams.

With the attributes identified we need to move on to the methods. This is not too complicated. Obviously each attribute will need a selector and a mutator[4] but appropriate methods will need to be added so that the required program can be implemented. Also, it is probably a good idea to add methods that will become useful in future programs, even if they are not needed in the particular program that is required first.

When identifying the methods it is important to separate them from the functions that will be needed in the programs. A method is attached to a single instance of a class while functions in programs may well process collections of instances. For example, we might well expect that at some point Buddy is going to want to sort the ducks into some order. A sorting function will obviously be needed to do this but this is part of the program and not part of the class; it will process a collection of ducks. The only methods needed for this sorting process will be ones to return the required averages of an individual duck since this is used as the basis for the sort.

Since the averages are clearly of great interest it also makes sense to implement a method to output them all in some neat format. So that this can be used in any program this should not actually do the output itself; rather it should return a suitable string that can be processed in any way required.

Defining the Duck class

With the attributes and methods identified, starting to implement the `Duck` class is straightforward. The attributes will all be declared as private:

```
// Personal Information
private String name;
private String team;
```

---

2 Cricket buffs will realise that we are ignoring the complexities of "not out" innings; averages are usually calculated using only completed innings (that is innings when the player was out).

3 Again, we are going to assume that all overs are completed.

4 It is possible that the programs won't need all of these, but there are sufficiently few attributes that they might as well all have a selector and a mutator.

```
// Bowling Figures
private int overs;
private int wickets;
private int runsConceded;

// Batting Figures
private int innings;
private int runsScored;
```

Now for the methods. There will obviously need to be a constructor, and a decision needs to be made on whether to set any initial values. It seems reasonable to see all the numeric values to 0, but there are no obvious initial values for the strings. The constructor is therefore just a collection of assignment statements:

```
public Duck ()
{
  overs = 0;
  wickets = 0;
  runsConceded = 0;
  innings = 0;
  runsScored = 0;
}
```

The selectors come next. The pattern for these is very familiar by now, so here is just one example:

```
public void setName (String newName)
{
  name = newName;
}
```

The mutators for the attributes are similarly familiar. For example:

```
public int getOvers ()
{
  return overs;
}
```

There will be one selector and one mutator for each of the attributes; they are included in the complete listing at the end of this section.

Now we come to the statistics that we plan to derive from these values. There is a lot of division here, so we will need to be on the look out for attempted division by 0 errors. The values involved are also all integers, so we will also need to take care to convert the resulting averages to floating-point values.

The bowling average is the numbers of runs conceded divided by the number of wickets taken:

```
runsConceded / wickets
```

These values are both integers, so a cast is needed to make sure that the result is returned as a floating-point value. The simplest way to do this is to cast the *runsConceded* value to a `double`:

```
(double) runsConceded / wickets
```

Nearly there. The remaining problem is that an error would now occur if the *value* of wickets was 0. Let's look at the complete method so far to see why

this is a potential problem:

```
public double getBowlingAverage ()

{
  return ( (double) runsConceded) / wickets;
}
```

This method can only return a number. We don't want to start displaying error messages here since we don't know any details of how the programs using the method might be written; it is quite possible that the programs will trap the error before calling the method. The simplest solution, and the one that most programmers would adopt when faced with problems like this, is to return a value that is clearly an error. The program using the method can examine the value returned if it needs to detect the error. Here an average of −1 is clearly an error, so we will return that.[5] The complete method now becomes:

```
public double getBowlingAverage ()
{
  if (wickets == 0) {
    return -1.0;
  }
  else {
    return ( (double) runsConceded) / wickets;
  }
}
```

The other two methods required for the derived values follow exactly the same pattern. They both need the cast, and they both need to handle the possibility of a division by 0. They are included in the complete listing.

Finally, there is the method for output, or rather the method that produces a string that could be output. This does not stop a programmer implementing another way of doing this in a program, of course; the values can still be accessed directly via the selectors. The point of this method is to provide generic output that can be used in many situations.

The precise way in which the values were displayed is not especially important, but the following would produce something reasonably neat:

```
public String getFigures ()
{
  String temp = " ";

  temp += "***** Statistics for " + name
       + " (of team " + team + ") *****\n\n";
  temp += " Total innings batted : " + innings + "\n";
  temp += " Total runs scored : " + runsScored + "\n";
  temp += " Batting average : " + getBattingAverage() + "\n\n";

  temp += " Total overs bowled : " + overs + "\n";
  temp += " Wickets taken : " + wickets + "\n";
  temp += " Runs conceded : " + runsConceded + "\n";
  temp += " Bowling Average : " + getBowlingAverage () + "\n";
  temp += " Run Rate : " + getRunRate () + "\n";

  for (int i = 0; i < name.length (); i ++) {
    temp += "*";
  }
```

5 Actually, we will return −1.0 since a floating-point value is needed.

```
    for (int i = 0; i < team.length (); i ++ ) {
      temp += "*";
    }

    temp += "**************";
    return temp;
  }
```

The two loops in this code are worth a second look. If you think about it they're making sure that the top row (including the duck's name and team name surrounded by stars) is the same length as the bottom.

Finally, you might remember that it is customary for a Java class to have a method called *toString* that does pretty much what this *getFigures* method achieves. We might as well add one in; it can just call *getFigures*:

```
public String toString ()
{
  return getFigures ();
}
```

With the addition of a suitable header block of comments we now have the full implementation of the class:

```
/* Duck.java - a class representing a cricketing duck.

   Author    : AMJ
   Date      : 17th July 2003
   Tested on : Linux (Red Hat 7.3), JDK 1.4.1
*/

public class Duck
{
  // Personal information
  private String name;
  private String team;

  /* Bowling figures - we are bothered about :

     runsConceded / wickets (bowling average)
     runsConceded / overs (run rate)
  */

  private int overs;
  private int wickets;
  private int runsConceded;

  /* Batting figures - we are bothered about :
     runsScored / innings (batting average)
  */

  private int innings;
  private int runsScored;

  public Duck () // initialise all attributes to 0
  {
    overs = 0;
    wickets = 0;
    runsConceded = 0;
    innings = 0;
    runsScored = 0;
  }
```

```
public void setName (String n)
{
  name = n;
}

public void setTeam (String t)
{
  team = t;
}

public void setOvers (int o)
{
  overs = o;
}

public void setWickets (int w)
{
  wickets = w;
}

public void setRunsConceded (int rc)
{
  runsConceded = rc;
}

public void setInnings (int i)
{
  innings = i;
}

public void setRunsScored (int rs)
{
  runsScored = rs;
}

// Standard accessor methods

public String getName ()
{
  return name;
}

public String getTeam ()
{
  return team;
}

public int getOvers ()
{
  return overs;
}

public int getWickets ()
{
  return wickets;
}

public int getRunsConceded ()
{
  return runsConceded;
}

public int getInnings ()
{
  return innings;
}
```

```
public int getRunsScored ()
{
  return runsScored;
}

// Some methods to keep the statisticians happy

public double getBowlingAverage ()
{
  if (wickets == 0) {
    return -1.0;
  }
  else {
    return ( (double) runsConceded) / wickets;
  }
}

public double getRunRate ()
{
  if (overs == 0) {
    return -1.0;
  }
  else {
    return ( (double) runsConceded) / overs;
  }
}

public double getBattingAverage ()
{
  if (innings == 0) {
    return -1.0;
  }
  else {
    return ( (double) runsScored) / innings;
  }
}

public String getFigures ()
{
    String temp = " ";

    temp += "***** Statistics for " + name
         + " (of team " + team + ") *****\n\n";
    temp += " Total innings batted : " + innings + "\n";
    temp += " Total runs scored : " + runsScored + "\n";
    temp += " Batting average : " + getBattingAverage()
         + "\n\n";

    temp += " Total overs bowled : " + overs + "\n";
    temp += " Wickets taken : " + wickets + "\n";
    temp += " Runs conceded : " + runsConceded + "\n";
    temp += " Bowling Average : " + getBowlingAverage ()
         + "\n";
    temp += " Run Rate : " + getRunRate () + "\n";
    for (int i = 0; i < name.length (); i ++ ) {
        temp += "*";
    }

    for (int i = 0; i < team.length (); i ++ ) {
         temp += "*";
    }
```

```
    temp += "****************";
    return temp;
  }

  public String toString ()
  {
    return getFigures ();
  }
}
```

Testing the classes

The next stage in the development is to write a small driver program to test the class. This program should test all the methods (paying particular attention to the methods dealing with the derived values) so that the programmers can be sure that they work correctly. We saw a similar process with Elvis's reminders in the last chapter.

Driver programs may simply call the methods of a class one after another, reporting the results, or they may be more complex. A driver for a sophisticated class would probably provide some sort of menu that could be used, and there is more about driver programs and testing in the next chapter. For the moment, we'll include a short `main` method for some basic testing.

This method will just declare a *`Duck`* object, use the mutators to set some suitable values for the attributes, and then use the *`toString`* method to display the results. The expected results could, of course, be calculated by hand first. The selectors will be tested by confirming the values entered by the user. Since there is now going to be some input, the class will have to include the usual *`Console`* library.

The method is quite simple, but would allow for some basic testing:

```
public static void main (String args[])
{
  Duck aDuck = new Duck ();

  System.out.print ("Name: ");
  aDuck.setName (Console.readString ());
  System.out.println ("Name set to \" " + aDuck.getName ()
                      + "\" ... ");

  System.out.print ("Team: ");
  aDuck.setTeam (Console.readString ());
  System.out.println ("Team set to \" " + aDuck.getTeam ()
                      + "\" ... ");

  System.out.print ("Innings: ");
  aDuck.setInnings (Console.readInt ());
  System.out.println ("Innings set to \" " + aDuck.getInnings ()
                      + "\" ... ");

  System.out.print ("Runs Scored: ");
  aDuck.setRunsScored (Console.readInt ());
  System.out.println ("Runs Scored set to \" "
                      + aDuck.getRunsScored () + "\" ... ");

  System.out.print ("Wickets Taken: ");
  aDuck.setWickets (Console.readInt ());
  System.out.println ("Wickets set to \" " + aDuck.getWickets ()
                      + "\" ... ");
```

```
    System.out.print ("Runs Conceded: ");
    aDuck.setRunsConceded (Console.readInt () );
    System.out.println ("Runs Conceded set to \""
                        + aDuck.getRunsConceded () + "\" ... ");

    System.out.print ("Overs Bowled: ");
    aDuck.setOvers (Console.readInt () );
    System.out.println ("Overs Bowled set to \" " + aDuck.getOvers ()
                        + "\" ... ");

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

}
```

The output from the program is simple as it is intended only for testing, but it does give us some basic confidence that the class works as expected, at least when valid data is provided.

```
tetley% java Duck
Name: Elvis
Name set to "Elvis" ...
Team: Quackshire
Team set to "Quackshire" ...
Innings: 10
Innings set to "10" ...
Runs Scored: 170
Runs Scored set to "170" ...
Wickets Taken: 20
Wickets set to "20" ...
Runs Conceded: 200
Runs Conceded set to "200" ...
Overs Bowled: 10
Overs Bowled set to "10" ...
***** Statistics for Elvis (of team Quackshire)  *****

Total innings batted  : 10
Total runs scored     : 170
Batting average       : 17.0
Total overs bowled    : 10
Wickets taken         : 20
Runs conceded         : 200
Bowling Average       : 10.0
Run Rate              : 20.0
**************************************************
```

This is, of course, only a basic test. It would have to be repeated with a proper test plan before we had any confidence that the class worked as expected.

The program

With the specification and implementation of the *Duck* class in hand, we must now turn our attention to the program which will use instances of this *Duck* class to store data about our cricketing ducks.

As we intend to store data on more than one duck, it makes sense for the program to maintain some form of collection of *Duck* objects. The question remains as to which sort of collection to use, but this is an easy decision. As the user can add and delete ducks from the collection at will, an array would be too inflexible and inefficient for our needs, so an `ArrayList` is the ideal choice.

Examining the program specification in more detail, we can split the program into several smaller problem areas:

- Adding a new duck to the list.
- Removing an existing duck from the list.
- Displaying statistics for an individual duck.
- Displaying statistics for the entire list of ducks.
- Displaying a list of teams.
- Modifying the statistics for a duck in the list.

It is worthwhile developing a method (or several methods) to perform each task independently of the other tasks; this way they can be tested in isolation before adding any complications that may arise from using them together. As it happens we saw methods very similar to those needed here in the examples in the last chapter.

The list can be used by any of the methods, so this will be created in the constructor. The new class is going to amount to a database of Ducks, so *DuckDatabase* is a reasonable name.

```
public class DuckDatabase
{
  private ArrayList ducks;

  public DuckDatabase ()
  {
    ducks = new ArrayList ();
  }
}
```

Adding a new duck to the list

As we have seen in the previous chapter, it is easy to add a new item to an `ArrayList` – the add method takes an object as a parameter, and adds this object to the list, increasing the size of the list by one.

Creation of the list is straightforward:

As each object needs to be initialised first, we need to declare and initialise a *Duck* object, then add this to the `ArrayList`:

```
Duck d = new Duck ();
ducks.add (d);
```

This introduces a problem straight away – the *Duck* object we have created has absolutely no distinguishing attribute values, so there is no (easy) way to keep track of it within the list. As every duck on the reserve has a unique name, it makes sense to use the value of the name attribute as the search key.[6] Therefore we must at least set the value of this attribute before adding the object to the list. The *Duck* class provides mutator methods for all its attributes, so we simply use the one to change the value of the name attribute (*setName*).

Using the *Console* class we can receive user input, so we can prompt the user to enter a suitable name. There is no need to use a temporary variable to

6 That is, the attribute we can search for and be confident of finding no more than one duck with that value for that attribute. This is also known as a primary key when discussed in the field of databases.

store the name as the method call can be used as a parameter:

```
System.out.print ("Enter duck's name : ");
d.setName (Console.readString () );
ducks.add (d);
```

Since every duck should be a member of a team, it would make sense to set the value of the team attribute at this stage as well. This way, it is impossible for any of the *Duck* objects to end up in the list with an empty team attribute. This is achieved similarly to the way we set the duck's name, via the *setTeam* method.

It would be useful to put the functionality we have defined so far into a suitably named method (*addDuck*, say). The method does not require any parameters, as the *Duck* object is created within the method body, and the user is prompted to input the name and team values. In addition, no value need be returned, as the result of calling the method is that the ArrayList is updated.

So, the method will end up looking like this:

```
public void addDuck ()
{
  Duck d = new Duck ();

  System.out.print ("Enter duck's name: ");
  d.setName (Console.readString () );
  System.out.print ("Enter duck's team: ");
  d.setTeam (Console.readString () );

  ducks.add (d);
}
```

Deleting a duck from the list

Again, this is fairly straightforward, as the ArrayList class provides the remove method to delete an element. The only major complication is in finding the *Duck* object we wish to delete.

As we said earlier, every duck on the reserve has a unique name, so we can search the list for a *Duck* object whose name attribute matches that name. The *Duck* class has a *getName* method which returns the value of the name attribute (a String). So, in pseudocode, all we need to do is:

```
FOR EACH DUCK IN THE LIST
   CALL THE getName METHOD
      IF THIS MATCHES THE SEARCH TERM, DELETE THIS DUCK
```

This seems simple – now to implement it!

As we go through the list of ducks, we need to initialise a temporary *Duck* object so we can call the *getName* method. This is done by calling the get method of the ArrayList class, and performing a cast on the object returned (as we saw in the previous chapter).

A final nicety would be to stop searching the list if we have found a match (after all, if there's only going to be one duck called Richie, there's no point searching for another after we've found a match!). This could be achieved in several ways, from breaking forcibly out of the loop, to setting some flag when the match is found and checking for this flag with every iteration. However, in this case the neatest solution would be to simply return from the method when a match is found.

The method needs to take a parameter representing the name of the duck whose entry we want to delete. The return type could be void, as we don't

actually need to return a value. However, it can be useful for a method that might potentially fail in its task to return a `boolean`, denoting whether or not the method performed its task successfully. The code which calls the method can then handle the situation more robustly. In this case, the method should return true if the name was found in the list, and false if not (the calling code could then, for example, print an error message). This all sounds terribly complicated, but it's actually just the same as searching through Elvis's list of jobs in the last chapter.

Here is the complete method:

```
public boolean removeDuck (String name)
{
  Duck tmpDuck; // create once here, not repeatedly inside loop

  for (int i = 0; i<ducks.size (); i++) {
     tmpDuck = (Duck)ducks.get (i);

     if (tmpDuck.getName ().equals (name) ) {
      ducks.remove (i);
      return true;
     }
  }

  return false; // only reached if no match is found
}
```

Displaying statistics for a single duck

Now we need to find the statistics of a particular duck. The *`removeDuck`* method involved searching through the list to find a certain duck, and this method will require a similar approach. The *`Duck`* class provides a *`getFigures`* method, which returns a `String` representation of that duck's bowling and batting figures. In pseudocode, the task is:

```
FOR EACH DUCK IN THE LIST
    CALL THE getName METHOD
      IF THIS MATCHES THE SEARCH TERM, PRINT DUCK'S STATISTICS
```

We can re-use the *`removeDuck`* method for much of the code for our *`getFigures`* method. The code to search the list is exactly the same, the only change is in the method we call when we find a match as this time we will call the *`getFigures`* method of the *`Duck`* class.

```
public boolean getFigures (String name)
{
   Duck tmpDuck; // create once here, not repeatedly inside loop

   for (int i = 0; i<ducks.size (); i++) {
     tmpDuck = (Duck)ducks.get (i);

     if (tmpDuck.getName ().equals (name) ) {
         System.out.print (tmpDuck.getFigures () );
         return true;
     }
   }

   return false; // only reached if no match is found
}
```

Displaying statistics for the entire list of ducks

And now for all the ducks and finding the best cricketers. This method is slightly more complicated than either of the previous two in that we have no search term to match. Instead, we need to traverse the entire list and decide what the best figure is for each of the performance attributes (runs scored, runs conceded, wickets taken, run rate, and bowling average). In addition to recording these figures, we will also need to record the name of the duck holding each of the best figures found so far.

We will need a `for` loop and temporary *`Duck`* object as before. We will also need variables to hold the current best figures and the names of the corresponding ducks. Finally, we will need to print out the figures and return from the method.

