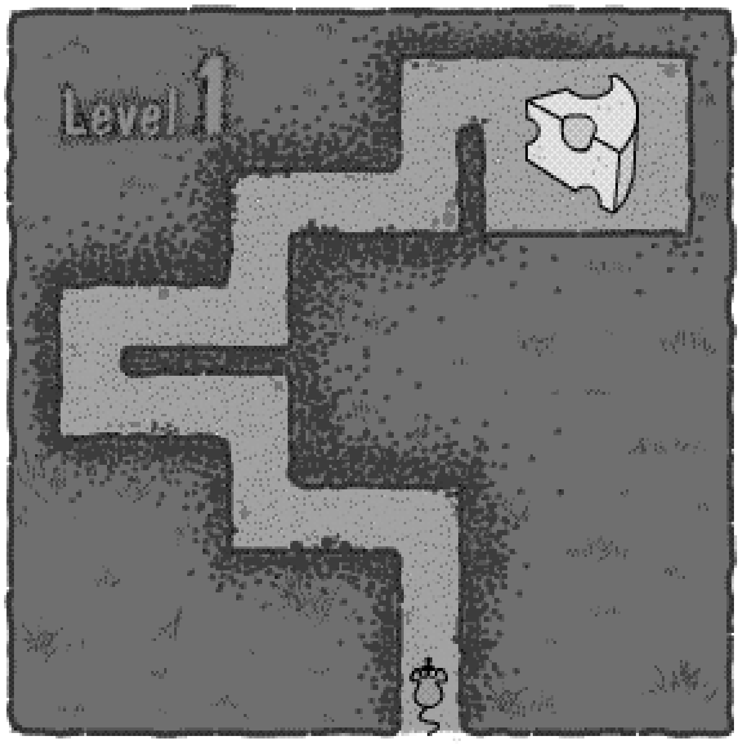
The Guide to

Programming for Computer Scientists (CS118)



Term 1, 2005-2006

Dr Stephen Jarvis

Contents

Chapter 1

Preface

1.1 What is this course all about?

The Programming for Computer Scientists course is designed to give you knowledge and confidence in using a computer as a scientific tool. During the course you will have a chance to work on control software. You will get to spend a good deal of time solving software problems; your solutions to these problems will be implemented on a computer and run in order for you to check the resulting behaviour.

Despite the slightly misleading title, Programming for Computer Scientists is not simply about ‘programming’. Problem solving using computer software involves three important steps - design, build and test. The exercises in this course encourage you to look at each of these steps in turn. Each step must be carried out successfully if the software which you are going to create is to be correct.

The laboratory exercises for this course are set in the context of getting a robot to travel through a maze. This is what you will program; you will be able to see your progress as the robot will either reach the end of the maze or not.

By the end of the course you will have learnt a number of skills: You will have gained a great deal of experience in the art of software development, you will also know what it means to plan and develop sophisticated Java code. The exercises also touch on other interesting areas of computer science such as games programming, data structures, algorithmics and artificial intelligence.

1.2 Course structure

1.2.1 Lectures and seminars

The course consists of two lectures per week (Monday 1-2pm and Thursday 3-4pm, both in L3). The lectures are important. You cannot possibly know all the material which the lectures will cover - even if you are a ninja programming guru - and the exam at the end of the course will cover that material which was introduced during these lectures. Lecture notes are supplied, but you will probably find that these do not make complete sense outside the scope of the lectures - so don’t miss the lectures!

The first lecture will take place on Thursday of week 1 in room L3.

You are also required to go to one seminar per week[1]. The seminar groups are posted in the reception area of the computer science building (and also on the course web page); you should check the lists to find out when and where your first seminar is. **The seminars are scheduled to begin on Thursday of week 1 (from 5pm).**

The seminars serve two purposes. Firstly they allow you to interact with a seminar tutor who will be an expert in the subject area and who will be able to guide you through any difficult moments. Make good use of your seminar tutor as they are there to assist you and answer any questions which you may have concerning this course. Secondly the seminars are designed to test you, to make sure that you are keeping up with the course material. To do this there are five problem sheets which you must complete.

The seminars and problem sheets are not optional. You will have to sign in at each seminar and you must hand in your answers to you seminar tutor 24 hours before your seminar begins. To do this you should prepare your solutions in advance and post them in the CS118-marked submission letter box at the back of computer room 006 (in Computer Science). At the end of the course your attendance and your submission of seminar solutions will be noted and will contribute towards your final grade for this course.

Problem sheet 1 can be found in the 'Getting Started' chapter of this guide. The remaining problem sheets can be found in Appendix A.

1.2.2 Coursework

This course is not merely an introduction to programming. Rather it is intended to give you a modest taste of what problem solving using computer software is all about: combining creativity with a rigorous, analytic approach to produce an end result which can be relied upon to achieve its design goals. The fact that you will learn about the Java programming language is simply an added bonus. This being the case, it is not the purpose of the course to teach you about all the facilities of the Java programming language; you may decide that further reading is useful to your studies, but the assessment of your programs will be based on their correctness and clarity, rather than their 'high-tech' programming style.