Now, we will need to establish how to work out which is the best figure for, say, runs scored. In pseudocode, we could do:

```
highestRuns = -1 // DELIBERATELY LOW VALUE
  FOR EACH DUCK IN LIST
   IF DUCK'S TOTAL RUNS SCORED IS HIGHER THAN highestRuns
    SET highestRuns TO DUCK'S TOTAL RUNS SCORED
```

We start by setting a variable (*`highestRuns`*) to a very low value. We then loop through all the ducks and whenever a higher figure is encountered, the variable is increased to match this new highest value. When this loop terminates, the *`highestRuns`* variable is guaranteed to hold the best figure.

In actual fact, we also need to record the name of the proud batsduck:

```
highestRuns = -1
batsduck = " "
FOR EACH DUCK IN LIST
  IF DUCK'S TOTAL RUNS SCORED IS HIGHER THAN highestRuns
    SET highestRuns TO DUCK'S TOTAL RUNS SCORED
    SET batsduck TO DUCK'S NAME
```

This procedure can be repeated separately for each of the figures required. However, for efficiency reasons, it would make sense to perform all the calculations within the same `for` loop, to avoid having to go through the list for each statistic.

We cycle through the loop in exactly the same way as before, calling the `get` method of the `ArrayList` class and casting the returned object to type *`Duck`*. We then compare each of the duck's figures in turn with the current "bests", updating the figures and names where necessary.

This time the method does not need to return any value, as it will print its results to the screen. Also, it need not take any arguments, as the name of the list to be searched is known in advance. So the method will look like this:

```
public void getStatistics ()
{
  int runsScored = 0, wickets = 0;
  double battingAvg = 0.0, bowlingAvg = 0.0
  double runRate = 99999999.0; // we want the *lowest* here

  String runsScoredName = " ", wicketsName = " ",
         battingAvgName = " ", bowlingAvgName = " ",
         runRateName = " ";
```

```
    Duck tmpDuck;

    for (int i = 0; i<ducks.size (); i++) {
      tmpDuck = (Duck)ducks.get (i);

      if (tmpDuck.getRunsScored () > runsScored) {
          runsScored = tmpDuck.getRunsScored ();
          runsScoredName = tmpDuck.getName ();
      }
      if (tmpDuck.getWickets () > wickets) {
          wickets = tmpDuck.getWickets ();
          wicketsName = tmpDuck.getName ();
      }
      if (tmpDuck.getBattingAverage () > battingAvg) {
          battingAvg = tmpDuck.getBattingAverage ();
          battingAvgName = tmpDuck.getName ();
      }
      if (tmpDuck.getBowlingAverage () > bowlingAvg) {
          bowlingAvg = tmpDuck.getBowlingAverage ();
          bowlingAvgName = tmpDuck.getName ();
      }
      if (tmpDuck.getRunRate () < runRate) {
          runRate = tmpDuck.getRunRate ();
          runRateName = tmpDuck.getName ();
      }
    }

    System.out.println ("********");
    System.out.println (" Most runs scored : " + runsScoredName
                        + "  (" + runsScored + ") ");
    System.out.println (" Best batting average : " + battingAvgName
                        + "  (" + battingAvg + ") ");
    System.out.println (" ");

    System.out.println (" Most wickets taken : " + wicketsName
                        + "  (" + wickets + ") ");
    System.out.println (" Best bowling average : " + bowlingAvgName
                        + "  (" + bowlingAvg + ")");
    System.out.println (" Best run rate : " + runRateName
                        + "  (" + runRate + ") ");
    System.out.println ("********\n");

  }
```

Displaying a list of teams

The requirement for this method is that we obtain a list of unique team names. Again, we will need to go through (properly called *iterate)* the `ArrayList`, this time checking the team attributes, and storing a new name every time one is encountered.

Iterating through the list should need no explanation by now! Calling the *`getTeam`* method of the *`Duck`* class and checking the return value is a process found in most of the methods so far, and is indeed an example of a very common process. The complication this time comes in how we store the return value. It would be grossly inefficient to create a large number of `String` variables to store the team names (not to mention that this method is prone to errors – you can create 1000 variables, but it's almost guaranteed that one day, someone somewhere will want to store 1001 names!). There is no reliable way

we can know in advance how many teams there are,[7] so this situation calls for a collection. Not just any old collection, though – a variable-length collection. You may have realised that this means we need another `ArrayList`!

The pseudocode for this method would be something like:

```
CREATE EMPTY LIST list
FOR EACH DUCK IN DUCK LIST
  IF DUCK'S TEAM IS NOT PRESENT IN list
    ADD DUCK'S TEAM TO LIST
FOR EACH TEAM IN list
  PRINT TEAM
```

The `ArrayList` for the teams is created in the same way as the one for *`Duck`*s:

```
ArrayList teams = new ArrayList ();
```

Elements are added in the same way as in the *`addDuck`* method:

```
teams.add (aTeamName);
```

We also need to check whether the team list already contains a certain name. Thankfully, we met the `contains` method of the *`ArrayList`* class in the last chapter:

```
if (teams.contains (anotherName) ) {
  // do something
}
```

This method need not take any arguments, as again the list it operates on is known beforehand. Again the return type is `void`, as the method will display its results. In a different scenario, it could well return a `String` (or an array or `ArrayList` of `String`s) containing the team names.

Tying all this together, we have the *`printTeams`* method:

```
public void printTeams ()
{
  ArrayList teams = new ArrayList ();
  Duck tmpDuck;

  for (int i = 0; i<ducks.size (); i++) {
    tmpDuck = (Duck)ducks.get (i); // second nature by now?

    if (!teams.contains (tmpDuck.getTeam () ) ) {
       // add new team only if not present already
       teams.add (tmpDuck.getTeam () );
    }
  }

  if (teams.size () == 0) {
    System.out.println ("No teams found.");
  }
  else {
    System.out.println ("*********************************");
    System.out.println ("The following teams were found :\n");
```

7 Actually, it is possible to know the upper bound for this figure – a group of n cricketing ducks represents, at most, n teams.

```
    for (int i = 0; i<teams.size (); i++) {
      System.out.println ( (String)teams.get (i) );
    }

    System.out.println ("*********************************");
  }
}
```

Modifying the statistics for a duck in the list

The *Duck* class contains seven modifiable attributes – name, team, runs scored, innings batted, overs bowled, runs conceded, and wickets taken. It is unlikely that a duck will ever change its name, so we'll discard the need for our program to handle this. If we allow that a duck might some day change its team, this still leaves us with six ways to modify a *Duck* object.

Now, we could write individual methods to modify each of the attributes, but in many ways it is just as easy to write a single *modifyDuck* method to interface with the mutator methods of the *Duck* class.

Once again, we will need to iterate over the list of *Duck*s and attempt to match a search string, so this code at least can be copied from one of the earlier methods. When the correct duck is found, we need to establish which attribute to update, and what its updated value is. This should be simple enough as we have already seen simple menu systems, and it would not be too much effort to implement one for our purposes here:

```
PRINT MENU AND PROMPT USER FOR CHOICE
IF FIRST OPTION IS CHOSEN
  <do something>
ELSE IF SECOND CHOSEN
  <do something else>
.
.
ELSE QUIT
```

We can use `System.out.print` (or `println`) for outputting the menu. It is very easy to be creative with the output, in order to produce a neat, aesthetically pleasing interface and we will do so with asterisks to create a box for the menu.

To read the user input, we will use the *Console* class again. This time, it would make sense to use the *readChar* method, as we would expect such a menu to prompt by initial letter (for example, "*press W to update figure for wickets*"). Since `char` is an ordinal variable type – it can hold one of a finite number of possible values – we can use a single `switch` statement to control the menu logic:

```
char choice = Console.readChar ();

switch (choice) {
  case 'w':
  case 'W':
           // update wickets
           break;

  case 'r':
  case 'R':
          // update runs scored
          break;

  // And so on
}
```

At first glance, it is tempting to think that each option on the menu leads to nothing more than "*enter a new value for X*", followed by updating attribute X. However, this does not account for any error checking at all. It is almost trivial to check for common errors, such as the new figure being impossible (the number of runs scored can never decrease, for example). Also, if the statistic is undefined, the method will return −1 and while we could handle this case separately, we will leave this as an exercise at the end of the chapter!

So, taking the runs scored as an example, we might expect to see:

```
case 'r':
case 'R':
          System.out.println ("Current figure is"
                            + duck.getRunsScored () + ".");
          System.out.print ("Enter new figure : ");
          int tmpInt = Console.readInt ();
          if(tmpInt <= duck.getRunsScored () ) {
             duck.setRunsScored (tmpInt);
          }
          else {
            System.err.println ("ERROR - new figure lower than"
                               + "old one!");
          }
          break;
```

We would expect a similar block of code for each menu option (to save space, we'll leave them until the end of the section!). This would ideally be included inside a loop which prompted repeatedly until told to quit (since more often than not, we will want to update multiple attributes for a single duck). The loop would execute at least once, so a `do...while` loop is the correct choice:

```
do {
    // print menu
    // switch statement
} while (quit option not chosen);
```

As with all the other methods involving a search for a specific duck, this method returns a `boolean` value – true if the duck is found, false otherwise – and takes the name to search for as an argument.

Another touch to add here would be to send error messages to a different location than ordinary output. This would be useful if, for example, you wanted to print normal output to the screen as usual, but record error messages in a file. The syntax to do this is incredibly similar to what we have already seen; instead of:

```
System.out.print (message);
```

we use

```
System.err.print (message);
```

By default, `System.out` (the standard output stream) and `System.err` (the error output stream) both display their output on the command line, but in more advanced Java programs (well beyond the scope of this book!), it is possible to redirect one or both. Therefore splitting the output in this way is a good habit to get into.

The final code for this method is:

```
public boolean modifyDuck (String name)
{
  Duck tmpDuck = null;
  char choice = ' ';
  int tmpInt = -1;
  boolean foundDuck = false;
  for (int i = 0; i<ducks.size(); i++) {
    tmpDuck = (Duck)ducks.get(i);

    if (tmpDuck.getName ().equals (name) ) {
        foundDuck = true;
        break;
    }
  }

  if (!foundDuck) {
    return false;
  }

  do {
      System.out.println (" ");
      System.out.println ("****************************");
      System.out.println ("* T - team                 *");
      System.out.println ("* *");
      System.out.println ("* S - runs scored          *");
      System.out.println ("* I - innings batted       *");
      System.out.println ("*                          *");
      System.out.println ("* C - runs conceded        *");
      System.out.println ("* O - overs bowled         *");
      System.out.println ("* W - wickets taken        *");
      System.out.println ("*                          *");
      System.out.println ("* Q - return to main menu *");
      System.out.println ("****************************");
      System.out.println (" ");
      System.out.print ("Enter choice : ");
      choice = Console.readChar ();

  switch (choice) {
  case 'c':
  case 'C':
          System.out.print ("Current figure is "
                          + tmpDuck.getRunsConceded ()
                          + ". Enter new figure : ");
          tmpInt = Console.readInt ();

          if (tmpInt > tmpDuck.getRunsConceded () ) {
              tmpDuck.setRunsConceded (tmpInt);
          }
          else {
              System.err.println ("ERROR - new figure lower "
                                  + "than old one!");
          }
          break;

  case 'i':
  case 'I':
           System.out.print ("Current figure is "
                           + tmpDuck.getInnings ()
                           + ". Enter new figure : ");
           tmpInt = Console.readInt ();
```

```
          if (tmpInt > tmpDuck.getInnings () ) {
            tmpDuck.setInnings (tmpInt);
          }
          else {
            System.err.println ("ERROR – new figure lower "
                                + "than old one!");
          }
          break;

  case 'o':
  case 'O':
          System.out.print ("Current figure is "
                            + tmpDuck.getOvers ()
                            + ". Enter new figure : ");
          tmpInt = Console.readInt ();
          if (tmpInt > tmpDuck.getOvers () ) {
            tmpDuck.setOvers (tmpInt);
          }
          else {
            System.err.println ("ERROR – new figure lower "
                              + "than old one!");
          }
          break;

  case 's':
  case 'S':
          System.out.print ("Current figure is "
                            + tmpDuck.getRunsScored ()
                            + ". Enter new figure : ");
          tmpInt = Console.readInt ();
          if (tmpInt > tmpDuck.getRunsScored () ) {
            tmpDuck.setRunsScored (tmpInt);
          }
          else {
            System.err.println ("ERROR – new figure lower "
                                + "than old one!");
          }
          break;

  case 't':
  case 'T':
          System.out.print ("Current team is "
                            + tmpDuck.getTeam ()
                            + ". Enter new team : ");
          String tmpString = Console.readString ();
          if (!tmpString.equals(tmpDuck.getTeam () ) ) {
            tmpDuck.setTeam (tmpString);
          }
          else {
            System.err.println ("ERROR – no change!");
          }
          break;

  case 'w':
  case 'W':
          System.out.print ("Current figure is "
                            + tmpDuck.getWickets()
                            + ". Enter new figure : ");
          tmpInt = Console.readInt ();
```

```
            if (tmpInt > tmpDuck.getWickets ()) {
              tmpDuck.setWickets (tmpInt);
            }
            else {
              System.err.println ("ERROR - new figure lower "
                                  + "than old one!");
            }
            break;

    case 'q':
    case 'Q':
            break;

    default:
            System.err.println (choice + " was not an option!");
            break;
    }
  } while (choice != 'q' && choice != 'Q');

  return true; // only executed if duck was found initially
}
```

The checks in this code are all quite simple, but it is noticeable that the great majority of the program is concerned just with checking the values that have been entered.

Bolting it all together

Now we have a collection of methods which will manipulate our list of ducks in a variety of ways, according to the program specification. However, our code so far is just that – a collection of methods. We still do not have a working program that solves a problem.

The first thing to determine with any program is how to handle its command line arguments, but this is not a problem here, as the program relies solely on interactivity. As this is the case, we need to think about user-friendliness, and that means designing another menu system to guide the user around our program.

We need a list of options; one for each of the tasks we've discussed, plus the ubiquitous "*press Q to quit*". We also need to incorporate some kind of formatted output, to make the menu system easier on the eye. In fact, it is easy enough to use asterisks as we did for the previous menu. The architecture of this menu system will be almost identical to the one for *modifyDuck* with the only changes being in the output, and the choice of method for each option. We still have a variable to store the user's choice of option, and a `switch` statement to control the program according to this option.

We could simply implement this menu system in the `main` method of the program. However, this causes problems if we ever wish to use this functionality as part of another, larger program (say, if a well-known sports broadcaster decided to show coverage of the Waterfowl Cricket Cup, and wanted to use such a database alongside the rest of their fancy graphics). Therefore we will implement this "main" menu in a separate method (named *run*), and simply call this method from within `main`.

```
public void run ()
{
  char choice = ' ';
  String input = " "; // more efficient to declare this once
```

```
do {
    System.out.println (" ");
    System.out.println ("******* MAIN MENU *********");
    System.out.println ("*                          *");
    System.out.println ("* A - Add Duck             *");
    System.out.println ("* D - Delete Duck          *");
    System.out.println ("*                          *");
    System.out.println ("* M - Modify Duck          *");
    System.out.println ("* F - Duck Figures         *");
    System.out.println ("*                          *");
    System.out.println ("* S - Overall Statistics   *");
    System.out.println ("* T - Print teams          *");
    System.out.println ("*                          *");
    System.out.println ("* Q - Quit                 *");
    System.out.println ("****************************");
    System.out.println (" ");
    System.out.print ("Enter choice : ");
    choice = Console.readChar ();

    switch (choice) {
            case 'q':
            case 'Q':
                    break;

            case 'a':
            case 'A':
                    addDuck ();
                    break;

            case 'd':
            case 'D':
                    System.out.print ("Enter duck's name : ");
                    input = Console.readString ();

                    if (!removeDuck (input) ) {
                      System.err.println ("*** ERROR - No such
                                  duck" + " was found! ***\n");
                    }
                    break;

            case 'm':
            case 'M':
                    System.out.print ("Enter duck's name : ");
                    input = Console.readString ();

                    if (!modifyDuck (input) ) {
                      System.err.println ("*** ERROR - No such
                                  duck" + " was found! ***\n");
                    }
                    break;

            case 'f':
            case 'F':
                    System.out.print ("Enter duck's name : ");
                    input = Console.readString ();

                    if (!getFigures (input) ) {
                      System.err.println ("*** ERROR - No such
                                  duck" + " was found! ***\n");
                    }
                    break;
```

```
                case 's':
                case 'S':
                        getStatistics ();
                        break;

                case 't':
                case 'T':
                        printTeams ();
                        break;

                default:
                        System.out.println ("*** ERROR : \' " +
                          choice + "\' was not an option! ***\n");
                        break;

      }

    } while (choice != 'q' && choice != 'Q');

}
```