There are two pieces of coursework. The first coursework must be complete by **Wednesday 26 October (Week 5)**; the second must be complete by **Friday 2 December (Week 10)**. Your work will be assessed as follows:

you will be required to come to the IBM lab in the Computer Science Department at set times on the 26 October and 2 December to demonstrate your working program; here you will be interviewed, during which you will be asked to explain how your code works as well as being tested on some of the fundamentals of the coursework.

This may sound a little heavy, but I can assure you that if you have done the necessary study and produced your own independent solutions then you will not have any problems.

A record of your work will be stored by the University of Warwick BOSS online submission system. This software allows us to record your progress, justify the mark which

[1]Don't be confused by the fact that your timetable might have more than one CS118 seminar slot marked on it. All this means is that they have printed all the seminar times, only one of these will apply to you.

you were awarded and detect plagiarism. Using this system also means that you don't have to hand in loads of paper, which is good for everyone.

More information regarding BOSS can be found at

`www.dcs.warwick.ac.uk/boss/`

1.2.3 Class test

There will be a class test on **Monday 31 October**. This will take place in the usual lecture theatre and will include questions on all the course material up to that date. Your mark for this test will contribute towards your final grade, so it is important that you turn up to the test.

1.2.4 Adding it up

The coursework, seminars and class test will account for 40% of you final mark for this course. The remaining 60% of the marks are taken from your two-hour end of year exam which usually takes place in week 2 of term 3.

1.3 Computing support

1.3.1 User codes

Before you do anything computing-like, you need to get hold of a user code. Rather confusingly you will be allocated two user accounts, one by the University IT services and one by the Department of Computer Science[2].

The first thing you should do is get your IT services account. The University of Warwick's Online Enrolment System is now well established. This should mean that when you collect your enrolment certificate (by going to `www.warwick.ac.uk/enrolment`) you will also be asked to complete the on-line Computer Use Registration form, and as a result will receive a username and password. This will give you access to the University's IT network and facilities; it will not however give you access to the computers in the Computer Science Department.

Sometime after setting up your IT services user code and account, you will be emailed with your Computer Science account details[3]. You should therefore check the mail on your IT services account and when these details arrive you should make a note of your CS user code and password and bring these with you the next time you come to the Department. You are then ready to log on to the machines in Computer Science. Please remember that the IT services and Computer Science accounts are different and although they might have the same user code, they may well have different passwords. Please try not to forget your password, you will find that it is a real pain if you do[4].

[2]If you are doing a course such as Maths and Computing, you will probably end up with three accounts; if you are doing Maths, Physics and Computing, may be you end up with four!

[3]You should expect this to be done before your first CS118 seminar.

[4]This year there is a £8000 charge for resetting your password.

If all this business of getting an account seems too confusing to be true, do not panic. Ask around to see if anyone else has worked it out and talk to them. Failing that, ask one of the second years. Failing that, ask me.

1.3.2 Course web page

All the information in this guide (and more) can be found on the course web page:

`www.dcs.warwick.ac.uk/people/academic/Stephen.Jarvis/cs118/`

It is certainly worth bookmarking this page in your favourite web browser as it will be very useful as the course progresses. The course web page contains:

- This *Guide* to CS118;
- All the lecture notes as they become available;
- Copies of the seminar sheets;
- A lecture summary, course overview and course syllabus;
- Trouble shooting information and handy hints for coursework and seminars;
- General systems information (including guides to UNIX etc.);
- Java resources including the robot-maze software.

Do take a look at the course web page from time to time as it will be updated periodically.

1.3.3 Java sources

Both the IT services computers and also the Computer Science computers have copies of Java running on them. If you want to get hold of a copy for your home computer then this is very easy. I would recommend that you get a copy of Java 2 version 1.5[5] as this is the latest version. A download can be found at:

`http://java.sun.com/j2se/1.5.0/download.jsp`

There are Microsoft, Linux and Solaris versions of the software at this site and I have found that installing the software is easy and should not take you more than about ten minutes. Make sure you download *J2SE(TM) Development kit 5.0 Update 5* as again, this is the latest version.

I usually run Java on my laptop via a *Command prompt* window that you will find on most versions of Microsoft Windows. The advantage of this is that the set-up then looks just like a terminal window on one of the computers in the Department and all the commands that you type in to each are the same. You will probably find that Java installs to the directory `c:\Program Files\Java\jdk1.5.0_05\bin`.

[5] If you are running Java J2SE 4.2 then this will be fine.

The Java package for the robot-maze software which you will use during your coursework can be found on the course web page. You can download all the necessary files to the `c:\Program Files\Java\jdk1.5.0_05\bin` directory of your home computer and then use them as described in these notes[6]. This is not something that you need to do straightaway, but might be useful later in the course.

1.4 Books

Programming is a complicated business and there is no substitute for a well-written introductory text book when you are starting out. Each year I try to select the most appropriate Java books for this course. In making the selection I try to ensure that the books will be understandable, will reflect the range of talents and abilities of the people on the course, and will be useful later in your degree.

This said, it became apparent last year that many of the course books that you were recommended cost a small fortune; the price of a small car to be precise. So this year we have struck a deal with one of the publishers (Pearson) so that they are now offering a *value pack*[7] whereby you can pick up a Java book (the Liang book below) and two further books (Weiss - *Data Structures and Problem Solving Using Java* - £42.99, and Ayres - *Essence of Professional Issues in Computing* - £22.99)[8] from the University book shop for a little more than a night on the town (£87.99)[9].

Anyway, if I were going to buy a good book from new then I would pick up one of:

- *Introduction to Java Programming Comprehensive* by Y Daniel Liang, published by Prentice Hall, ISBN 0131489526 (£60.99 on Amazon.co.uk)[10]
- *Understanding Java* by Barry Cornelius, published by Addison Wesley, ISBN 0-201-71107-9 (£40.99 on Amazon.co.uk or £20.99 second-hand)
- *Core Java 2: Volume 1-Fundamentals* by Cay S. Horstmann and Gary Cornell, published by PH PTR and Sun Microsystems (£27.99 on Amazon.co.uk or second-hand from £19.18)

[6]Everything will work fine if you just stick to using the `bin` directory, but probably what you really want to do is set `PATH` and `CLASSPATH` variables so that you can access the Java compiler from any directory. There are lots of web links that tell you how to do this, one of which is `http://www.cs.uscb.edu/~teliot/Path_and_Classpath.htm`

[7]Don't get too excited...

[8]That you will need for some other courses this academic year.

[9]Depending on your constitution of course.

[10]I like this book very much, but here is some advice. This is about the 900th edition of this book and it is the most expensive, which must mean that he is selling fewer books (but his mortgage is still the same). This means that there are loads of second-hand books floating around. The hard up second and third years will advertise these on the CS118 forum (see later notes); you can also get second-hand copies from Amazon. At the time of writing I could pick up an International Edition for £30, a Comprehensive edition by Liang and Frandsen for £29.99, a United States Edition (big writing, no words longer than four letters) for £9.96, and an old, but nevertheless useful original Introduction to Java Programming for £6.67. Please do not buy a copy for £61; although do try and get a fairly up-to-date version, i.e. for Java 2, 3, 4 or 5. If you need a new one then go for the value pack. If you don't mind buying a second-hand one then shop around. There will be a prize for the person who picks up the cheapest copy.

The Liang book follows the lectures very closely so is your best bet if you are feeling a bit unsure about this programming lark.

If you are already an established programmer and have spent, for example, a couple of years programming in C or VB then you might like to buy the Cornelius book. He explains things very well, though not necessarily in the same order as my lectures.

Finally if you already know Java and do not already have a copy, buy the Horstmann and Cornell book. You get a lot of pages for your money, and it's size alone will be enough to impress all your friends.

You may already have a Java book. I guess the only reason that you might not want to use this is if it is using Java 1.0. This is now pretty old and there are enough differences to warrant splashing out and buying a new book. If you have a book that deals with Java 2 then you should be fine.

1.5 Practicalities

If you have any problems regarding CS118 matters you can come and find me in my office (2.07) in the Computer Science department.

My email address is `stephen.jarvis@dcs.warwick.ac.uk`.

You will probably find that many of the issues which concern you are also experienced by others on the course. The CS118 forum is an excellent term-time newsgroup for discussion. This will also be the first place that important events/dates etc. are announced.

1.6 And if it all starts to go wrong?

You may get to a point in this course where you just don't know what is going on any more. The golden rule is not to panic, get depressed or start supporting Arsenal football club[11]. There are a number of procedures in place to catch you before you fall, but you have to be pro-active in finding help.

This year we are very lucky to have a lady called Zabin Visram to organise troubleshooting classes. These will be held outside of the usual seminars and can be on any topic of your choosing. If you are particularly confused about a topic after attending the lectures and seminars, and having read the books, then you can email Zabin for some additional help. Her email address is `zabin@dcs.warwick.ac.uk`.

The seminar tutors are usually pretty good at answering questions. You can find them out of hours by asking the receptionist in Computer Science where their offices are. As long as you don't pester them every five minutes, I am sure they would be pleased to help out. If you can not find them, you can of course come and find me.

[11]Or Manchester United come to that. Still, with loony Rooney in the team we do not need to worry about them winning the Premiership.

1.7 Acknowledgements

The robot-maze software which you will use in your coursework has an interesting history. It started life at Queen Mary College, University of London as the brainchild of Ian Page. When Ian moved to Oxford he and Colin Turnbull made extensive rewrites and the robot-maze to this day exists as a means by which engineering students learn the C programming language. Kevin Parrott, Alison Noble, Andrew Zisserman and myself ran this software largely untouched at Oxford for a number of years. The new Java version of the software is thanks to Phil Mueller from the University of Warwick (now himself at Oxford), and the new exercises are courtesy of Ioannis Verdelis (formerly of Warwick and now at Manchester University). I think you will agree that the result is a wonderfully interactive way in which to learn the art of programming.