The entire program

Finally, we are in a position to list the entire program. Assuming you have the *htpuj* package and the *Duck* class from earlier in the chapter in your classpath (see your *Local Guide* to establish how to do this), you should be able to compile and run this program, and then add, delete, manipulate, and query duck cricketing figures to your heart's content!

```
/* DuckDatabase.java - A fully-featured database for
                       maintaining cricket statistics.

   Author    : GPH
   Date      : 5th July 2003
   Tested on : Linux (Red Hat 7.3), JDK 1.4.1
*/

import htpuj.Console;
import java.util.ArrayList;

public class DuckDatabase
{
  private ArrayList ducks;

  public DuckDatabase ()
  {
    ducks = new ArrayList ();
  }

  public void addDuck ()
  {
    Duck d = new Duck ();

    System.out.print ("Enter duck's name : ");
    d.setName (Console.readString () );
    System.out.print ("Enter duck's team : ");
    d.setTeam (Console.readString () );

    ducks.add (d);
  }

  public boolean modifyDuck (String name)
  {
    Duck tmpDuck = null;
    char choice = ' ';
    int tmpInt = -1;
```

```
    boolean foundDuck = false;
    for (int i = 0; i<ducks.size(); i++) {
      tmpDuck = (Duck)ducks.get(i);
      if (tmpDuck.getName ().equals (name) ) {
        foundDuck = true;
        break;
      }
    }
    if (!foundDuck) {
      return false;
  }

do {
    System.out.println (" ");
    System.out.println ("Updating statistics for "
                      + tmpDuck.getName() );
    System.out.println (" ");
    System.out.println ("****************************");
    System.out.println ("* T - team                 *");
    System.out.println ("*                          *");
    System.out.println ("* S - runs scored          *");
    System.out.println ("* I - innings batted       *");
    System.out.println ("*                          *");
    System.out.println ("* C - runs conceded        *");
    System.out.println ("* O - overs bowled         *");
    System.out.println ("* W - wickets taken        *");
    System.out.println ("*                          *");
    System.out.println ("* Q - return to main menu *");
    System.out.println ("****************************");
    System.out.println (" ");
    System.out.print ("Enter choice : ");
    choice = Console.readChar ();
    switch (choice) {
      case 'c':
      case 'C':
               System.out.print ("Current figure is "
                                 + tmpDuck.getRunsConceded ()
                                 + ". Enter new figure : ");
               tmpInt = Console.readInt ();
               if (tmpInt > tmpDuck.getRunsConceded () ) {
                 tmpDuck.setRunsConceded (tmpInt);
               }
               else {
                 System.err.println ("ERROR - figure lower"
                                     + "than old one!");
               }
               break;
      case 'i':
      case 'I':
               System.out.print ("Current figure is "
                                 + tmpDuck.getInnings ()
                                 + ". Enter new figure : ");
               tmpInt = Console.readInt ();
               if (tmpInt > tmpDuck.getInnings () ) {
                 tmpDuck.setInnings (tmpInt);
               }
```

```
            else {
              System.err.println ("ERROR - figure lower "
                                  + "than old one!");
            }
            break;
        case 'o':
        case 'O':
            System.out.print ("Current figure is "
                              + tmpDuck.getOvers ()
                              + ". Enter new figure : ");
            tmpInt = Console.readInt ();
            if (tmpInt > tmpDuck.getOvers () ) {
              tmpDuck.setOvers (tmpInt);
            }
            else {
              System.err.println ("ERROR - figure lower "
                                  + "than old one!");
            }
            break;
        case 's':
        case 'S':
            System.out.print ("Current figure is "
                              + tmpDuck.getRunsScored ()
                              + ". Enter new figure : ");
            tmpInt = Console.readInt ();
            if (tmpInt > tmpDuck.getRunsScored () ) {
              tmpDuck.setRunsScored (tmpInt);
            }
            else {
              System.err.println ("ERROR - figure lower "
                                  + "than old one!");
            }
            break;
        case 't':
        case 'T':
            System.out.print ("Current team is "
                              + tmpDuck.getTeam ()
                              + ". Enter new team : ");
            String tmpString = Console.readString ();
            if (!tmpString.equals(tmpDuck.getTeam () ) ) {
              tmpDuck.setTeam (tmpString);
            }
            else {
              System.err.println ("ERROR - no change!");
            }
            break;
        case 'w':
        case 'W':
            System.out.print ("Current figure is "
                              + tmpDuck.getWickets()
                              + ". Enter new figure : ");
            tmpInt = Console.readInt ();
            if (tmpInt > tmpDuck.getWickets () ) {
              tmpDuck.setWickets (tmpInt);
            }
```

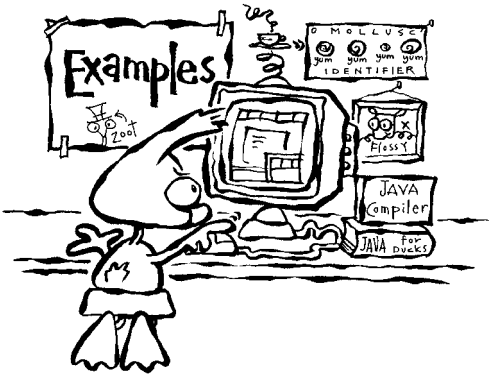
```
            else {
              System.err.println ("ERROR – figure lower "
                                  + "than old one!");
            }
            break;
    case 'q':
    case 'Q':
            break;
    default:
       System.err.println (choice + " not an option!");
       break;
    }
  } while (choice != 'q'  choice != 'Q');
  return true; // only executed if duck was found initially
}
public boolean removeDuck (String name)
{
  Duck tmpDuck;
for (int i = 0; i<ducks.size (); i++) {
  tmpDuck = (Duck)ducks.get (i);
  if (tmpDuck.getName().equalsIgnoreCase (name) ) {
    ducks.remove(i);
    return true;
  }
}
  return false;
}
public boolean getFigures (String name)
{
Duck tmpDuck;
for (int i = 0; i<ducks.size (); i++) {
   tmpDuck = (Duck)ducks.get (i);
   if(tmpDuck.getName ().equalsIgnoreCase (name) ) {
      System.out.println (tmpDuck.getFigures () );
      return true;
    }
}
  return false;
}
public void printTeams ()
{
  ArrayList teams = new ArrayList ();
  Duck tmpDuck;
  // search ducks list, adding any new teams to teams list
  for (int i = 0; i<ducks.size (); i++) {
    tmpDuck = (Duck)ducks.get (i);
    if (!teams.contains (tmpDuck.getTeam () ) ) {
    teams.add (tmpDuck.getTeam () );
  }
}
if (teams.size () == 0) {
   System.out.println ("No teams found.");
}
```

```
    else {
      System.out.println ("**********************************");
      System.out.println ("The following teams were found :\n");
      for (int i=0; i < teams.size (); i++) {
          System.out.println ( (String)teams.get (i) );
      }
      System.out.println ("**********************************");
     }
    }

    public void getStatistics ()
    {
     int runsScored = 0, wickets = 0;
     double battingAvg = 0.0, bowlingAvg = 0.0,
            runRate = 99999999.0;
     String runsScoredName = " ", wicketsName = " ",
            battingAvgName = " ",
            bowlingAvgName = " ", runRateName = " ";
     Duck tmpDuck;
     for (int i = 0; i<ducks.size (); i++) {
       tmpDuck = (Duck)ducks.get (i);
       if (tmpDuck.getRunsScored () > runsScored) {
            runsScored = tmpDuck.getRunsScored ();
            runsScoredName = tmpDuck.getName ();
       }
       if (tmpDuck.getWickets () > wickets) {
           wickets = tmpDuck.getWickets ();
           wicketsName = tmpDuck.getName ();
       }
       if (tmpDuck.getBattingAverage () > battingAvg) {
           battingAvg = tmpDuck.getBattingAverage ();
           battingAvgName = tmpDuck.getName ();
       }
       if (tmpDuck.getBowlingAverage () > bowlingAvg) {
           bowlingAvg = tmpDuck.getBowlingAverage ();
           bowlingAvgName = tmpDuck.getName ();
       }
       if (tmpDuck.getRunRate () < runRate) {
           runRate = tmpDuck.getRunRate ();
           runRateName = tmpDuck.getName ();
       }
    }

     System.out.println ("********");
     System.out.println (" Most runs scored : " + runsScoredName
                         + " (" + runsScored + ")");
     System.out.println (" Best batting average : " + battingAvgName
                         + " (" + battingAvg + ")");
     System.out.println (" ");

     System.out.println (" Most wickets taken : " + wicketsName
                         + " (" + wickets + ")");
     System.out.println (" Best bowling average : " + bowlingAvgName
                         + " (" + bowlingAvg + ")");
     System.out.println (" Best run rate : " + runRateName
                         + " (" + runRate + ")");
     System.out.println ("********\n");
  }
```

```
public void run ()
{
    char choice = ' ';
    String input = " "; // more efficient to declare this once

    do {
        System.out.println (" ");
        System.out.println ("******** MAIN MENU *********");
        System.out.println ("*                          *");
        System.out.println ("* A - Add Duck             *");
        System.out.println ("* D - Delete Duck          *");
        System.out.println ("*                          *");
        System.out.println ("* M - Modify Duck          *");
        System.out.println ("* F - Duck Figures         *");
        System.out.println ("*                          *");
        System.out.println ("* S - Overall Statistics   *");
        System.out.println ("* T - Print teams          *");
        System.out.println ("*                          *");
        System.out.println ("* Q - Quit                 *");
        System.out.println ("****************************");
        System.out.println (" ");
        System.out.print ("Enter choice : ");
        choice = Console.readChar ();

        switch (choice) {
          case 'q':
          case 'Q':
                   break;

          case 'a':
          case 'A':
                   addDuck ();
                   break;

          case 'd':
          case 'D':
                   System.out.print ("Enter duck's name : ");
                   input = Console.readString ();

                   if (!removeDuck (input) ) {
                     System.err.println ("*** ERROR - No such duck"
                                  + " was found! ***\n");
                   }
                   break;

          case 'm':
          case 'M':
                   System.out.print ("Enter duck's name : ");
                   input = Console.readString ();

                   if (!modifyDuck (input) ) {
                     System.err.println ("*** ERROR - No such duck"
                                 + " was found! ***\n");
                   }
                   break;

          case 'f':
          case 'F':
                   System.out.print ("Enter duck's name : ");
                   input = Console.readString ();

                   if (!getFigures (input) ) {
                     System.err.println ("*** ERROR - No such duck"
                                  + " was found! ***\n");
                   }
                   break;
```

```
            case 's':
            case 'S':
                    getStatistics ();
                    break;

            case 't':
            case 'T':
                    printTeams ();
                    break;

            default:
                    System.out.println ("*** ERROR : " + choice
                                + " was not an option! ***\n");
                    break;
        }
      } while (choice != 'q' && choice != 'Q');
    }

    public static void main (String args[])
    {
      // minimalist main method, but this style is not uncommon
      DuckDatabase dd = new DuckDatabase ();

      dd.run ();
    }
  }
```

This is by a long way the longest and most complicated program in this book. Don't be daunted by it; take some time to go through it so that you can see how it works. The chapter showed you how it was built up in small easy-to-handle stages. By far the easiest way to understand how the program works is, of course, to get hold of it and to try it out. Perhaps you can find a mistake!



There are no examples this time; the programs in this chapter have provided quite enough. They have illustrated how a complete program is built up gradually by adding functions and how functions can often be used in many different programs.

18.1 Our definition of batting average is actually inaccurate. The average is really calculated as the number of runs scored divided by the number of *completed* innings. Add an attribute to the `Duck` class to store the number of times an innings has not been completed; call it `notOuts`. Now rewrite the class so that the average is calculated correctly as:

```
runsScored / (innings - notOuts)
```

18.2 As we discussed earlier, in the `getBattingAverage`, `getBowlingAverage`, and `getRunRate` methods, if a value is undefined, the method will return −1. Currently the `DuckDatabase` program does not handle this error, and will quite happily tell you that a duck has a batting average of −1.0. Modify the program so that this condition is handled neatly. You may want to replace the −1.0 with something more informative ("*undefined*", say), or even the entire line of output ("*Duck A has not batted*", for example).

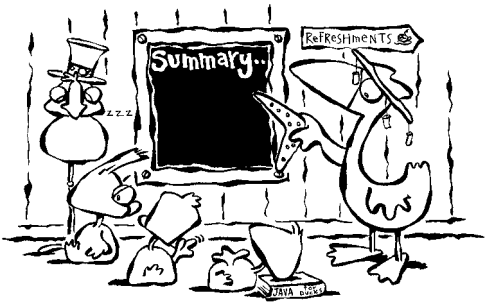
18.3 Write a program that reads the attributes of a complete team of 11 ducks and displays the list sorted by the number of runs scored.

18.4 Modify your program to sort the ducks based on their batting average.

18.5 Currently the `getStatistics` method displays only the best figure in each category. Modify the method so that it displays the top three (handling the cases where fewer than three ducks are present, or that fewer than three sets of figures are available). This exercise will involve searching and sorting, but since it deals with a definite number of figures (three), arrays (rather than `ArrayLists`) can be used.

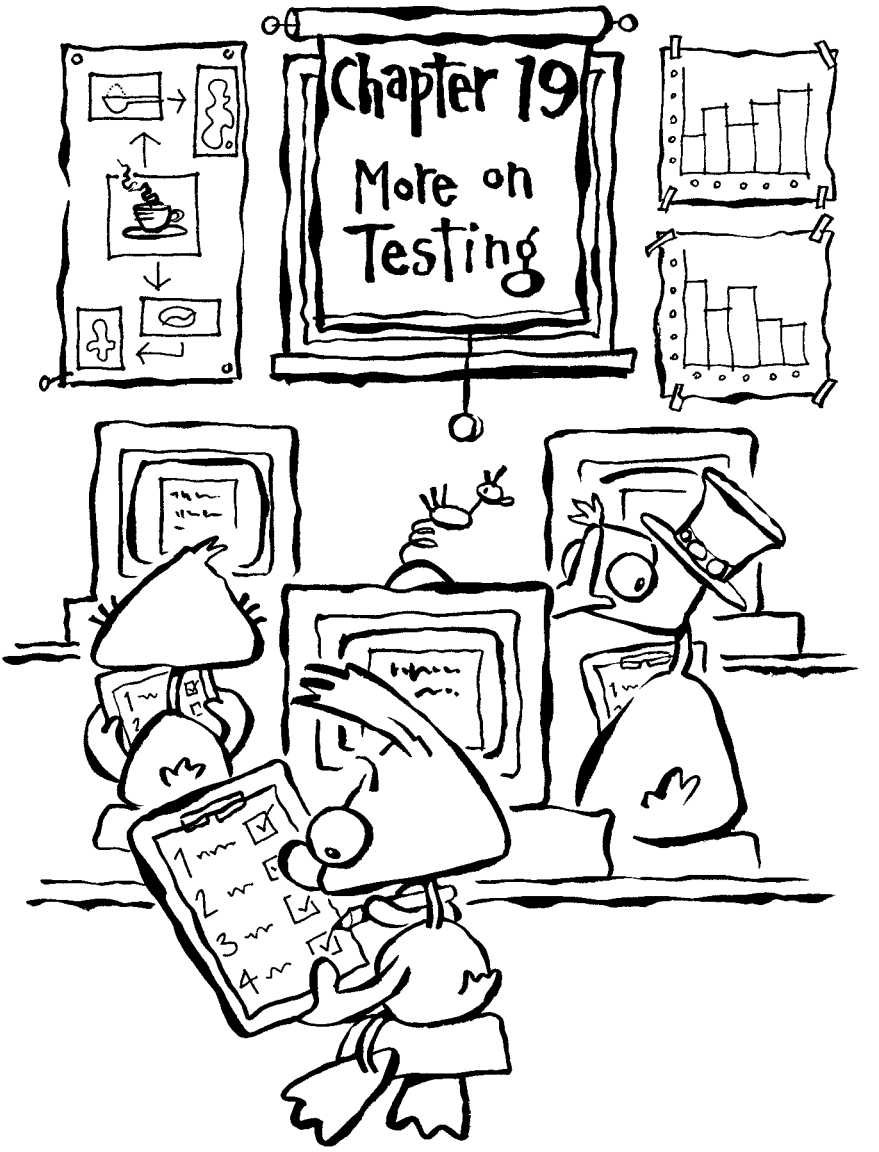
18.6 This chapter included a very basic program for testing the class. Write a more thorough menu-based program that could be used to test the `Duck` class.

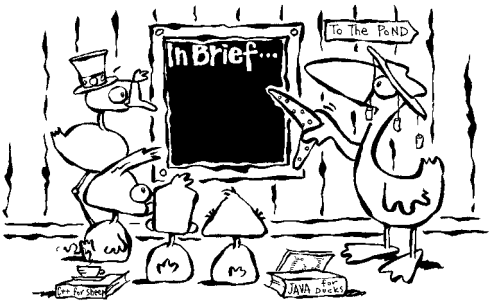
18.7 All the averages in this chapter are traditionally displayed to only two decimal places. It is likely that your Java system is displaying many more places of decimals; investigate the facilities in Java to do this.



When developing a complex Java program it is essential to develop the classes that the program will use first. This does not mean that the programs themselves cannot be designed and perhaps even some of its functions started, but no serious work should be done until the classes are complete. The golden rule when developing a complex program is to develop and change one thing at a time; this way, if something goes wrong the cause can only be in one place.

Of course complex programs need to be tested, as do classes. Before the end of the book the next chapter provides some final advice on how to go about testing programs.


Chapter 19
More on
Testing



This is very nearly the end of the book. You have now seen all the Java that you are going to see in detail. You should by now be able to write quite complex Java programs making use of many of the features available in the current versions of Java. You should be able to analyse a problem and then go on to design and develop the classes and programs that will form the basis of a solution.

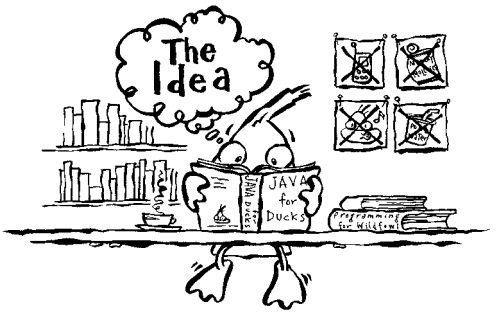
Of course, no solution is of the slightest use unless it works correctly. Many, many pages ago, Chapter 10 explained the importance of testing and showed you how to write a *test plan* for each of your programs. Hopefully you've been doing this as you've written all your programs since then! There is truly no point whatsoever in spending time on developing a program unless you also take the time to test it thoroughly. An untested program is a worthless program and is potentially a very dangerous program.

Since Chapter 10, the programs you have been working on have become longer and more and more complicated. This chapter briefly explains some ideas on how to test larger programs. The processes for testing larger programs are much the same as in Chapter 10, so you might want to go back and quickly read through it again. Larger programs bring with them the need for a very structured approach to testing; unstructured and haphazard testing can be worse than none at all.

Even with all this brave talk of testing it is important to remember that it is never possible to prove that a program works correctly in every case. Testing can only build up confidence in a program; it can never provide complete confidence. It is rarely, if ever, possible to test a program with all the input values that it will have to process. Nevertheless, the ideas in this chapter should mean that you are able to have a reasonably high level of confidence in your programs.

After reading this chapter you should be able to systematically test large programs and classes. You should be able to write a driver program for a class, and you should be able to use it to test a class. You should understand the difference between *black box* and *white box* testing, and you should understand how to decide which to use for a particular test. You should also be able to *dry run* a program.

Your programs should have a much better chance of working correctly!



Testing a program thoroughly is just as important as writing the program in the first place. In many ways it is in fact rather more important as an untested program may contain many costly errors. Since the last chapter on testing your programs have made use of more and more features of Java and have become more and more complicated. It is crucial that you always test these complex programs very thoroughly.

This chapter is not intended to provide a complete description of the correct way to test programs. Whole books have been written on testing and a complete description is way beyond the scope of any book on programming. Instead, this chapter gives you some ideas of ways in which you can test your programs and tries to convince you of the importance of testing.