Chapter 2

Getting Started: Problem Sheet 1

This chapter is intended for use during your first seminar session. Seminars will take place both in the programming labs and also in the class room. It is important that you know where you should be each week. **For your first seminar you should meet in the reception area of the Computer Science department building**. You seminar tutor will meet you and take you into one of the programming labs where you will spend the hour working through the exercises in this chapter. If you are having problems with user accounts then this is a chance to sort these out.

2.1 UNIX

All the computers in the Computer Science building run the UNIX operating system. This is the third year that Red Hat, the version of UNIX that we use in Computer Science, has been running on all the computers. This is good news as it means that as you move between computer rooms everything should look pretty much the same.

You will certainly need to pick up some UNIX skills while you are at Warwick. One way to start is to buy a copy of the excellent *Introducing UNIX and Linux* by the very distinguished authors Joy, Jarvis and Luck[1]. Indeed you might like to pick up a few copies for Christmas presents, keep-sakes, birthdays, anniversaries, gifts for girl/boy friends etc...

♡ I would suggest that you have a look at this book and study the chapter titled 'Getting Started'. I would personally ignore the direction to the text editor known as *vi* and also the section on electronic mail using *mailx*. I did not write these sections and you will find out about better alternatives later in this guide.

2.1.1 Login

When you log in to the computers you will enter the Red Hat windows environment. There are a number of other *Session* environments that you can select from the log-in screen, but this guide is written from the point of view of the *Default* session and so I can assume no responsibility for problems outside of this domain.

[1]*Introducing UNIX and Linux*, Mike Joy, Stephen Jarvis and Michael Luck, Palgrave MacMillan, 0-333-98763-2.

2.1.2 Mail

♡ There are a number of ways in which you can read mail in this environment. I'll list three options below (and suggest you go down the first or second route). One thing to note is that because this account is different from your IT services account the mailboxs will also be different; so if someone mails your CS account, don't expect it to show up at your IT services account and vice versa.

- *Route 1 - using Thunderbird:* You can read your mail in your Computer Science account through the Thunderbird mail client. Invoke Thunderbird by either typing **thunderbird &** in a terminal window or by clicking the **redhat** icon at the bottom left of the screen, and then following the links to the **Internet** and then selecting **Thunderbird Email**.

  The first time you run this mail client you will be asked for any import settings; these you don't need so you should just select the *do not import* option. Now you will be directed to the *Account Wizard* to setup a new account. Select the **Email account** option and click *next.* You are then required to enter your name and email address, here you should just enter the details you were given when you obtained your email address from the Department, then click *next.* In the next screen enter the server information as follows: Select **IMAP**, enter **mail.dcs.warwick.ac.uk** as both the incomming server and outgoing server and click *next.* Next, check that your incomming *User name* is correct. Then name your account and click *next* to review your settings and finish.

  After asking for your password, Thunderbird should now be ready to use.

- *Route 2 - KMail:* If the above seems like too much work then you can use the simple mail tool called *KMail.* You can start this by clicking on the *kmail* icon on the desktop. You might like to send and receive a few emails just to get used to using this mail system.

- *Route 3 - a text-based tool:* If you do not like the browser option or *KMail* then you can use a text-based mailtool known as *pine.* You can run *pine* by first invoking a terminal window (click on the button that looks like a screen) and then type `pine` in the terminal window which appears.

2.1.3 The Internet

♡ You can invoke a web browser (the default is Mozilla Firefox) by clicking on the globe-and-mouse button at the bottom of your screen.

If you would like to use a different browser from Mozilla then click on the red-hat button, select *Internet* and then *More Internet Applications.* Here you will find other browser options such as Konqueror etc.

While you are here you should find the course web-page.

```
www.dcs.warwick.ac.uk/people/academic/Stephen.Jarvis/cs118/
```

I would suggest that you *Bookmark This Page*.

2.1.4 News and forums

♡ As well as reading your department email you should also keep an eye on the *forums*. These are used as an efficient alternative to email; all important news and discussion are to be found on the forums. You may wonder why these are used as well as email. The reason is simply that if you want to send an important message to everyone in the department then you could email it to everyone, which would mean about 1000 copies of the same mail stored over 1000 different mail boxes, or you could send a single copy to a forum and then ask everyone to read it. The department authorities obviously prefer the latter.

Reading news via the forums

You can reach the Warwick forums from the University web page. Go to `www.warwick .ac.uk` and click on **Forum** on the left-hand side. At the bottom-right of this page you will see a link to **Warwick Forums**, follow this and then select **Departments** and **Computer Science** and **UGyear1** (for the first year newsgroups)[2].

You should make sure that you read the `CS118` forum and also the `misc` forum. You can shortcut to these pages using the following URLs:

`http://forums.warwick.ac.uk/departments/computer-science/ugyear1/cs118/`

`http://forums.warwick.ac.uk/departments/computer-science/ugyear1/misc/`

2.1.5 Editing files

Red Hat contains a number of text editing tools. One of these, called *kedit*, is available from the desktop and will be our editor of choice. Other editing tools can be found from the Red Hat window manager, or run from a terminal window (such as `xemacs`, which is one of my favourites).

2.2 Editing, compiling and running Java code

♡ Create a new file which contains the following text:

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

[2]At some stage you may be asked to log in. You should use your IT services user code (i.e. your University account) to do this

Make sure you copy the program exactly (the capitalisation is important). Save the program as `Hello.java`

♡ Now compile the program. To do this you need to open a terminal window (select the screen button from the window manager) in which you should type:

```
javac Hello.java
```

If the Java compiler detects any errors, then you must have typed the program incorrectly. Correct any errors in the program by re-entering the editor, making corrections, and then recompiling.

♡ Run the program by typing:

```
java Hello
```

The program should display `Hello World!` Congratulations! You have just learnt how to edit, compile and run a Java program[3].

2.2.1 Some easy programs

♡ Edit the `Hello.java` program again and change the message so that it says something else. You should be able to do this without knowing any Java. Save, compile and run it.

♡ Create a new file called `Age.java`. Type the following program exactly as it is written:

```
import uk.ac.warwick.dcs.util.io.IO;
public class Age
{

  public static void main(String[] args)
  {
     int age = IO.readint("Enter your age: ");
     int doubleAge = age * 2;
     System.out.println("How old? That's half way to "
                             + doubleAge + "!");
  }

}
```

[3]It is some bizarre tradition that when learning a new programming language, the first thing you start with is the 'Hello World!' program. I have no idea why this is the case, but far be it for me to break from decades of tradition...

You should note that when you make a new Java file it must always end with `.java` and the name which follows the `public class` in the program file must be the same name as the file.

This program uses the `readint` method of the `uk.ac.warwick.dcs.util.io.IO` class. This class is written especially to make input easier for you. You will not find `readint` in any of your textbooks, but don't worry, you will learn about it in lectures. Can you work out roughly how the program works?

You may get some warning messages appear in the terminal window when you run this program. If this is the case then just ignore them. This is something to do with the Java set-up that we will not have to worry about later in the course.

♡ The marks for this course might finally be weighted as follows: Coursework 30%; seminars and written test 10%; final exam 60%.

Write a program called `TotalMark.java` that reads in the three marks (each out of 100) and writes out the total weighted mark[4].

2.3 Web-based course material

♡ All the course material is available on the Web. Get a web browser running on your computer and take a look at the page:

```
www.dcs.warwick.ac.uk/people/academic/Stephen.Jarvis/cs118/
```

Add this page to your bookmarks.

If you have not done so already then you might also like to try and read the CS118 forum and post an introductory message[5].

It is strongly recommended that you all spend several hours at a terminal during the first two weeks of term getting familiar with UNIX, the text editor, the forums and mail etc. In any event, you should also be spending several hours per week writing Java programs. Copy examples out of your textbooks, customise them for your own use and experiment by trying to write programs of your own. Learning to program is impossible without the practical experience that is gained only by sitting down at a computer and doing it.

[4]There are a couple of things which you might like to investigate for this question - try and find out more about *Numeric Datatypes* and *Numeric Operators* from the lecture notes or textbooks etc.

[5]Name, age, shoe size, earning potential, etc.

2.4 What next?

♡ This is the end of the first seminar session and if you have gone through the work and signed in with your seminar tutor then you are free to go. I would just check that you know when and where your next seminar session will take place[6]. I would also check that you have a copy of the second problem sheet (it should be in Appendix A of this guide) and that you know when and where you have to hand in your answers.

A useful thing to do if you get to the end of these exercises way before the end of the seminar hour is to look at Chapter 4, and also the first part (exercises 1, 2 and 3) of Chapter 5. This introduces you to the programming environment which you will use for your coursework.

If you decide not to look at this now, then you should put this on your to-do list to be completed by the end of week 2 (latest).

[6]It will not be in this lab!

Chapter 3

An Introduction to Programming

3.1 Designing computer programs

Designing a computer program is a complicated business. It requires a great deal of creativity, a considerable understanding of the task or process being automated, a good eye for accuracy and finally a large degree of patience.

In industry, the process of designing a computer program is separated from the task of actually writing the program itself. This is because the skills required for each task can be quite different.

The process of program design often starts with an idea supplied by a *customer*. The customer may telephone you stating that they are considering automating a shampoo bottling plant, and that they need a program to control the machinery. The program designer's task is then to write a document stating exactly what the customer requires, which can then be passed on to the programmers and turned into code; this document is called the *program specification.*

Writing a program specification is difficult, particularly because you need to pitch it at exactly the right level. A specification which states

> "Get some bottles, put some shampoo in and then put them in the boxes"

is probably not detailed enough for a programmer to successfully write a piece of code which will do exactly what the customer is looking for. The programmer may (legitimately) write some code which gets 2000 bottles at a time for example; this after all meets the specification, though probably does not fit with the machinery available.

Alternatively, a specification which states

> Line 10: Assign to the first variable the result of - If the first bottle is ready and there is some shampoo in the machine or the green light is on then ...

may be too detailed for the programmer to have complete control over the implementation. The specification will probably also be extremely long and may, as here, contain ambiguities.

So as you can see, writing a specification is not as straightforward as it first seems.