You are already familiar with the idea of a *test plan* made up of a number of *test cases*. This idea translates directly to longer programs; the only thing that needs to be ensured is that the test cases in the plan cover every possible way in which a program can execute, or every possible route through a program. Your programs now include conditional statements and loops and so there are now many different ways in which they can execute; there must be enough test cases to test each of these possible execution routes thoroughly.

The test plans that you have written so far have relied on specifying a set of input values and the expected output values. The programs have been tested with each case and if the actual output matches the expected output in each case the program has passed the tests. The approach is fine for simple and short programs but it is rarely sufficient for more complex programs. A more thorough approach to testing is needed.

Your programs now also make use of classes and these also have to be tested. A class is a building-block that is potentially useful in many programs. Each class that a program makes use of must be properly tested before it is used; there is very little point in writing and testing a program that relies on a class that doesn't work properly. This chapter ends by explaining how to test a class before it is ever used in a program.

Routes through a program

By now your programs will probably be quite complex and there will be many possible routes through. Conditional statements and loops control the possible routes and each provides several possible ways for a program to execute. A test plan must make sure that every possible route through a program is properly tested. This means first of all that every possibility from each conditional statement must be tested, as must every possible outcome of each loop.

When a user is presented with a prompt to enter a value into a program there is very little that limits what they can provide. Many failures in programs are caused by users managing to provide completely unforeseen input values and so managing to take completely unexpected routes through the program. It is absolutely vital that every possible route through a program is thoroughly tested, even if it seems very unlikely that it will ever be used.

First, each possibility in a conditional statement must be tested. For example, in a program including this conditional statement:

```
int aNumber;

if (aNumber > 1 && aNumber < 10) {
   // statements
}
else if (aNumber >= 10 && aNumber < 20) {
   // statements
}
else {
   // statements
}
```

the test plan must include values of *aNumber* that will cause each of the three possible sets of statements to be executed. These will probably already be boundary values in the test plan, but more values must be included if not.

When testing a loop the crucial thing to consider is the initial value of the condition controlling the loop. Here:

```
int aNumber;

while (aNumber < 10) {
   // statements
}
```

it is important to test the program with values that will make the condition true and so cause the loop to execute and also values that will make the condition false and so mean that the loop is never executed.

It should be possible to represent each route through a program by a set of input values. If it is not possible to specify a set of values that will cause a certain route to be followed this rather begs the question of why the route is there! This list of values can be quite long, even for quite a small section of a program:

```
int grade;

char answer;

do {
   System.out.print ("Enter the grade : ");
   grade = Console.readInt ();

   if (grade > 69) {
      System.out.println (grade + " is an A");
   }
   else if (grade > 59) {
      System.out.println (grade + " is a B");
   }
```

```
    else if (grade > 49) {
      System.out.println (grade + " is a C");
    }
    else if (grade > 39) {
      System.out.println (grade + " is a D");
    }
    else {
      System.out.println (grade + " is a E");
    }

    System.out.println ("Run Again? (y/n) ");
    answer = Console.readChar ();

    while (answer != 'y' && answer != 'n') {
      System.out.println ("Please enter 'y' or 'n'");
      System.out.println ("Run Again? (y/n) ");
      answer = Console.readChar ();
    }

  } while (answer == 'y');
```

This code (which we believe is correct!) converts a numeric grade into a simple letter scale. It also includes a simple dialogue to allow the user to run the program again. It is fairly trivial.

The test plan for this code (and this would be only one small part of a program) is much less trivial. It must include:

- values of *grade* that cause each possibility in the conditional statement to be executed;
- values of *answer* that cause each possible dialogue to be executed.

The first of these is quite straightforward. Each possibility requires boundary values and at least one typical value. The dialogue is more complex. The testing for this must deal with the possibility that the user provides a correct value of *y*, a correct value of *n*, and a range of incorrect values. This must be tested in both places where the user is asked to enter a value even though the code appears to be identical.

The three possible test cases for the first prompt in the dialogue are:

- the user enters *y* to the first prompt, where the program should prompt for another grade;
- the user enters *n* to the first prompt, where the program should finish.
- the user enters another value to the first prompt, where an error message should be displayed and the user prompted again.

The same test cases also apply to the second prompt. It is easy to see how identifying all the possible cases can take some time. It is still important to do this thoroughly.

There may be a temptation to test parts of a program in isolation. In this example, the `if` statement and the two parts of the dialogue could be isolated and tested separately. This would shorten the test plan and the time taken for testing considerably but it could lead to overconfidence in the program. In a program of any size it is entirely possible that a statement in one part of the program can have an unexpected side effect elsewhere in the program. While testing of the individual parts of a program is still a good idea, it is vital that the final program is tested as a complete unit before it is released.

Other ways of testing

So far all the testing described has concentrated on providing input values and examining the output values. If the output values are as expected then the program is assumed to work for that particular test case. In this process the program itself has been treated as a "black box", which is precisely what this type of testing is called. The program is a black box in that its inner workings are never examined; if it produces the correct output then it is assumed to work.

Black Box testing is usually effective but thorough testing requires that the program itself be examined. To see why this can be necessary consider this example:

```
int first, second;

System.out.print ("Enter the first number : ");
first = Console.readInt ();

System.out.print ("Enter the second number : ");
second = Console.readInt ();

int total = first + first;
System.out.println ("The total is " + total);
```

You can probably immediately see the error in this program. If you can't, look closely at the calculation just before the final output line. The error is easy to see but only because the program code is available. This program would pass some test cases in any black box testing; cases where the two numbers input were the same would appear to work. It is even possible that a poorly written black box test plan would lead someone to assume that the program did indeed work as expected.

Testing by examining the program is called "White Box" testing. The program is now a white box and its statements are available to be examined. There are two possible approaches to white box testing. Both involve examining a printout of the program rather than running it on a computer.

The first approach is to "dry run" the program. This involves taking a test case and going through the program by hand, keeping track of the values of all the variables and of what is output. In a way this is the same as black box testing because the final output is available but the advantage is that all the intermediate values are available too. This process can also be a very useful tactic when debugging a program rather than testing it.

The other approach is similar but involves two people. The idea is that someone, probably the programmer, "walks through" the program with someone else. The walkthrough involves explaining each line of the program in turn and explaining the possible routes through, much like a dry run. The process of working through the program can often highlight errors or reveal cases that have not been considered. The programmer is rarely the best person to test a program and should certainly not be allowed to test the program alone. The presence of another person makes the testing more likely to be thorough.

Black box or white box?

Both these approaches to testing have their advantages. White box testing can often find errors in programs that black box testing cannot and can sometimes show up holes in the black box test plan. A black box test can only ever show the presence of an error in the program, it can never show for certain exactly

what is wrong. It is only when the program is examined in a white box test that the cause of the error can be found.

For thorough testing both approaches should be used. It is probably sensible to use black box testing first, not least because most of the time the process can be automated. But when a program passes all the black box tests it should still be thoroughly examined in a proper white box test.

Testing classes

Much of the discussion of testing so far has focused on testing complete programs. This has assumed that the classes used by the programs are known to work correctly. Obviously for this to be the case the classes themselves must also have been tested. They should certainly be thoroughly tested before they are ever used in a program, and we have been testing many of the classes in this book as we have gone along. It can be very frustrating and time consuming to find and correct an error in a program only to discover that the cause or the error is hidden away in a class used by the program.

Of course, classes cannot be tested in the same way as programs. Classes cannot be run; there is no input to provide and no output to trace. The key to testing a class is to use a small program that tests each method of the class in turn. This allows for black box testing; the methods themselves should also be examined in a white box test. We have met such *driver programs* before, when we have written simple methods to allow us to test a class.

When a class is being developed, a program to test it may already be provided or may be developed by another programmer. There may be a *test harness*, a program that can take the implementation of the class and test it. The class will be acceptable only when it passes all the tests run by the test harness successfully. Alternatively, the programmer who developed the class may have to write a simple driver program that will allow the class's methods to be tested.

In many of the classes we have written we have also written a small driver program in the `main` function. This is a fine habit to get into so that you can make sure that your classes work. Another possibility is to use a standard name for a method, *`test`* say, that runs a small test program for each class.

The first and simplest approach to writing a driver is simply to write a program that calls each of the methods and displays the results. These are examined to see if they are as expected in a way that is very similar to black box testing. A simple program like this for one of our very familiar *`Duck`* classes might include:

```
public static void main (String args[])

{
   String name;
   int x, y;

   // Declare a Duck

   System.out.println ("Creating Duck...");
   Duck aDuck = new Duck ();

   // Set some initial values

   System.out.println ("Setting as 'Elvis' at 5, 3...");
   aDuck.setName ("Elvis");
   aDuck.setX (5);
   aDuck.setY (3);
```

```
    System.out.println ("Attributes are now: ");
    System.out.println ("Name: " + aDuck.getName ());
    System.out.println ("Position: " + aDuck.getX () + ", "
                                    + aDuck.getY ());
}
```

When it is run the program announces which methods it is using and displays the results:

```
Creating Duck...
Setting as 'Elvis' at 5, 3...
Attributes are now:
Name: Elvis
Position: 5, 3
```

The programmer can determine the correct results by hand and then examine those produced by the driver program to see if the class is working as expected.

A more thorough approach, and one that allows a complete test plan to be carried out, involves writing a simple menu to allow the user to call each method as required. This type of driver program can be quite long but it is easy to write; the menu structure is just like the one we used in the case study. As before, the programmer can work out a set of tests and can then apply them using the menu. The actual results can be compared to those produced to determine whether the class works correctly.[1]

These driver programs are simply used to test the programs so there is no need for any sophisticated error checking in them or even for any checking of the user's input. They simply allow the tester to use the class and to examine the results. When armed with a suitable plan the tester should be able to discover whether or not the class works as expected. There is an implicit assumption here, of course, that the driver programs are correct. They should also be tested! It is very annoying to discover that an apparent error in a class is caused by sloppy programming in the driver.

Building and testing a program

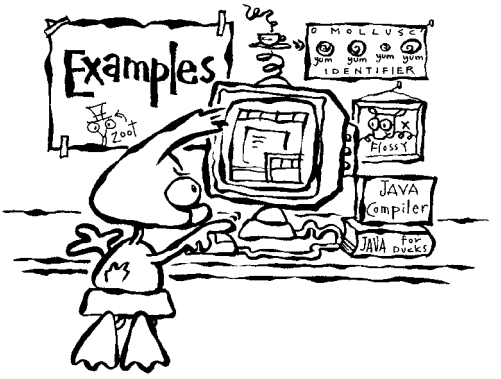
A complete Java program involves classes as well as the program itself. It is important that the classes are written and tested before the program is started (or at least before there is any contact between the two). There is very little point in writing a program that relies on an untested class; if the two are developed and tested separately it is always much easier to trace and correct errors and bugs. This does not eliminate the possibility that writing a program using a class will show up errors in the class, but it should reduce the likelihood.

Thorough testing is a vital part of developing a program. It is even more important in a language like Java, where programs are built up of various components and are often written in many separate files. You should always test

1 That could have been written as "appears to work correctly". Maybe it should have been. Remember that this is what testing is demonstrating. Testing gives us confidence that a program or class works as we expect and want; testing can never prove that a program or class works correctly.

your programs and you should certainly never be tempted to regard testing as an afterthought.

Never be tempted to say things like "it looks as if it works"; always be sure that it works!



Example 1 – Counting letters

Elvis has decided to take up word games. To help him in this new enterprise he has written a simple program to count the number of vowels and consonants in a word. Unfortunately, his program is not producing the correct results. Elvis cannot see the error and so he decides to dry run the program to see what that reveals.

Here is the program:

```
/* WordStats.java - Count consonants and vowels in a word.

   Author    : AMJ
   Date      : 31st December 2002
   Tested on : Red Hat 7.3, JDK 1.4.0
*/

import htpuj.*;

public class WordStats
{
   private int consonants, vowels;
   private String word;

   public WordStats ()
   {
      consonants = 0;
      vowels = 0;
      word = "";
   }

   public void setWord (String w)
   {
      word = w;
   }

   public void count ()

   {
      for (int i = 0; i < word.length () - 1; i ++) {
         if (word.charAt (i) == 'a' || word.charAt (i) == 'e'
            || word.charAt (i) == 'i' || word.charAt (i) == 'o'
```

```
            || word.charAt (i) == 'u') {
          vowels ++;
        }
        else {
          consonants ++;
        }
      }
    }

    public void printResults ()
    {
      System.out.println ("\"" + word + "\" contains ");

      if (vowels == 1) {
        System.out.print ("1 vowel and ");
      }
      else {
        System.out.print (vowels + " vowels and ");
      }

      if (consonants == 1)
        System.out.println ("1 consonant");

      }
      else {
        System.out.println (consonants + " consonants");

      }
    }

    public static void main (String args[])
    {
      WordStats ws = new WordStats ();

      // get word
      System.out.print ("Enter word: ");
      ws.setWord (Console.readString ());

      // analyse
      ws.count ();

      // output results
      ws.printResults ();

    }
}
```

What would the dry run reveal? What is the error?

There is nothing obviously wrong with this program. Running it would reveal that the results are always wrong. You might want to get hold of the program from the web site and try it out. Remember that it doesn't work!

Examples of its output are:

```
Enter word: bluej
"bluej" contains 2 vowels and 2 consonants

Enter word: elvis
"elvis" contains 2 vowels and 2 consonants
```

Some people (people who are very good at word games and puzzles, probably) might be able to deduce what the error is from just these two examples, but dry running the program provides a more structured way to investigate what is going on.

The first stage is to construct a simple table to keep track of the values of the variables. The initial values of the variables are added first:

	word	consonants	vowels	i
Initial		0	0	

Then the program is examined a line at a time, and each change in the value of a variable is recorded in the table. Immediately before the `for` loop is reached the table reads:

	word	consonants	vowels	i
Initial		0	0	
Before loop	bluej	0	0	

The loop uses two methods of the string class, `charAt` and `length`. To keep track of these it is easiest to add them to the table too:

	word	consonants	vowels	i	length	charAt(i)
Initial		0	0			
Before loop	bluej	0	0			

The table is updated again and again as the program progresses. By the end of the first execution of the loop it is:

	word	consonants	vowels	i	length	charAt(i)
Initial		0	0			
Before loop	bluej	0	0			
After loop 1	bluej	1	0	0	5	b

After this point the condition controlling the loop:

```
i < word.length () − 1
```

can be seen from the table to correspond to:

```
0 < 5 − 1
```

which is plainly still true and so the loop will execute again. This continues:

	word	consonants	vowels	i	length	charAt(i)
Initial		0	0			
Before loop	bluej	0	0			
After loop 1	bluej	1	0	1	5	b
After loop 2	bluej	2	0	2	5	l
After loop 3	bluej	2	1	3	5	u
After loop 4	bluej	2	2	4	5	e

At this point the condition is still:

```
i < word.length () − 1
```

which the tables reveals is now:

```
5 < 5 − 1
```

or indeed:

```
5 < 4
```

which is obviously false. This means that the loop will stop and the results will be output.

It should now be clear what the error is; the final character of the string is never being processed. The mistake is in the `for` loop continuation condition:

```
for (int i = 0; i < word.length () - 1; i ++)
```

which is clearly causing the loop to execute one too few times. The solution is obvious:

```
for (int i = 0; i < word.length (); i ++)
```

This change will[2] make the program work correctly.

Example 2 – Baddy's pies

Buddy is rather amused at Elvis's testing dilemma, but he is less smug when he encounters a testing dilemma of his own.

Buddy has been working on a program to divide the weekly supply of pies (kindly provided by Mr Martinmere), but is dismayed when it doesn't seem to work properly. The program is not very complicated; it just takes the number of pies and the number of ducks and displays how many Buddy should give to each duck and how many will be left over. Here it is:

```
/* PieShare.java - Shares pies between ducks.

   Author    : AMJ
   Date      : 31st December 2002
   Tested on : Red Hat 7.3, JDK 1.4.0
*/
```

2 should?

```
import htpuj.*;
public class PieShare
{
  private int pies, ducks;
  private int piesPerDuck, leftOver;
  public PieShare ()
  {
  }
  public void setPies (int p)
  {
    pies = p;
  }
  public void setDucks (int d)
  {
    ducks = d;
  }
  public void calculate ()
  {
    piesPerDuck = ducks / pies;
    leftOver = ducks % pies;
  }
  public void printResults ()
  {
    System.out.print ("Each duck should get ");
    if (pies == 1) {
      System.out.println ("1 pie.");
    }
    else {
      System.out.print (pies + " pies.");
    }
    if (leftOver == 1) {
      System.out.println ("There will be one pie left over.");
    }
    else {
      System.out.println ("There will be " + leftOver +
                          " pies left over.");
    }
  }
  public static void main (String args[])
  {
    PieShare ps = new PieShare ();
    // get input values
    System.out.print ("How Many Pies? ");
    ps.setPies (Console.readInt () );
    System.out.print ("How Many Ducks? ");
    ps.setDucks (Console.readInt ());
    // analyse
    ps.calculate ();
    // output results
    ps.printResults ();
  }
}
```

At the moment, the output of the program is wrong, and it has to be said somewhat baffling. For example:

```
How Many Pies? 100
How Many Ducks? 10
Each duck should get 100 pies.
There will be 10 pies left over.
```

This is clearly wrong, but what is wrong with the program?

Problems such as this are often very difficult for the programmer to find. Black box testing is clearly not helping very much so some sort of white box testing is the only way forward. It is quite likely that by showing his program to another programmer Elvis could quickly have the program solved.

The error in the program is in fact quite subtle, but very obvious once you notice it. If you can see it immediately, go and get yourself a prize!

It is precisely the sort of error that another programmer seeing this program for the first time could well spot immediately, but which the original programmer could spend many fruitless hours trying to find.

The problem is that Buddy has mixed up the order of values in his calculation. If you think about it for a minute, this:

```
piesPerDuck = ducks / pies;
leftOver = ducks % pies;
```

is clearly nonsense. The values are the wrong way round. This is correct:

```
piesPerDuck = pies / ducks;
leftOver = pies % ducks;
```

This is a simple mistake, but mistakes like this can be very difficult to spot, especially if they're in a program that you have written. Subtle errors such as this are often best found by other programmers.

19.1 Examine this conditional statement. What is odd about it?

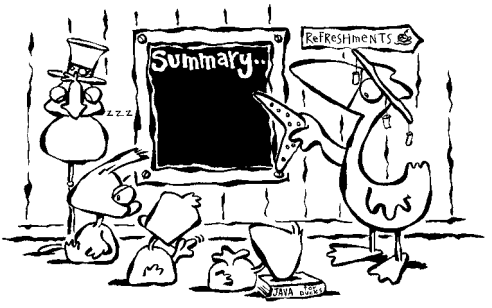
```
int aNumber;
if (aNumber < 10)
{
   // statements
}
else if (aNumber >= 10}
{
   // statements
}
```

```
else
{
   // statements
}
```

19.2 Write a program that prompts the user to enter five words and then sorts them into alphabetical order. Use an array list and a Bubblesort. Dry run the sorting function in your program with the words "sausage", "egg", "bacon", "tomato", and "mushroom".

19.3 Implement the *Duck* class from this chapter. Use a driver program to test your implementation.

19.4 Write a driver program for the *Cricketing Duck* class from last chapter. Use it to test the methods in the class. Pay particular attention to the methods that were not used in any of the programs!



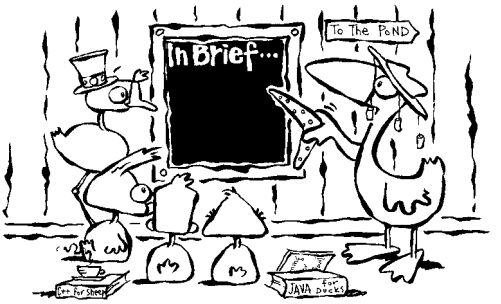
As programs become more and more complex so does the process of testing. Testing should never be ignored. It should be seen as part of the process of programming; a good programmer will always test the program as it is being developed. A program is not complete when it has been written; it is complete only when it has been thoroughly tested.

When you start to learn to program there can be a temptation to be so relieved that a program works that you cannot bear to test it. Testing can uncover mistakes that you have made and can mean that you are in for a lot more work. Try not to succumb to this temptation. Always test your programs as thoroughly as you can, and never be afraid to help other people test theirs.

Testing should always be a structured process and should always be carried out according to a plan. The plan should be written before the program; don't be tempted to write the test plan to fit the parts of the program that work and to hide the rest!

Testing a program is as important as developing it in the first place. Without thorough testing any program is worthless.


Chapter 20
Onward!



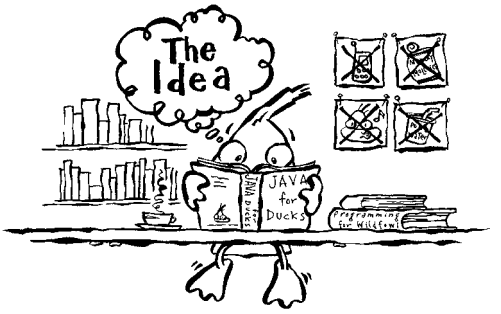
This is the end of the book but it is not the end of Java.

Far from it, in fact.

Java has a great many more features that you haven't yet seen or had any need to use. Java is continuing to develop and more and more features will certainly be added in the future. As Java becomes more popular, and as computer programs become more and more complex, more and more powerful features are added to the language. Although you haven't seen any of these yet you should hopefully now be able to pick them up as and when you need to use them.

This chapter describes five of the more advanced features that would make a good next step in your learning. There's nothing especially complicated here but all this chapter aims to give you is a quick overview. By now you should be able to understand a new programming concept and then go and try it out for yourself.

After reading this chapter you should know about some of the more advanced features of Java that have not been covered in this book. You should understand what they offer and you should know how to find out more about them if you need to.



This chapter describes five more advanced features of Java:

- Exception handling;
- Input and output using files;
- Overloading;
- Inheritance;
- A very basic introduction to Graphical User Interfaces (GUIs).

Each of these is described in turn; they are arranged in no particular order.

These sections do not aim to provide a complete description of these ideas. They should give you a flavour of what each topic is about; this should be just about enough so that you can try them out yourself in your own programs.

Exception handling

Sometimes, when a program is running, something happens that is unexpected. In Java, this is called an *exception*. If Java detects such an event, it "throws" an exception to indicate that something has gone wrong. We have ignored the possibility of exceptions in all the programs in this book (even if they have cropped up at times); obviously this is not a good idea in programs that are required to be robust.

The main difference between exceptions and other ways of signalling errors is that when an exception is thrown, the program is guaranteed to halt unless the programmer takes measures to handle the exception. A mechanism is provided to "catch" any exception that Java "throws" allowing the program to perform some suitable action. Let's look at an example.

So far we have assumed that a program's user has always entered an integer when one was expected. This is actually flawed thinking, since users are unlikely to behave in such a helpful and predictable way. Here is a simple program fragment that reads an integer:

```
int i;

System.out.print ("Enter a number: ");
i = Console.readInt ();
System.out.println ("You entered " + i);
```

This fragment works just as expected when the user enters an integer as expected:

```
tetley% javac ReadInteger.java
tetley% java ReadInteger
Enter a number: 12
You entered 12
```

Things start to go wrong when the user enters a string instead:

```
tetley% java ReadInteger
Enter a number : seven
Exception in thread "main" java.lang.NumberFormatException: For input
string: "seven"
                  at
java.lang.NumberFormatException.forInputString(NumberFormatException.
java:48)
                        at java.lang.Integer.parseInt(Integer.java:468)
                        at java.lang.Integer.parseInt(Integer.java:518)
                              at htpuj.Console.readInt(Console.java:28)
                                at ReadInteger.main(ReadInteger.java:8)
```

The output on your Java system might look a little different, of course, but it should look something like this. Crucially, it should contain the name of the exception that has been thrown.

Java is telling us here that an exception has been thrown. The name of the exception is on the first line of the error – `NumberFormatException`. The rest of the message is telling us where in Java's internals this has happened; to a beginner, this is not especially useful information, except that the last two lines

do tell us that this has happened in a call to *Console.readInt* from line 8 of the file *ReadInteger.java*. This error is obviously best avoided.

The trick in doing this is to catch the exception. The statements that might throw an exception are enclosed in braces with the `try` statement:

```
try {
   i = Console.readInt ();

   System.out.println ("You entered " + i);
}
```

This has no effect on the way in which the program runs. The difference is now that the program can catch the exception:

```
try {
   i = Console.readInt ();

   System.out.println ("You entered " + i);
}

catch (NumberFormatException exception) {
  System.out.println ("Not an Integer!");
}
```

This code simply prints out an error message if the exception is thrown. The effect is now:

```
tetley% javac ReadInteger.java
tetley% java ReadInteger
Enter a number: twelve
Not an Integer!
```

You can place any legal Java code inside the `catch` block, so in the case above, we print a useful warning message. Often, the `catch` block is used to fix or work around an error where possible, or to tidy up before exiting abnormally, by closing open files, logging the error and the like. A common and useful method to include in the catch block is the built-in `printStackTrace` method (which can be called on any exception object). This prints a lot of debugging information (as we saw above with the `NumberFormatException`), and can be useful for tracing obscure errors in your code, especially when shown to a seasoned Java buff:

```
catch (NumberFormatException exception) {
  exception.printStackTrace (); // display debugging information
}
```

A final word about exceptions. Some programs you see might have methods that look like the following:

```
public void read () throws Exception
```

This has effectively stated that the method is allowed to throw exceptions, and that the code which calls the method should handle them. Without the final part the compiler will check the method implementation and will warn about possible exceptions that are not trapped. In programs designed to be robust (hopefully all of them!) it is essential to trap all exceptions and to deal with them sensibly.

Input and output using files

Taking the case study program as an example, it is obvious that only the most basic programs operate without receiving input from files, and without writing at least some data to files. However, up to now we have not learned how to achieve this in Java. Thankfully help is at hand, as Java provides all manner of classes allowing you to read, write and otherwise manipulate files. These classes live in the package `java.io`, and we will briefly discuss only two such classes (`FileReader` and `FileWriter`), and two wrapper classes which handle buffering the reading and writing for efficiency.

The overhead of using the `FileReader` and `FileWriter` classes directly to transfer data can be massive, especially if there is a lot of data to access in many small chunks. Think back to the cooking analogy near the start of the book – when weighing out 200 g of sugar, it is easier and more efficient to open the packet and transfer the contents onto the scales a handful, rather than a granule, or even a spoonful, at a time.

Similarly with file reading and writing, it is far more efficient to read and write in a few large chunks (say a line at a time), rather than a large number of small chunks (say word by word, or character by character). Java provides two classes, `BufferedReader` and `BufferedWriter`, which take care of reading and writing data in large chunks, sometimes even without the programmer knowing. Another handy side effect of using the buffer classes is that the interface is much easier for beginners to learn; the `FileReader` and `FileWriter` classes tend to operate on a character at a time, or with arrays of characters, but the buffer classes hide this complexity, allowing the programmer to work entirely with more familiar and less cumbersome `String`s. Full details of all these classes can, of course, be found in the API documentation.

One final point to mention, which is important to consider whenever you are performing file input or output in Java, is that just about all the methods can potentially throw exceptions (usually `IOException`), so a method which calls file input or output code should contain a `try` block with a matching `catch` block to handle the exception.

Reading data from a file

`FileReader` has several constructors, the one we will use takes one argument, a `String`, containing the path to the file that we want to read from:

```
FileReader file = new FileReader ("c:\path\to\file");
```

In order to make the reading more efficient, we pass a `FileReader` object as an argument to the `BufferedReader` constructor:

```
FileReader file = new FileReader ("c:\path\to\file");
BufferedReader reader = new BufferedReader (file);
```

The two statements can be combined, by creating the `FileReader` object on the fly (which is fine in this case, as we don't need to access the object directly again):

```
BufferedReader reader = new BufferedReader
                        (new FileReader ("c:\path\to\file") );
```

Using a `BufferedReader` to read data is easy. Internally, the class keeps track of the current position in the file, and the single method we will discuss,

readLine, reads all the data it can from the current position up to but not including the next carriage return. Let's create a dummy file to test this on:

```
This is a line of text.
This is another line.
```

Save the file and make sure you remember its name. For the sake of argument, let's call it *testfile.txt*. Now to initialise the reader object (remembering to include the appropriate libraries):

```
import java.io.*; // IMPORTANT!

try {
  BufferedReader reader = new BufferedReader
                          (new FileReader ("testfile.txt") );
}
catch (IOException ioe) {
  System.err.println ("Error reading from testfile.txt");
// ioe.printStackTrace (); // uncomment for more verbose output
}
```

We can now read from the file using the readLine method:

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

This gives the output:

```
This is a line of text.
```

Reading a whole file is not as straightforward as including the readLine call in a loop because no exception is thrown when the end of the file is reached. Thankfully, the readLine method returns the special value null when it attempts to read past the end of the file, so we can include a check for this in our loop:

```
String tmpString = " ";

do {
    System.out.println (tmpString);
    tmpString = reader.readLine ();
} while (tmpString != null);
```

This code should loop through the file, printing out each line in turn. When the end of the file is reached, the readLine method returns the value null, and the check tmpString != null fails, dropping control out of the loop.

The final thing we need to do is close the input stream. This stops the file from getting corrupted, and also releases resources that the system can use elsewhere. The method to close the stream is the imaginatively titled close:

```
reader.close ();
```

Writing data to a file

We have encountered the classes needed for input from files, so now let's look at their partners in crime, the output classes. As we said at the start of the section, the classes are called FileWriter and BufferedWriter, and their behaviour is again fairly easy to comprehend.

Similar to the way we created a reader object above, we create a BufferedWriter object, which takes a FileWriter object as an argument. This FileWriter object in turn takes a String containing a filename as an

argument. This all takes place inside the requisite `try` block, with its companion `catch` block below.

Creation of the writer object is straightforward:

```
BufferedWriter writer = new BufferedWriter
                        (new FileWriter ("c:\path\to\file") );
```

Once again, we only require a handful of methods. The first of these is the `write` method, which takes a `String` as an argument, and writes the contents of that `String` to the file:

```
writer.write ("testing 1 2 3");
```

One limitation of this method compared with the more familiar `System.out.print` methods is that you should pass a `String` and only a `String` as an argument. There is another version of this method which takes an `int`, but this does something completely different, and can cause all manner of confusion. Thankfully, there is a way to force the compiler and JVM to treat something of a different type as if they were a `String`; since we can append any of these values to a `String`, we simply append them to an empty `String`:

```
writer.write ("" + 4); // writes the String '4' to the file
```

It is probably also worth remembering that the `write` method will write exactly what you tell it to, so if you want to write a line of text, you'll need to make sure you also write the '\n' (*newline*) character at the end of the line![1]

The second method we need to know about is the `close` method, for the same reasons outlined earlier for the reader class. The syntax is exactly the same:

```
writer.close (); // close the output stream
```

However, this is not the whole story. As we are buffering the data to output, we need to ensure that it has all been written to the file before we close the output stream. There is a third method, `flush`, which forces a transfer of the buffer's contents to the file (known as flushing the buffer). This is frequently found in file writing code:

```
writer.flush (); // write every last byte
writer.close (); // close file
```

So, let's look at some example code for file writing. Then we'll revisit the case study to incorporate what we've learned into a real program.

```
import java.io.*;

...

try {
    BufferedWriter writer = new BufferedWriter
                            (new FileWriter ("testfile2.txt") );
    writer.write ("Oh let the sun beat down upon my face\n");
    // any other output we wish
    writer.flush (); // flush the buffer
```

---

1 In fact, if we instantiated a `PrintWriter` object (from package `java.io`), taking this `BufferedWriter` object as an argument, we would have access to the same `print` and `println` methods as the `System.out` class. This means that we could use `printWriter.println (line);`, rather than `writer.write (line + "\n");` Try it for yourself.

```
    writer.close (); // close the file
}
catch (IOException ioe) {
  System.err.println ("Error writing to testfile2.txt");
// ioe.printStackTrace (); // uncomment for more verbose output
}
```

Now, examining the file (`testfile2.txt` in this case) in your editor of choice, it should hopefully contain the line that we wrote in our program.

Extending the DuckDatabase class to incorporate file I/O

Rather than include the whole class again, it makes sense to only include the changes we need to make over the first version.

For the time being, let us make the use of files optional; this means we need to extend the main menu to include options for reading from and writing to files. We should also locate the reading code in a method of its own (*`readFromFile`*, say), and likewise the writing code (in the interest of consistency, *`writeToFile`*). This leaves the class's interface unchanged so that any existing programs will still work as expected.

Rather than hard-wiring filenames into our code, it would be useful if we could pass the name of the required file to these methods as an argument.

Extending the menu is easy enough and we print a couple of extra lines to the console, and add a couple of extra options to the `switch` statement:

```
System.out.println ("* R - Read data from file    *");
System.out.println ("* W - Write data to file     *");

case 'r':
case 'R':
         System.out.print ("Enter filename : ");
         String name = Console.readString ();
         readFromFile (name);
         break;

case 'w':
case 'W':
         System.out.print ("Enter filename : ");
         String name = Console.readString ();
         writeToFile (name);
         break;
```

Next we need to implement the *`readFromFile`* and *`writeToFile`* methods. But before this, we should really decide on a file format for these methods – that is, the order in which we will store data elements for each duck, and more generally, how we order the ducks in the file. Probably the most obvious format would be to have each attribute on a separate line, in the order in which they are found in the *`Duck`* class:

- name
- team
- overs bowled
- wickets taken
- runs conceded
- innings batted
- runs scored

The easiest way to separate the data for different ducks would be to have a blank line between the last value of the first duck and the first value of the second duck, and a marker after the last duck (say "EOF" for end of file). This way the code for reading the data can be written inside a loop, and the loop condition can be:

```
if the next line contains nothing but the String "EOF",
  finish,
else continue
```

The logic for the *writeToFile* method is as follows:

```
OPEN FILE FOR WRITING
FOR EACH DUCK IN THE LIST
  WRITE DATA TO FILE IN SPECIFIED ORDER
  IF THIS IS THE LAST DUCK
    WRITE "EOF" TO FILE
  ELSE
    WRITE BLANK LINE TO FILE
FLUSH AND CLOSE FILE
```

Transferring this into working Java code is not too difficult, especially as most of the concepts should be familiar to you by now. We will need a `for` loop similar to existing ones in the *DuckDatabase* class, in that it must loop through an `ArrayList`, creating a temporary *Duck* object with each iteration, and dump the data from this object to the file.

```
public void writeToFile (String filename)
{
  Duck tmpDuck; // temporary object

  try { // code may throw exceptions!
    BufferedWriter writer = new BufferedWriter
                          (new FileWriter (filename) );

    for (int i = 0; i<ducks.size (); i++) {
      tmpDuck = (Duck)ducks.get (i); // get reference to duck

      writer.write (tmpDuck.getName () + "\n");
      writer.write (tmpDuck.getTeam () + "\n");
      writer.write (" " + tmpDuck.getOvers () + "\n");
      writer.write (" " + tmpDuck.getWickets () + "\n");
      writer.write (" " + tmpDuck.getRunsConceded () + "\n");
      writer.write (" " + tmpDuck.getInnings () + "\n");
      writer.write (" " + tmpDuck.getRunsScored () + "\n");

      // if we've written the last duck, write "EOF"
      if (i == ducks.size () - 1) {
         writer.write ("EOF");
         }
         else { // otherwise, write a blank line
           writer.write ("\n");
         }
      }
      writer.flush (); // ensure every last byte is written
      writer.close (); // close the file
  }
  catch (IOException ioe) { // handle exceptions
      System.err.println ("Error writing to " + filename);
```

```
    // ioe.printStackTrace (); // uncomment for full output
  }
}
```

The *readFromFile* method is similar to the *writeToFile* method in many ways (which should be obvious, given that they are exact opposites). Logically, it should work as follows:

```
OPEN FILE FOR READING
UNTIL END OF FILE REACHED
  READ DATA, SETTING DUCK VALUES IN SPECIFIED ORDER
CLOSE FILE
```

Again, translating this logic into Java code should be fairly easy by now, as the principles are familiar. This time, rather than a loop where we read values from a *Duck* object and then write them to a file, we will read values from a file and then write them to a *Duck* object!