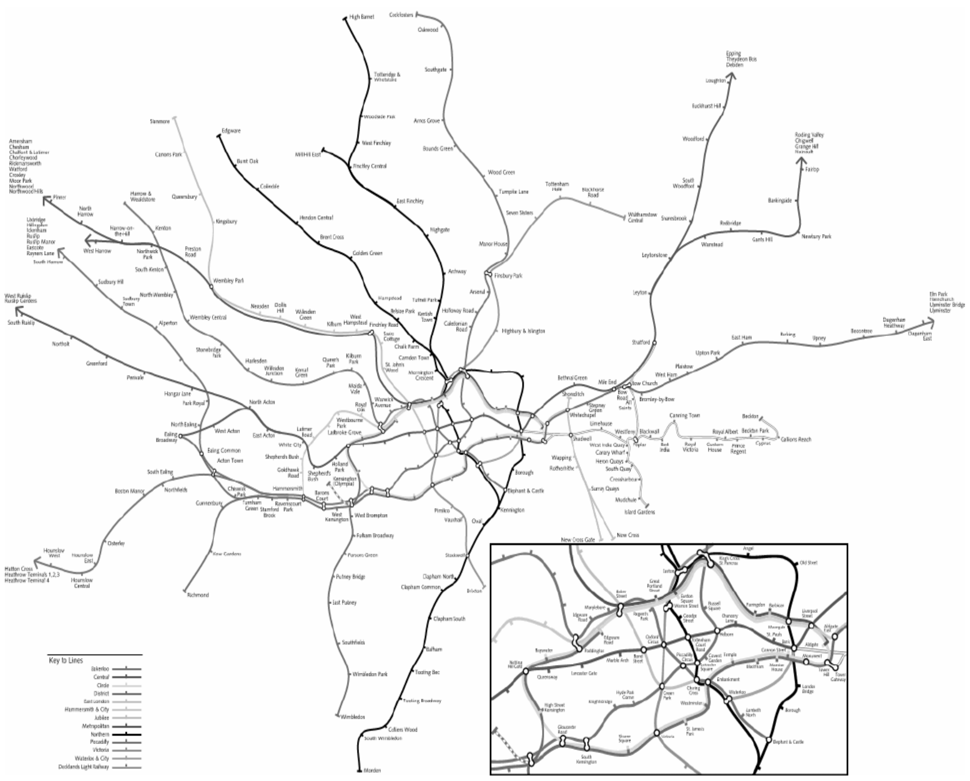


Figure 3.1: The geographical representation of the London underground

3.1.1 The Diagram

Consider the following example based on that found in the excellent book *Using Z: Specification, Refinement and Proof* by Jim Woodcock and Jim Davies:

Writing a specification at an appropriate level of abstraction is essential. A good example of this is provided by the various maps of the London Underground. When the first map was published in 1908, it was faithful to the geography of the lines: the shape of the track and distance between stations were faithfully recorded. However, the purpose of the map was to show travellers the order of stations on each line, and the various interchanges between lines; the fidelity of the map made it difficult to extract this information. Figure 3.1 shows the geographical representation of the London underground.

The map was changed in 1933 to a more abstract representation which was rather nicely named the Diagram. The draughtsman Harry Beck, who produced the imaginative yet stunningly simple design, based the map on the circuit diagrams he drew for his day job. All the detail concerning connectivity was maintained, though the simplification meant that passengers could see at a glance the route to their destination. Abstraction from superfluous detail - in this case the physical layout of the lines - was the key to the usefulness of the Diagram. The Diagram can be seen in Figure 3.2.

The Diagram was, and still is, a good specification of the London Underground. It is

- *Abstract.* Since it only records logical layout, not the physical reality in all its detail.
- *Concise.* Since it is printed on a single A5 piece of paper which is folded in such a