The only real complication is that the file contains only `String`s, and most of the values we wish to change are `int`s, so we will need to convert between the two. This is not a major problem, as we have already met the `Integer.parseInt` static method:

```
String s = "4";
int i = Integer.parseInt (s); // i now contains the int value 4
```

Therefore in all the cases where we expect an `int` value, we will need to call this `parseInt` method on the return value of the `readLine` method. Other than that, there is very little Java in this method that you haven't already met several times.

```
public void readFromFile (String filename)
{
  Duck tmpDuck; // temporary object
  String buf = " ";

  try {
    BufferedReader reader = new BufferedReader
                            (new FileReader (filename) );

    do {
        tmpDuck = new Duck ();
        tmpDuck.setName (reader.readLine ());
        tmpDuck.setTeam (reader.readLine ());
        tmpDuck.setOvers (Integer.parseInt (reader.readLine ()));
        tmpDuck.setWickets (Integer.parseInt (reader.readLine ()));
        tmpDuck.setRunsConceded (Integer.parseInt
                                        (reader.readLine ()));
        tmpDuck.setInnings (Integer.parseInt (reader.readLine ()));
        tmpDuck.setRunsScored (Integer.parseInt
                                        (reader.readLine ()));
        ducks.add (tmpDuck);

        // read next line - blank means more ducks, EOF means end.
        buf = reader.readLine ();

    } while (!buf.equals ("EOF"));

    reader.close (); // close file
  }
```

```
    catch (IOException ioe) {
       System.err.println ("Error reading from " + filename);
       ioe.printStackTrace ();
    }
  }
```

Inheritance

It is not uncommon for an application to contain many classes that are very similar. Applications will deal with different types of people, bank accounts, books, or animals. There are many occasions when a method written for one class can be used in another and where different classes have the same attributes. In this case there can often be a great deal of code that has to be duplicated in the different classes. We have seen this several times when code for different types of animal has had to be duplicated in different classes.

Sometimes the classes form a hierarchy. For example, if an application needs classes representing different kinds of people – students, teachers, bus conductors – many of the different classes will have the same attributes. It would be very wasteful to have to implement the same methods in each class.

Java provides a mechanism to allow for this, called inheritance. Inheritance allows the programmer to define one basic class and to allow other classes to inherit all the attributes and methods of this class. The inheriting classes have all the attributes and methods of this class and can also have their own. The most common example is probably a class for `People` that would hold names, birthdays, and so on, and would be inherited by classes representing particular types of people such as students and bus conductors.

The class that contains the common attributes and methods is called the *base class* and the classes inheriting from it are called *derived classes*. The base class is defined first in the usual way:

```
public class Person {
  private String name;
  private int age;
  private String address;

  public Person ()
  {
  }

  public void setName (String newName)
  {
    name = newName;
    return;
  }

  public void setAge (int newAge)
  {
    age = newAge;
    return;
  }

  public void setAddress (string newAddress)
  {
    address = newAddress;
    return;
  }
```

```
  public String getName ()
  {
    return name;
  }

  public int getAge ()
  {
    return age;
  }

  public string getAddress ()
  {
    return address;
  }
}
```

Derived classes include only the attributes and methods that are unique to them. All the other attributes and methods are inherited from the base class, the name of which is specified on the first line of the class's definition:

```
public class BusConductor extends Person
{
  private int busRoute;

  public BusConductor ()
  {
  }

  public void setBusRoute (int newBusRoute)
  {
    busRoute = newBusRoute;
    return;
  }

  public int getBusRoute ()
  {
    return busRoute;
  }
}
```

So here an object with the type *BusConductor* has the attribute *busRoute* (together with a selector and a mutator) together with all the attributes and methods of the *Person* class. Similar classes could be implemented for any particular type of person.

Inheritance can save a great deal of coding. There can be many levels of inheritance in an application with derived classes acting as base classes for others.

Overloaded methods

Methods in Java are identified by the combination of the name of the method, the type of the returned value, and the types of any parameters. It is this combination of information together that identifies the method; there is no need for every method in a class to have a different name. A number of methods can have the same name if there is something else that can be used to distinguish them; the method can be overloaded.

Suppose that some application exists that requires much calculation of averages. It might be necessary to calculate the mean of two numbers or three

numbers; to complicate matters further the three numbers might be stored in an array or in individual variables. These methods might be defined as:

```
public double meanOfTwo (int x, int y)
{
  return (double) (x + y) / 2;
}

public double meanOfThree (int x, int y, int z)
{
  return (double) (x + y + z) / 3;
}

public double meanOfArray (int n[])
{
  return (double) (n[0] + n[1] + n[2]) / 3;
}
```

These methods have been given different names but there is no particularly good reason to do this. If all the methods were called simply *mean*:

```
public double mean (int x, int y)
{
  return (double) (x + y) / 2;
}

public double mean (int x, int y, int z)
{
  return (double) (x + y + z) / 3;
}

public double mean (int n[])
{
  return (double) (n[0] + n[1] + n[2]) / 3;
}
```

it would still be obvious which one was being called from the types and number of parameters. This call:

```
mean (a, b);
```

is clearly calling the first version since two parameters are provided, while this:

```
mean (a, b, c);
```

must be a call to the second version as there are three parameters.

In this example the method name *mean* has been overloaded; there are three methods with this name. This is fine as long as the method that is being called can be uniquely determined by examining the type and number of its parameters.

This technique can make programs much more readable and can save the programmer from having to use what amount to artificial names for methods.

The following program illustrates this idea with some more examples. Each time the method *mean* is called the parameters uniquely determine which is to be used.

```
/* Numbers.java

   Example of Overloading Methods.

   Author    : AMJ
   Date      : 8th February 2003
   Tested on : Linux (Red Hat 7.3), JDK 1.4.1
*/
```

```
public class Numbers
{
  private int num1;
  private int num2;
  private int num3;

  private int nums[];

  public Numbers ()
  {
    num1 = 1;
    num2 = 2;
    num3 = 3;

    nums = new int [3];

    nums[0] = 4;
    nums[1] = 5;
    nums[2] = 6;

    return;
  }

  public double mean (int x, int y)
  {
    return (double) (x + y) / 2;
  }

  public double mean (int x, int y, int z)
  {
    return (double) (x + y + z) / 3;
  }

  public double mean (int n[])
  {
    return (double) (n[0] + n[1] + n[2]) / 3;
  }

  public static void main (String args[])
  {
     Numbers n = new Numbers ();

     System.out.println ("Two integer mean : "
                         + n.mean (n.num1, n.num2) );
     System.out.println ("Three integer mean: "
                         + n.mean (n.num1, n.num2, n.num3) );
     System.out.println ("Array mean : " + n.mean (n.nums) );
  }
}
```

All the methods of a class can be overloaded, including the constructor. It is possible to provide a set of constructors that set a variety of combinations of initial values, for example.

A (very) basic introduction to GUIs

Part of the reason Java was developed was to provide a language for cross-platform production of GUIs, and hence packages for building elegant GUIs are part of the core of the language. Therefore it would be a shame not to take a look at this feature of the language, at least in passing. However, this is a huge

topic (indeed, very thick books on the matter adorn Java programmers' bookshelves everywhere!), so this small dip into GUIs must be treated as very basic, and merely a springboard to further study if you so wish.

In larger, more complex GUI-based programs, literally millions of lines of code can be devoted to the workings of the user interface, and very little to the actual core tasks that the program is performing. The program needs to keep track of keyboard and mouse interactions, monitor settings, and much much more, and just about all the Java you would need to do this is way beyond the scope of this book. Thankfully, the designers of Java have provided a way to ignore most of these overheads and simply display a message in a box on the screen. This is hardly ground-breaking, but it can add an impressive touch to an otherwise ordinary program.

The `JOptionPane` class (in package `javax.swing`) has many static methods, which allow the programmer to create simple pop-up dialogue, which should appear on any system running Java with a windowing system (of which Microsoft Windows, Mac OS and X11 are the most likely). So without further ado, let us take one of our earlier examples, which is crying out for a graphical makeover: Bruce's *`Sign`* class.

The current version of this class displays Bruce's messages on the console. This is fine, but something more graphical would surely look more impressive. As we make these changes, most of the existing version of *`Sign.java`* remains intact; we still have a message variable, a constructor, a *`setMessage`* method, and a display method. However, the change comes in the display method where:

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

becomes:

```
JOptionPane.showMessageDialog (null,
                               message,
                               "Bruce's Sign",
                               JOptionPane.INFORMATION_MESSAGE);
```

This looks complicated at first glance, but it is actually quite straightforward. The first argument would normally be a reference to the parent window (that is, the window over which you want your dialog to appear). As we do not know enough Java to create such a window, we leave this as `null`, which the compiler treats as meaning the desktop. Hence the box appears in the centre of the screen.

The second and third arguments are the message and title bar strings, respectively. The message string is the text you wish to display in the dialog box, and the title bar string is the text you wish to display in the bar at the top of the box.

The last argument requires a little explanation. The `JOptionPane` class provides four default icons for the dialog box: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, and `QUESTION_MESSAGE`.

Passing these as the fourth argument will change the icon that appears in the dialog box. There is also fifth option, `PLAIN_MESSAGE`, which corresponds to 'no icon'.

```
/* Sign.java - Bruce's graphical wonder sign

   Author     : GPH
   Date       : 17th July 2003
   Tested on : Linux (Red Hat 9), JDK 1.4.2
*/
```

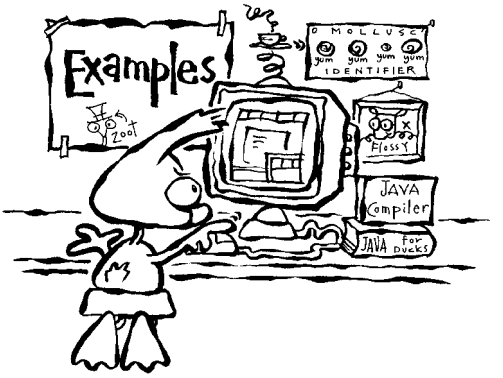
```
import javax.swing.JOptionPane;

public class Sign
{
  private String message;
  public Sign ()
  {
  }
  public void inputMessage ()
  {
    message = JOptionPane.showInputDialog ("Enter a message :");
  }
  public void display ()
  {
    JOptionPane.showMessageDialog (null, message, "Bruce's Sign",
          JOptionPane.INFORMATION_MESSAGE);
  }
  public static void main (String args[])
  {
    Sign sign = new Sign ();
    sign.inputMessage ();
    sign.display ();
  }
}
```

The only line of this code that requires any explanation is the body of the *inputMessage* method. Rather than taking a string value as an argument, here we prompt the user to enter the message for Bruce's sign via another dialogue box. This is the simplest way to obtain user input via a GUI, and is extremely useful. The `showInputDialog` method returns the text entered as a `String` object, and the message attribute is set to this value.

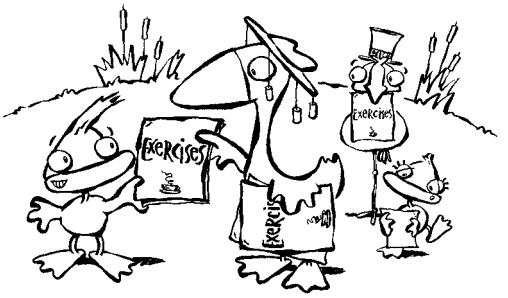
That really is all there is to creating a simple graphical application. Of course, to create a more complicated one, there is a *lot* of learning ahead…

There's more!

That really is all the Java in this book. There is still much more. Many Java reference books run to well over a thousand pages and the language is still growing.

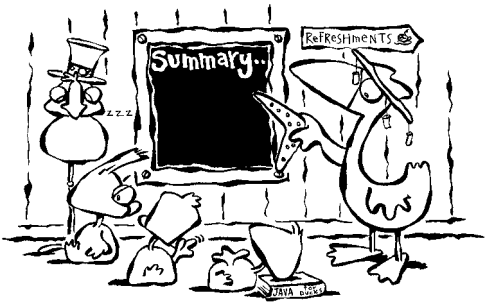


There are no examples this time; there were quite enough examples in this chapter anyway! The time has come now when you should be able to think of your own examples to try …



Only one exercise this time, but make sure that you do it a few times!

20.1 Write a program. It can do anything you like as long as it uses most of the Java that you have seen and used up to now. When you have finished and tested this program, write another. Remember that the only way to become a good programmer is to practise.



This is the end of the book.

Congratulations on getting this far. You'll probably be quite relieved to have made it here. You'll certainly have seen and learned a lot. Remember that learning to program is difficult and if you've got this far you've achieved something that should make you rather proud.

This chapter has shown you just a few of the other features that are available in Java. You should now be able to find out more about these and also about the ones that haven't been mentioned. You should be able to understand and use a Java reference book and you should be able to develop more and more as a programmer.

In a way the end of this book is just a beginning. Programming is a skill that develops with practice. Even the most experienced programmer is always learning. Programmers will always be learning a new language or a new technique. As computers develop in the future so will the languages that are used to program them. There is no end to this trend in sight. Hopefully you are now in a good position to start to become a programmer.

Remember that you have not just learned Java. You have learned programming. Virtually all the ideas and techniques you have seen and used in Java are also available in other languages. The syntax will be slightly different but that is all that you will have to learn when you come to a new language. You'll know what you need to do and will just have to look up the syntax. The hard work is done.

What you need to do now is practise. When you've done that, do some more. The more you write programs the better you become at it and the easier and more natural it becomes. Many programmers find programming an extremely rewarding and creative process. Hopefully you'll come to believe that too.

Our job here is done. Now it's down to you. Whether you're going to go on and learn more Java or whether it's going to be a different language we wish you the best of luck. Remember to keep on learning.

Onward!


Java Reference & Examples
welcome to DUCK iT services
CHARLES DUCK
FLOSSY
JAVA Reference & Examples
Mollusc Index
How to LAND on ICE
Mexpert Gripping

Every programmer, even the most experienced, works with a handy language reference beside the computer. No programmer can be expected to remember all the fiddly little details of the syntax of a modern programming language and it is essential that you get used to using a reference. In the future that might be a book describing all of the Java language but for the moment this section will do.

This chapter gives you examples of all the Java you've seen in this book, all arranged in the same place. It's in just about the same order as the book and the sections are numbered (there's a small contents list at the start) so that you can (hopefully!) easily find the example you need. If you're lucky, some of the examples here might slot right into the program you're trying to write.

If you can't find what you need here you can also try looking through the solutions to the exercises. One of the best ways of working out how to write a program is to look at a program that does something similar; you might even be able to adapt something. Failing that, there's always the index!

Contents

1. Comments

Any line starting `//` is a comment.

```
// This is a comment
// So is this
```

Any part of a line after a `//` is also a comment.

```
x = 0; // This explains why that is complicated
```

Any section of a program enclosed between `/*` and `*/` is also a comment.

```
/*
Anything that appears here is a comment.
*/
```

2. Basic structure

Any file containing a Java program or class definition starts with a comment describing its purpose and providing other useful information. The basic structure after this follows a set pattern.

```
// Any included libraries or files
public class someClass {
   // Attributes
   // Methods
   // main method (if included)
}
```

3. Defining attributes or variables

An attribute variable has an identifier (name) and a type. It is our convention that attributes are always declared to be private, so that they may only be accessed by methods of the class.

```
public int aNumber;
public String aName;
public boolean finished;
```

The variable can also be given an initial value.

```
public int aNumber = 0;
public String aName = "Elvis";
public boolean finished = false;
```

4. Defining constants

Constant values are declared in a similar way to variables. Constant values cannot be changed.

```
final int MAXTURNS = 10;
final String BOOK = "How to Program Using Java";
final double PI = 3.1415;
```

5. Types

The following data types have been used in this book:

`int`	an integer – positive or negative (or 0)
`double`	a floating-point number
`char`	a single character
`String`	a sequence of characters
`boolean`	a Boolean value – true or false
`ArrayList`	a collection of other values

6. Converting between types

The names of these types can be used to convert values between types. This process is known as casting.

```
public int aNumber = 11;
public double aDouble;

aDouble = double (aNumber);
```

7. Assignments

Variables are assigned values in assignment statements:

```
aNumber = 10;
aString = "hello";
aDouble = 3.1415;
```

The right-hand side can be an expression.

```
aNumber = 10 + 2;
aString = "hello " + "world";
aDouble = 1.5 + 3.2;
```

Identifiers can appear on both sides of the assignment operator.

```
aNumber = anotherNumber + 10;
aNumber = aNumber + 1;
```

Various shorthand forms for common operations exist.

```
aNumber ++;   // aNumber is incremented
aNumber --;   // aNumber is decremented
aNumber += 3;  // 3 is added to aNumber
aNumber -= 4;  // 4 is subtracted from aNumber
aNumber /= 2; // aNumber is divided by 2
aNumber *= 4; // aNumber is multiplied by 4
```

8. Operators

There are four basic arithmetic operators. In order of precedence they are:

- `/` Division
- `*` Multiplication
- `+` Addition (of numbers) or concatentation (of `strings`)
- `-` Subtraction

These work as expected for numeric data.

```
aNumber = 2 + 2;      // assigns 4
aDouble = 1.5 + 3.7; // assigns 5.2
```

The addition operator concatenates two strings.

```
aString = "Elvis " + "the Duck";
        // assigns "Elvis the Duck" (without the quotes)
```

Operators are applied in order of precedence. The order is as in the table above.

```
aNumber = 10 + 6 / 2; // aNumber is assigned 13
aNumber = 10 / 2 + 3; // aNumber is assigned 8
aNumber = 10 + 3 * 2; // aNumber is assigned 16
```

The order can be affected by adding brackets.

```
aNumber = (10 + 6) / 2; // aNumber is assigned 8
```

```
aNumber = 10 / (2 + 3); // aNumber is assigned 2
aNumber = (10 + 3) * 2; // aNumber is assigned 26
```

9. Output

Values are displayed on the screen using `System.out.print` and `System.out.println`. The first prints the string supplied as its argument, and the second does exactly the same but also moves output to the next line after printing.

```
System.out.print ("hello");
System.out.println (aNumber);
```

Numeric values can be included in the argument:

```
System.out.println ("The number is " + aNumber);
```

If quotes are required in the output they are preceded with `\`.

```
System.out.println ("\"hello\"");
```

10. Input

We have used our own *`Console`* class for reading values from the user.

```
aNumber = Console.readInt ();
aString = Console.readString ();
aDouble = Console.readDouble ();
aChar = Console.readChar ();
```

The *`Console`* class is not standard Java. The listing of the class included at the end of this section shows the standard Java that is implementing the class.