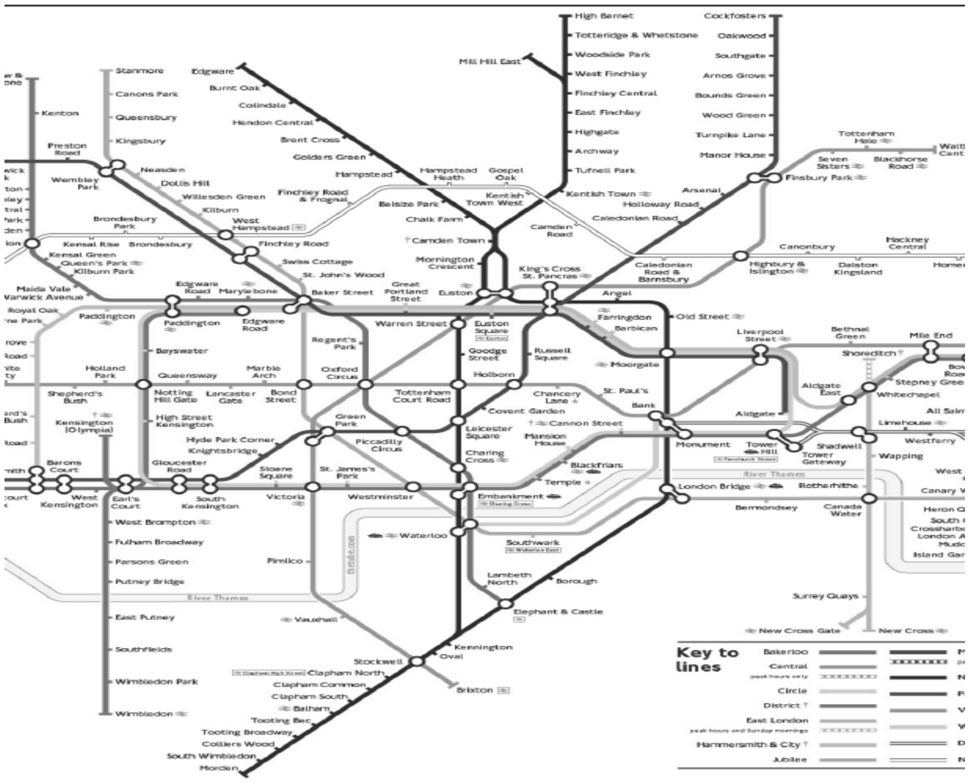


Figure 3.2: The Diagram, 1933; a more abstract description

way that it fits exactly into your pocket.

- *Complete.* As every station on the London Underground network is represented.
- *Unambiguous.* Since the meaning of the symbols used is explained, and the Diagram is represented in simple geometric terms.
- *Maintainable.* Since it has been successfully maintained over the last 60 years, reflecting the changes to the network as new stations and lines have been opened and others have been closed.
- *Comprehensible.* It must be readily understood by the general public. This has been the case as it has been regarded fondly by its users since 1933.
- *Cost-effective.* Since it only cost five guineas to commission the specification from the engineering draughtsman Harry Beck.

The Diagram gives its users a good conceptual model. It embodies a specification structure that enables users to make sense out of a rather complex implementation. To do this it uses abstract shapes, colours and compression. All lines are reduced to ninety or forty-five degree angles and the central area, where there are more stations, is shown in greater detail than the outlying parts, as if the Diagram were viewed through a convex lens.

The Diagram is an excellent example of a specification. You may be interested to know that it was first rejected by the Publicity Department of the London Underground, as the abstract notation was thought to be too strange and incomprehensible for the ordinary user of the Underground network.

3.1.2 Writing your own specifications

Of course not all specifications can be dealt with in a nice pictorial form such as the Diagram. Most specifications in fact use *Natural Language* and/or some form of *Mathematics* or *Logic*.

Natural Language is perhaps the easiest way to communicate ideas, as most of us understand one language or another, English or Spanish for example. If you are to write specifications in a natural language then you must make sure that the specification is unambiguous. The specification for a shampoo bottling firm was unclear.

> Line 10: Assign to the first variable the result of - If the first bottle is ready and there is some shampoo in the machine or the green light is on then ...

We cannot be sure whether the 'or' goes with the 'shampoo in the machine' part of the sentence, or with the 'If the first bottle is ready and there is some shampoo in the machine' part of the sentence.
We might make more sense of the definition by adding some mathematical notation (brackets in this case),

> Line 10: Assign to the first variable the result of - (If the first bottle is ready and there is some shampoo in the machine) or (the green light is on) then ...

or by employing some logic

> A = First bottle is ready
> B = Shampoo is in the machine
> C = Green light is on
>
> Line 10: Assign to the first variable the result of - $((A \wedge B) \vee C)$

This final example uses some *Propositional logic*; three propositions are defined (A, B and C) and they are combined using the logical AND ($\wedge$) and OR ($\vee$) operators. The advantage that we see here is that the closer we move towards maths (or logic) the less chance there is of introducing any ambiguities.

3.2 Building computer programs

Building a computer program is the task traditionally described as *programming*.

Despite many misconceptions, programming is not about sitting at a desk full of cans of Coke and bashing out some obscure lines of text which resemble the programmer's thoughts on a particular problem. Programming is an exact and detailed science which involves translating *abstract* specifications into more *concrete* implementations. The concrete implementation is traditionally known as program *code*.

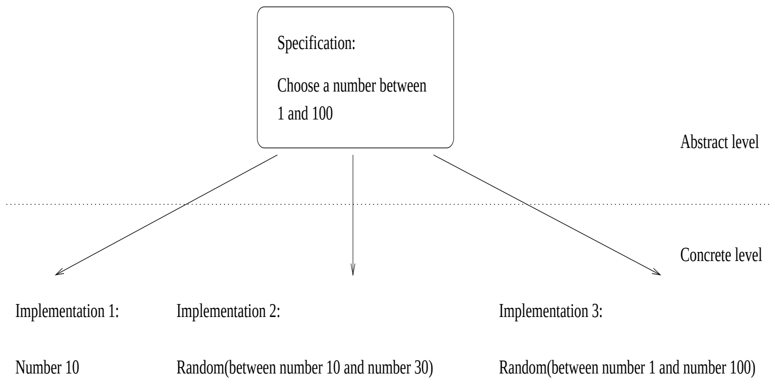

Specification:
Choose a number between 1 and 100
Abstract level
Concrete level
Implementation 1:
Number 10
Implementation 2:
Random(between number 10 and number 30)
Implementation 3:
Random(between number 1 and number 100)

Figure 3.3: Example of abstract- and concrete-level design

3.2.1 Abstract and concrete

So what exactly is all this talk about *concrete* and *abstract*?