11. Defining a class

A class is defined in a *`.java`* file. The name of the class is defined, together with its private and public sections.

```
public class Duck {
  // Private Attributes
  // Public Methods
}
```

Attributes are defined in the private section.

```
public class Duck {
  private String name;
  private int age;
  private double value;
  // public methods
}
```

Methods are defined in the public section. There must be a constructor and there are usually selectors and mutators.

```
public class Duck {
  private String name;
  private int age;
  private double value;
```

```
  // Constructor
  public Duck ()
  {
  }
  // Selectors
  public String getName ()
  {
    return name;
  }
  public int getAge ()
  {
    return age;
  }
  public double getValue ()
  {
    return value;
  }
  // Mutators
  public void setName (String newName)
  {
    name = newName;
  }
  public void setAge (int newAge)
  {
    age = newAge;
  }
  public double setValue (double newValue)
  {
    value = newValue;
  }
}
```

A special method called `main` may also be defined. This method is executed by the JVM if it is told to execute the class.

```
public static void main (String args[])
{
  // Any statements
}
```

12. Using a class

An instance of a class is declared by calling the constructor.

```
Duck elvis = new Duck ();
```

Methods are called by joining the identifier of the instance to the name of the method with a full stop.

```
elvis.setName ("Elvis");
elvis.setAge (6);
System.out.println (elvis.getName () + " is" +
                    elvis.getAge ( ) );
```

13. Conditional statements – `if`

The basic `if` statement tests a condition and executes a statement only if the condition is true.

```
String aName;
int age;
bool finished;

if aName.equals ("Elvis") {
  System.out.println ("The name is Elvis");
}

if (age > 10) {
  System.out.println ("The age is greater than 10");
}
if (finished) {
  System.out.println ("All finished!");
}
```

The optional `else` clause defines the statements to be executed if the condition is false.

```
if aName.equals ("Elvis") {
  System.out.println ("The name is Elvis");
}
else {
  System.out.println ("The name is not Elvis");
}

if (age > 10) {
  System.out.println ("The age is greater than 10");
}
else {
  System.out.println ("The age is 10 or less");
}

if (finished) {
  System.out.println ("All finished!");
}
else {
  System.out.println ("Onward!");
}
```

More than one condition can be tested.

```
if (aName.equals ("Elvis") ) {
  System.out.println ("The name is Elvis");
}
else if (aName.equals ("Buddy") ) {
  System.out.println ("The name is Buddy");
}
else {
  System.out.println ("The name is not Elvis or Buddy");
}

if (age > 10) {
  System.out.println ("The age is greater than 10");
}
else if (age < 10) {
  System.out.println ("The age is less than 10");
}
else {
  System.out.println ("The age is 10");
}
```

14. Conditional statements – `switch`

The `switch` statement is a shorthand form of the `if` statement where the possible values for a variable can be listed and different statements are executed for each value.

The variable must be of an ordinal type – integer or character.

```
char direction;

switch (direction) {
  case 'n': System.out.println ("North");
            break;
  case 's': System.out.println ("South");
            break;
  case 'e': System.out.println ("East");
            break;
  case 'w': System.out.println ("West");
            break;
}
```

The `default` case handles unexpected values.

```
switch (direction) {
  case 'n': System.out.println ("North");
            break;
  case 's': System.out.println ("South");
            break;
  case 'e': System.out.println ("East");
            break;
  case 'w': System.out.println ("West");
            break;
  default:  System.out.println ("Invalid direction!);
}
```

More than one value can be associated with a case.

```
switch (direction) {
  case 'n':
  case 'N': System.out.println ("North");
            break;
  case 's':
  case 'S': System.out.println ("South");
            break;
  case 'e':
  case 'E': System.out.println ("East");
            break;
  case 'w':
  case 'W': System.out.println ("West");
            break;
  default:  System.out.println ("Invalid direction!);
}
```

15. Loops – `for` loops

A `for` loop is determinate. It executes a number of times that can be determined before the first execution.

The loop is controlled by an initial statement, a continuation condition, and a statement that is executed each time the loop runs.

```
// Loop 10 times
for (int counter = 0; counter < 10; counter ++) {
  System.out.println (counter);
}

// Loop 10 times, but backwards
for (int counter = 10; counter > 0; counter --) {
  System.out.println (counter);
}
```

16. Loops – `while` loops

A `while` loop continues to execute while some condition is true. It is indeterminate.

```
while (!finished) {
  // run the program
}

while (!finished && attempts < 10) {
  // run the program
}
```

The value of the condition must change inside the loop, or the loop will never terminate.

```
while (!finished) {
  // run the program

  if (answer == 'y') {
    finished = true;
  }
}

while (!finished && attempts < 10) {
  // run the program

  attempts ++;
}
```

If the condition is initially false the statements inside the loop are never executed.

```
bool finished = true;
while (!finished) {
  // never executed
}
```

17. Loops – `do ... while` loops

A `do ... while` loop executes while a condition is true. It is indeterminate.

```
do {
  // run the program
} while (!finished);
```

The value of the condition must change inside the loop:

```
do {
  // run the program

  finished = answer == 'y';

} while (!finished);
```

The condition is tested after the statements inside the loop have been executed. This means that the loop is always executed at least once.

```
bool finished = true;
```

```
do {
  // executed once
} while (!finished);
```

18. Method prototypes

The first line of the definition of a method specifies its prototype. The prototype of a method specifies the name of the method, the type of value returned by it, and the types of its parameters. The types of the parameters are listed in brackets, which are left empty if there are none.

```
public int getAge ()
public void setAge (int)
public double getValuePlusTax (float)
```

A method that does not return any value is defined to return `void`. It is called a void method.

19. Method parameters – value parameters

Methods process values. These can be literal values or variables.

```
Duck elvis;
int age = 10;

elvis.setAge (10);
elvis.setAge (age);
```

Methods may alter the values passed to them but the changes are not made to the variable in the calling program.

If a method has more than one parameter the values must be supplied in the correct order.

```
public void printInColumns (int, char);
elvis.printInColumns ('#', 20); // wrong!
elvis.printInColumns (10, '*'); // correct
```

Values of the basic types – `boolean`, `char`, `double`, `int` – are passed by value.

20. Method parameters – reference parameters

Methods may also change values that are then available to the main program. Values of object types (including strings) are passed by reference.

```
public void summon (Duck d)
{
  int newX = getX ();
  int newY = getY ();
  d.moveTo (newX, newY);
}

  elvis.setX (5);
  elvis.setY (3);
  buddy.setX (0);
  buddy.setY (0);

  elvis.summon (buddy); // buddy is moved to 5, 3
```

21. Static methods

Methods defined as static can be called without reference to an object of the class.

```
public class Duck {
```

```
  public static void greet ()
  {
    System.out.println ("hello, world");
  }
}
Duck.greet (); // Method call
```

22. Arrays

An array stores a collection of values of the same type. The type and the number of values are specified in the declaration.

```
int numbers[];
numbers = new int[4];

int moreNumbers = new int[4];
```

Each value in the array is called an element and can be referenced by its index. The first element is at index 0.

```
System.out.println ("First element: " + numbers[0]);
System.out.println ("Last Element: " + numbers[3]);
```

A common operation is to scan an array using a `for` loop to examine each element in turn.

```
int highest = numbers[0];

for (count = 1; count < 4; count ++) {
  if (numbers[count] > highest) {
    highest = numbers[count];
  }
}
```

23. Array lists

An array list is similar to an array. The main difference is that its size is not fixed. An array list can store values of any object type. It is created in the same way as any other object.

```
ArrayList myJobs = new ArrayList ();
```

An array list is an instance of a class. A value is added the `add` method.

```
myJobs.add ("Washing Up");
```

An element in a vector can be accessed using the `get` method. The elements are indexed in the same way as an array, with the first element being at position 0.

```
System.out.println (myJobs.get (0));
```

The `size` method returns the number of elements in the array list.

```
System.out.print ("There are " + myJobs.size () );
System.out.println (" jobs to do");
```

The method removes an element. It can be used with the `indexOf` method to remove a particular value.

```
myJobs (0);
myJobs (indexOf ("Washing Up"));
```

The whole list may also be emptied.

```
myJobs.clear ();
```

The Console class

```
/* Console.java - convenience class for reading user input
                  from the command line.

   Author    : GPH
   Date      : 17th July 2003
   Tested on : Linux (Red Hat 9), JDK 1.4.2
*/

package htpuj; // states that this class is part of a package
import java.io.*; // for exceptions and reader objects
public class Console
{
  // create a reader object that reads from command line
  private static BufferedReader reader =
      new BufferedReader (new InputStreamReader (System.in));

  // read a String value from the console
  public static String readString ()
  {
    String tmp = "";

    try {
      tmp = reader.readLine ();
    }
    catch (IOException ioe) {
      ioe.printStackTrace ();
    }
    return tmp;
  }

  // read an int value from console
  public static int readInt ()
  {
    int tmpInt = -99999999;

    try {
      tmpInt = Integer.parseInt (reader.readLine ());
    }
    catch (IOException ioe) {
      ioe.printStackTrace ();
    }

    return tmpInt;
  }

  // read double value from console
  public static double readDouble ()
  {
    double tmpDouble = -99999999.9;

    try {
      tmpDouble = Double.parseDouble (reader.readLine ());
    }
    catch (IOException ioe) {
      ioe.printStackTrace ();
    }
    return tmpDouble;
  }

  // read char value from console
  public static char readChar ()
  {
    char tmpChar = ' ';
```

```
    try {
      tmpChar = reader.readLine().charAt (0);
    }
    catch (IOException ioe) {
      ioe.printStackTrace ();
    }
    return tmpChar;
  }
}

/* Notice how the single attribute and all the methods in this
   class are declared static. This avoids the need to create a
   Console object in every program that requires interactive
   user input.

   Possible modifications :
   * The readChar method actually reads a String and returns
     the first character - this could be made more robust by
     checking that the String only has one character to start
     with.
   * The exception handlers produce very descriptive error
     messages (possibly _too_ descriptive) - the
     printStackTrace calls could just as easily be replaced by
     custom error strings.
*/
```


Further Reading, Web Sites and Other Resources
Bruce's Boomerangs
Guitar Licks
welcome t
DUCK i
seRvice
CHARLES DUCK
Floss
GO
Monty's Python
JAVA
Reference
DUCKY COOKIN'
CHRIS P. DUCK

Since we're now at the end of the book, the question rather arises of where to go next.[1] This section contains some ideas. Here you'll find pointers to books and web sites that should help you along the way as you learn more about programming and computing in general.

All the web sites in this section worked at the end of December 2003. The web is a fast-changing place, and it is quite possible that by the time you read this some of them will have disappeared or moved on. A good search engine should help you find out where it's one. If you find a web site that's not working, you can email me at `tony@tony-jenkins.co.uk`, especially if you've also found a decent replacement! All the web sites are also linked from this book's web site, together with any changes.

Most people's search engine of choice these days seems to be Google at `http://www.google.com/`, not least because it's refreshingly advertisement-free. That's the one I used to find the sites in this section …

Other useful search engines are:

AltaVista	`http://www.altavista.com/`
Excite	`http://www.excite.com/`
Lycos	`http://www.lycos.com/`
Yahoo	`http://www.yahoo.com/`

More about Java

The best place to start if you want to learn more about the Java language is probably the main page maintained by Sun:

```
http://java.sun.com/
```

You should be used to using the API documentation available from here by now, but there's also plenty of interesting information on the history of Java and plans for the future. The collection of tutorials:

```
http://developer.java.sun.com/developer/onlineTraining/
```

will also provide an introduction to some of the other parts of Java that you might want to go on to explore now.

There are many other Java books. The best way to choose the one that's best for you is to spend some time in a shop leafing through them.

You will have gathered by now that I'm quite keen on things that are free. Happily, there are free books on Java. The most widely used of these is probably Bruce Eckel's *Thinking in Java*; the third edition of this is available in an electronic form at:

```
http://www.mindview.net/Books/TIJ/
```

This book is also available in other programming languages. The C++ version would be a good place to start if you want to learn something of that language.

---

1 You may well fancy the pub, a nice café, Hawaii, or Basingstoke, but I'm talking Java here.

Famous people

The names of a few other famous people in the history of computing have been used in some of the examples. These are some people that you really should know something about:

Charles Babbage The "Grandfather of Computing". Built (or more accurately attempted to build) the first mechanical devices that shared many of the characteristics of modern computers. `http://ei.cs.vt.edu/~history/Babbage.html`

Grace Hopper A pioneer of programming and compilers. Curiously made the first computing *Man of the Year* award in 1969. `http://www.sdsc.edu/ScienceWomen/hopper.html`

Ada Lovelace Generally thought of as the first computer programmer, although she lived long before anything that we might recognise now as a computer was built. `http://www.sdsc.edu/ScienceWomen/lovelace.html`

John von Neumann A pioneer in computer architecture – the *von Neumann Architecture*. `http://ei.cs.vt.edu/~history/VonNeumann.html`

Linus Torvalds The man behind the free Linux operating system. A famous Finn. `http://www.tuxedo.org/~esr/faqs/linus/`

Alan Turing A founder of artificial intelligence, probably now most famous for his work at Bletchley Park in the Second World War. `http://www.turing.org/`

Maurice Wilkes Developed many fundamental aspects of computer hardware and programming. `http://ei.cs.vt.edu/~history/Wilkes.html`

As usual, a decent search engine will turn up many more pages of information about these people.

There are also many sites offering information about more famous people and more about the history of computing in general. The site at Virginia Tech is a good starting point if you want to find out more:

`http://ei.cs.vt.edu/~history/`

Readers' choice

That's it for further reading.

If you want more, you'll have to send it to me. On the book's web site you'll find a form to submit your favourite, useful, related-to-programming-in-some-vague-sort-of-way web site. Send it to me and I'll add it to the links. Your chance to get a little bit of fame!

Software

This section describes how to get hold of all the software you need to get started writing Java. The best part is that all the software described here is completely free!

All the software in this section can be downloaded free of charge from the web. If this causes a problem, you can get it all on a set of CDs as part of the

excellent Brighton University Resource Kit for Students (BURKS). Details of BURKS are at:

```
http://burks.bton.ac.uk/
```

BURKS includes a very large collection of software and other useful stuff, including a full distribution of Linux, a free version of the Unix operating system. The web site also contains a mirror of the contents of the CD, so you can download everything from there too. If you choose to buy the CD, please be sure to tell John that I sent you.

Most of the Linux versions of this software will also be provided as a standard part of one of the many distributions of Linux. Links to the web sites of the providers of all of these distributions are maintained at:

```
http://www.linux.org/
```

A good place to find out more about free software is the home page of the Free Software Foundation:

```
http://www.fsf.org/
```

Of course, a computer needs an operating system. Most free software is, very reasonably, written for free operating systems. In practice this means that most free software is written for Linux, and other free versions of the Unix operating system. Do not panic if you're a user of a less free operating system such as Microsoft Windows; there are usually versions available for these too (although sometimes they're not the latest version).

You can find links to all this free software on the web site. You'll also find a form to tell me about any other useful software for Java programming that you might find so that I can add it to the list.

Compilers

The first thing that you need is obviously a compiler and a JVM. This is easy enough to get hold of from the usual place:

```
http://java.sun.com/
```

The latest versions of everything you need can be downloaded from here.

Editors

Now for something to edit the source code with. In the Unix world the most popular editor is `vi`. A fine version of this is *Vim*, available for a vast range of operating systems (including Microsoft Windows) at:

```
http://www.vim.org/
```

New programmers often find the windows-based version of *Vim* (called *gvim*) especially useful. *Vim* has many features that make programming easier; it will highlight different statements in helpful colours, and it will take care of indentation automatically. All the programs in this book were written using either *Vim* or *gvim*.

Vim is not strictly speaking free software. It is *charityware*. If you download *Vim* and find it useful you are invited to make a donation to help needy children in Uganda.

A popular editor for Windows platforms is the Programmer's File Editor (PFE). Development of PFE has now been discontinued, but the last version can

still be found:

```
http://www.lancs.ac.uk/people/cpaap/pfe/
```

If you fancy something a little more sophisticated for your programming, you could try out an IDE. BlueJ is an IDE designed specifically for new programmers. It comes with a set of example programs and can be downloaded from:

```
http://www.bluej.org/
```

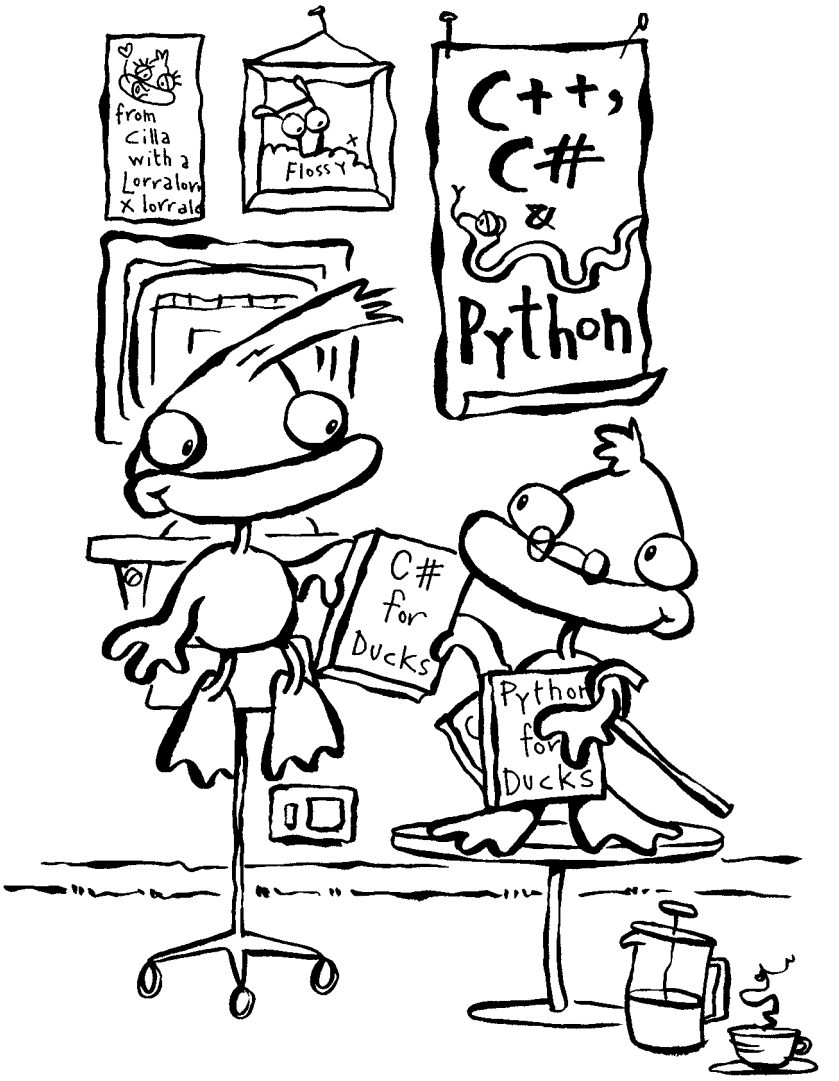
You can also download all the necessary Java system files from the BlueJ site.

This book

The programs for this book were developed using standard Java from a command line interface. The source code was created using *Vim*. All this happened using various versions of the Red Hat distribution of the Linux operating system.

The words you are reading now were typed using Microsoft Word 2000, running on Microsoft Windows 2000.

Getting the programs into and out of Word was a nightmare. It is not something that I would recommend. And this is the second time …


from Cilla with a Lorralon X lorrald
Flossy
C++, C# & Python
C# for Ducks
Python for Ducks

C++, C#, and Python

It's quite likely that you will soon want to move on to programming in another language. The principles that you have learned with Java should see you well placed to go on to program in other languages – the obvious candidates are two other object-oriented programming languages, C++ and C#.

This section also points you in the right direction to find out more about programming languages in general. In particular, Python is an interesting new language that is well worth a look.

C++

We've mentioned C++ a few times in this book, and we've also considered some of the differences between Java and C++. The main difference is that Java is seen by many to be more "pure" while C++ is more like a non-object-oriented language (C) with object-oriented features added.

With experience in Java, you should be able to recognise what simple C++ programs are doing. Much of the syntax is very similar; the designers of Java had obviously seen C++, and it made sense to keep some of the syntax identical. Here's the traditional first program in Java:

```
// hello.cc
//
// The traditional first C++ Program.
//
// AMJ 15/9/2003
#include <iostream>
using namespace std;
int main ()
{
  cout << "hello, world" << endl;
  return 0;
}
```

You can see that a C++ program does not have to use a class. But then some of the keywords used in Java are the same as those in C++, and you can probably take an educated guess at what's going on here.

The traditional second program written in C++ starts to show where the two languages are different, but there should still be plenty here that you can recognise or at least guess at the purpose of.

```
// hello.cc
//
// The traditional first C++ Program.
//
// AMJ 15/9/2003
#include <iostream>
#include <string>

using namespace std;

int main ()
{
  string name;

  cout << "Enter your name: ";
```

```
    cin >> name;
    cout << "hello," << name << endl;
    return 0;
}
```

C++ was a development of the C language, and both these languages are to some extent ancestors of Java. You can read more about C++ at the home page of its inventor, Bjarne Stroustrup:

http://www.research.att.com/~bs/

The home pages of Brian Kernigham and Dennis Ritchie provide a good bit of history of the development of C. The contribution of these two (K and R, as they are generally known) is hard to overstate. Both K and R work at Bell Labs in New Jersey and their pages are:

http://www.cs.bell-labs.com/who/dmr/
http://www.cs.bell-labs.com/who/bwk/

C#

You might also have heard of C# (pronounced C-sharp). C# is closely related to C++ and Java, and is worthy of quick mention here. C# is a proprietary product of Microsoft; it is the programming language associated with Microsoft's .NET initiative. Like Microsoft's other visual programming languages, C# is written using a complete IDE (you have probably seen or used tools such as Access or Visual BASIC).

C# is quite a controversial topic. Some have argued that it is simply Microsoft's attempt to compete with Java; Microsoft obviously has a proprietary operating system and could be expected to be worried by a language that would run on many different systems. Some have gone further and suggested that C# is little more than a copy of Java made specific for the Microsoft platform. Others, on the other hand, state that the emergence of C# and Java at roughly the same time is little more than coincidence. There are plenty of people who will argue for each of these.

You can make your own mind up. Microsoft's own site contains all you could need to know about C#, as well as handy pointers on where to buy it:

http://msdn.microsoft.com/vcsharp/

A search on any web search engine will lead you to many of the arguments!

Python

A final language worth a mention in its own right is Python. Python is, like Java, a platform-independent language and implementations of Python are free. You can find out all about it and download a copy at the Python web site:

http://www.python.org/

In particular the Beginners' Guide at:

http://www.python.org/doc/Newbies.html

will help you on your way to learning some Python. There is also an excellent free book by Jeff Elkner and others describing all the Python you might need available:

http://www.ibiblio.org/obp/thinkCSpy/

and an equally excellent free tutorial by Alan Gauld here:

```
http://www.freenetpages.co.uk/hp/alan.gauld/
```

Python differs from Java and C++ in that it is a *scripting language*. This means that Python programs are not compiled, but are interpreted as they are executed. Python programs can also be written interactively.

Python is quite a new language, but it is gaining in use. A particular attraction is that, since it is interpreted, it is especially suitable for developing quick prototypes of programs before they are rewritten in another language. Many recent applications have used Python, including the Google search engine and several feature films.

The history of Python is described on the home page of the author, Guido van Rossum:

```
http://www.python.org/~guido/
```

Other languages

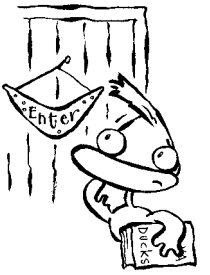
Computer programming languages are rather fascinating things. They have developed a complex family tree, even though they've only been around for a relatively short time. If you want to find out more about them, a good start is Éric Lévénez's excellent programming language page:

```
http://www.levenez.com/lang/
```

Equally fascinating is the splendid "Bottles of Beer" page at:

```
http://99-bottles-of-beer.ls-la.net/
```

where you can find programs to generate the lyrics to a well-known song in (as at December 2003) 598 different programming languages. That's 170 new languages in the last year!



Glossary

accessor A *method* which can inspect the value of an *attribute* but not alter it. Also called a *selector*.

algorithm A group of operations which perform a certain task (such as *sorting*). May be written in a particular programming language, or *pseudocode*.

Application Programming Interface (API) The list of *public methods* and *attributes* describing, for the benefit of a *programmer*, how a particular *package* or language can be used. Usually provided in the form of vast amounts of documentation.

argument See *parameter*.

array A structure for storing a fixed number of values of the same *type*.

array list A structure, similar to an *array* that can hold a variable number of values of some *type*. See also *vector*.

assignment operator =, the operator used in an assignment statement. Not the same as ==, which is the *comparison operator*.

assignment statement A statement where a value is assigned to a *variable*.

attribute A value corresponding to a feature of an *object type*, which is of interest in the real-world problem we are trying to represent. An example would be the "name" attribute of a "person" object.

backslash The "\" character, found between the left-hand shift key and the Z key on the UK keyboard.

base class A class from which *derived classes* inherit *attributes* and *methods*.

beta testing The testing of a *program* at an advanced stage of development, in order to find *bugs* which would only become apparent through "real" usage.

black box testing A form of testing where the actual *implementation* of the *program* is unknown.

body A collective name for the statements inside a method.

Boolean expression An *expression* which can be evaluated to one of true or false.

Boolean operator An *operator* used to make or compare *Boolean expressions*.

Boolean value A value which can be evaluated to either true or false.

boundary case (or boundary condition) In testing, a type of *test case* which tests the range of values containing a change from acceptable to unacceptable.

braces See *curly brackets*.

Bubblesort A simple sorting *algorithm* commonly used for teaching purposes.

bug Unintended or undocumented behaviour in a *program*. Usually a bad thing, and almost always discovered by a *user*.

bytecode Generated by a *compiler* from *source code*, and executed by an *interpreter*. Java bytecode is found in files with the `.class` *extension*.

call Both a noun and a verb. See *method call*.

calling method A *method* which *calls* another *method*.

calling program A *program* which *calls* a *method*.

cast A way of converting a value from one *type* (usually a *numeric type*) to another.

CLAs See *command line argument*.

class See also *object type*.

.class file See also *bytecode*.

classpath A list of locations the Java compiler and/or interpreter will search for packages imported into classes or programs.

code See also *source code*.

code reuse The ability to use *code* developed to solve one problem as part of a solution to another.

code walkthrough The process of going through a *program*'s *source code* a line or block at a time, in order to explain its operation to another person.

command line argument A value provided as input to a program when it is run, via the command line.
comment A piece of *code* which is ignored by a *compiler*, but useful for providing information to *programmers*.
compilation The process of translating *source code* into an *executable* form.
compiler A *program* which performs *compilation*.
concatenate To tag one value onto the end of another. For example, concatenating the *strings* "Z" and "Cars" produces another *string*, "Z Cars".
condition An assertion which is either true or false. Used in *conditional statements* to control the *flow* of a *program*.
conditional statement A *statement* which evaluates a *condition* or set of *conditions*, and based on the result of this evaluation, passes control to various different parts of a *program*.
constant An entity whose value will never change during the life of a *program*. Has a *type*, and is referred to via an *identifier*.
constructor A special *method* which is used to *instantiate* an *object* of a given *class*. Must be declared *public*.
control statement A *statement* which controls the *execution path* of a *program*.
control variable A *variable* used in a *control statement* which may affect the *execution path*.
curly brackets (or curly braces) The symbols { and } which signify the start and end of a block of statements in Java. Not to be confused with parentheses – (and).
data hiding The process of writing a *class* in such a way that the *programmer* need have no knowledge of the *attributes* used in its implementation.
data type The sort of value which can be stored in a given *attribute* or *variable*. Examples are *integers*, *strings*, *arrays*, and *floating-point numbers*.
debugger *Program* used to aid *debugging*.
debugging The process of systematically discovering and eliminating *bugs* in a *program. Programs* to help in this process are known as *debuggers*.
declaration A *statement* which introduces a *method, constant,* or *variable* into a *program* or *class*.
decrement To reduce the value of an *integer variable* by one.
derived class A class which inherits *attributes* and/or *methods* from a *base class*.
determinate loop A control *loop* which executes a predefined number of times. The eventual number of iterations is known at the time of the first iteration.
directory The Unix and MSDOS equivalent of a *folder* on an MS Windows system.
driver A *program* written especially to test a *class*.
dry run Simulating the execution of a *program* with pencil and paper.
editor A *program* used to create and alter *source code*.
element A value in an *array* or *vector*, accessible via its unique *index*.
exception An unexpected run-time error in a Java program.
executable A version of a *program* which can be run by a computer.
execution path The order of *statements* executed during the lifetime of a *program*. Each different execution path will cause the *program* to produce different results.
expression A combination of *literal values*, *operators*, and *variables*.
extension (or file extension) The part at the end of a filename (usually following the last dot) which is used by some *programs* and *operating systems* in an attempt to determine the file format.
flag A special *command line argument*, usually denoted by a / in DOS or – in Unix.
floating-point number A number which includes a decimal part.
flow (of control) The sequence of *statements* executed in a *program's* lifetime. Affected by *control statements* and *conditional statements*.
folder A set of files grouped together into one logical unit.
function A procedure for achieving some result, usually corresponding to a *method*.
hard copy *Source code* printed onto paper. Useful for testing purposes and *code walkthroughs*.

header block (or history block) *Comments* found at the top of a *source code* file, containing information about who wrote the *class* or *program* and when, plus any important changes made to the *code*, and usually a short description of its purpose.

high level programming *Programming* in a language such as Java, using a high level language that is easy for humans to understand.

identifier The name used to uniquely identify a *variable* within a *program*.

increment To increase the value of an *integer variable* by one.

indentation Use of *whitespace* to make *source code* easier for a *programmer* to read.

indeterminate loop A control *loop* which executes until a given *condition* is satisfied. The eventual number of iterations through the *loop* cannot necessarily be determined at the time of the first iteration.

index A variable's position within a *string*, *array* or *vector*.

infinite loop A *loop* whose exit *condition* can never be satisfied, and therefore executes forever. This is almost always *programmer* error.

initialiser list A list of values used to initialise an *array*.

instantiation The process of creating an *object* or *variable* of a certain *type*.

integer A number with no decimal part.

interactive (or integrated) development environment (IDE) A piece of software which combines an *editor* and *compiler* (and usually a *debugger*) to aid in the rapid development of *programs*.

interface The specification of a *class's methods*, indicating how a *programmer* can use that *class* in a *program*.

interpreter Reads *bytecode* and instructs the computer to perform some tasks.

iterate To repeat a process until some condition is met. To process every element of an *array* in turn, for example.

jar file (or jar archive) A sort of file containing a *package* or a group of packages, bundled together for convenience.

.java file Text file containing the *source code* for Java *classes* or *programs*.

Java Development Kit (JDK) A software bundle containing a Java *compiler*, *interpreter*, *debugger*, and other assorted tools, plus a whole host of predefined *classes*, and usually copious amounts of *API* and tutorial documentation. Sometimes referred to as the *SDK (Software Developers' Kit)*.

Java Virtual Machine (JVM) An *interpreter* for the Java programming language.

library A group of *classes* provided by other *programmers*, offering useful functionality. The Java term for this is a *package*.

list A structure, an *array list*, for example, used for holding a collection of related objects.

listing See *hard copy*.

literal value A simple value used in an *assignment*. For example, any *string* value, or 1, or 3.142, or the character "c".

local variable A *variable* created within a *method*, which only exists for the lifetime of that *method*.

loop A block of *statements* executed repeatedly. The number of executions is controlled by a *control statement*.

low level programming Programming at the machine level, perhaps using switches and levers.

method A block of code which corresponds to a particular task in the real-world problem we are trying to represent. An example would be a code to alter the name of a "person" *object*.

method call A *statement* which causes a *method* to be executed.

methodology A particular way of carrying out a task. In programming terms, this could be a way of analysing a problem, or a way of testing a *class*.

modulus The remainder operator, represented as %.

mutator A *method* which is used to alter the value of an *attribute* of an *object*.

numeric type A primitive *variable type* representing a mathematical quantity. Examples are *integers* and *floating-point numbers*.
object An entity representing a particular instance of a *class*, which can be manipulated by a *program*.
object-oriented programming A style of programming which relies on descriptions of *objects* to represent a problem area.
object type Refers to a group of objects which possess the same characteristics within a problem area.
operating system (OS) The *program* (or collection of *programs*) which provides the interface between *user* and computer. Common examples are Linux, MS Windows, and Mac OS.
operator A symbol used to build *expressions* along with *identifiers* and *literals*. For example, `/`, `*`, `-`, and `+` are mathematical operators, while `!`, `&&`, and `||` are *Boolean operators*.
operator precedence The order in which *operators* are treated within a calculation or comparison.
ordinal type A *data type*, the possible values of which can be listed.
package The Java term for a *library*.
parameter A value provided to a *method* by its *calling method*. Found within *parentheses* in a *method call*.
parentheses The symbols (and).
portable Property of a *program* which can be copied across different computers and *operating systems*, and *compiled* and run with little or no alteration required.
precedence See *operator precedence*.
primitive type One of the basic built in types such as `int`, `char`, `double`, `boolean`, as opposed to *object types*.
private A kind of *attribute* or *method* which cannot be accessed directly by any code outside of the *class* in which it is defined.
procedural language A *programming language*, like Java, that specifies a procedure that will solve a problem; some other languages can be described as functional.
program A set of *statements* which, together, perform some task.
programmer A skilled craftsperson who develops *programs*. The term is not necessarily interchangeable with *software engineer*.
programming language A specialised language used by a *programmer* to express an *algorithm* to a computer.
prompt A message displayed by a *program* indicating that *user* input is required.
prototype A name sometimes used for the heading of a *method* declaration.
pseudocode A representation of an *algorithm* which is not written in any particular computer language. It usually consists of more human-oriented language rather than computer-oriented, and is written in such a way as to be clear and unambiguous.
public A kind of *attribute* or *method* which can be accessed directly by code outside of the *class* in which it is defined.
recursion See *recursion*.
reference Property of a parameter, for example, where the parameter represents not a particular value (such as "1878"), but a particular object (for example, the duck "Elvis").
return The last stage in the execution of a *method*, where the *flow of control* reverts to the *calling method*. Sometimes involves a *return value* being provided to the *calling method*.
return type The *data type* of the value returned by a *method*.
return value The value passed from a returning *method* to its *calling method*.
scope Of a *method* or *attribute*, whether the *method* or *attribute* is *public* or *private*.
selector See *accessor*.
Software Development Kit (SDK) See *JDK*.

software engineer Skilled craftsperson who engages in *software engineering*. Not necessarily interchangeable with the term *programmer*.
software engineering The process of creating working software, including programming, testing, evaluation and documentation.
sorting algorithm An *algorithm* used to sort a collection of values into some order, determined by some rules (for example, sorting a list of names into alphabetical order).
source code A sequence of characters which are meaningful to both a *programmer* and a *compiler*.
statement A line of code which performs some task within a *class*.
static A *method* or *attribute* of a class which does not require an *object* of that *class* in order to be used.
string A *data type* representing a sequence of characters.
style The way in which a *programmer* lays out the *source code* of a *program*. Relies on *indentation*, and it is important that this is consistent.
test case (or test condition) A set of input values and their corresponding output values, used as part of a *test plan* to evaluate the correctness of a program.
tester A person, usually a *programmer* or a close associate, who applies a *test plan* to a *program*.
test harness A program used to test the implementation of a *class*.
test plan A set of *test cases* which, if all satisfied, provides sufficient evidence that a *program* will work as intended.
transaction Something that happens in a problem that changes some value; usually corresponding to a *method*.
truth table A visual representation used to show all the possible outcomes of a *Boolean expression*.
type See *data type*.
typical value A *test case* representing a (usually random) value which may be provided to a *program* by a *user* during the course of normal execution.
user Person who utilises a *program* written by a skilled *programmer*, and invariably discovers unexpected ways to break it. Very good at uncovering *bugs*.
variable A value which may change during the lifetime of a *program*.
variable declaration A *statement* where a variable is created. Includes the variable's type and *identifier*, and optionally an initial value.
vector Similar to an *array*, but can grow and shrink during the lifetime of a *program*. See also *array list*.
void method A *method* which does not *return* a value to its *calling method*.
white box testing A process of testing a *program*, involving examining its *source code*.
whitespace Spaces, tab characters, and newlines, used to make *source code* easier to read. See *indentation*.


INDEX!
Alphabetically...
Bruce
Cilla
Don
Elvis
...Zoot

Entries in `monospaced font` are legitimate Java terms, while entries in *`italicised monospaced font`* are Java terms which, although used in this book, are not part of the standard API. Page numbers in **bold type** are those especially recommended for that particular topic.