You have seen already, in the description of a specification, that when we describe a problem which we may want to computerise, we should choose carefully the level of detail at which the problem is described. In writing a specification we must not get drawn in to any nitty-gritty points which are not wholly in the domain of the problem itself. But why do we make such a fuss about this, and does it really matter?

Well, yes it does. When we program it is desirable to have as much freedom as possible: the freedom to choose our own programming language; choose our own structure and individual style; and maybe reuse bits of programs, to save time or money for example.

In fact, it is possible to have many different programs which implement the same specification. Consider figure 3.3 for example. Here we have a specification which states, “Choose a number between 1 and 100”. There are three implementations of this specification in the figure:

- The first program simply produces the number 10. This, you may think, does not meet the specification given, but think about it carefully. The specification says choose a number between 1 and 100, and the program does, it chooses the number 10. It chooses the number 10 each time the program is run of course, which is probably not what the person who wrote the specification wanted to happen. But the specification does not say that the number chosen should be different each time the program is run, so effectively the program is a perfectly good implementation of the specification.

- The second program produces a random number between 10 and 30. This also meets the specification as the program certainly does choose a number between 1 and 100. Again, this is probably not what the person who wrote the specification intended.

- The third program is probably what you would have expected. It randomly chooses a number between 1 and 100. This also meets the specification and had the specification been written more carefully, stating, “...a different number in this range should be chosen with equal chance each time the program is run...”, then this would be the only valid implementation of the specification written above.

This may seem a little confusing. Why is it useful to have a number of possible computer programs which implement a single abstract specification? The point is that it may not matter to the customer exactly what the program does, provided that it is within the bounds of the specification. Therefore, the programmer has flexibility when producing a program, and the customer receives a program which meets their requirements. Everyone wins.

Abstract specifications are useful as they allow customers who might be ordering a computer system to write a collection of unambiguous requirements. They may pass this specification to a number of different programmers and receive a number of different programs back. Although these programs may be different and may be written in a number of different programming languages, on a number of different machines, they will all act exactly as the specification states. The specification therefore acts as a *contract* between the customer and the programmer, and a contract between the abstract description and the concrete implementation.

3.2.2 Translation

Programming is the business of taking an abstract-level specification and translating it into a concrete-level piece of code, and, as we have already seen, programmers may do this translation in an assortment of different ways.

The translation between an abstract-level specification and a concrete-level design is actually called *refinement*. Each of the programs in figure 3.3 is a valid refinement of the specification.

Just as it is important to carefully write a specification, it is also important to make sure that the program implementation is an accurate coding of the description in the specification.

Usually a specification will have a number of complicated clauses, and may also span a number of pages. Although the specification may be exact in its description, a programmer may make a mistake when reading it and consequently code something different. For this reason, some specification methods have complicated mathematical rules which translate a piece of the specification (usually written mathematically) into the corresponding piece of program code. These rules are known as *refinement rules*. You will learn more about these if you choose to do the software specification course later in your degree.

3.3 Testing computer programs

Testing a computer program is an extremely important business. There are many examples which I can cite where software has failed due to inadequate testing. Rather than bore you with a complete chronology, consider the following example:

The story of Ariane 5 is a good one. In the thrust direction control unit, code was reused from Ariane 4. In this code, horizontal speed was represented by a 16 bit value. But horizontal speed in Ariane 5 was greater, and caused an overflow, which raised an exception. The specification said (very foolishly) that if an exception arose, the processor should be shut down and restarted. Shutting the processor down caused the thrust direction to jump suddenly sideways, which broke the rocket in half.

Of course not every example of software failure will end in a disaster quite as catastrophic as this. However, the consequences of your code failing may prove to have just as much of an impact on the results of a small company, or on the grade assigned to your computing assignment, for example.

Testing is defined as the detection of failure; failure is the departure of the behaviour of a program from its requirements. Unfortunately, it is not possible to show the absence of failure by testing, as testing will only tell us whether a program fails in a particular scenario or not. The purpose of testing is to eliminate as many problems in the code as possible. This increases the programmer's (and user's) confidence in the piece of code. As the number of failures detected in a program becomes less, the more you will feel that the program exhibits the correct behaviour.

3.3.1 Methods of testing

The experimental science of software testing has been the subject of research for a number of years. Consequently, there are a number of testing methods which are shown to be effective. We will see, and use, a few of these methods in this course.

Test of logical paths of program

One useful activity when testing a program is to check all the logical paths through the program. Consider the small example:

```
while ( x < 10 )
{
     if ( even(x) )
     {
          System.out.println("The number is even \n");
     }
     else
     {
          System.out.println("The number is odd \n");

          x = x + 1;
     }
}
```

To test the logical paths of this short piece of code the user would need to design tests to cover at least three cases: The case when `x` is greater than or equal to 10, in which case the while loop would not be executed at all; the case when `x` is less than 10 and is even, in which case you would expect `The number is even` to be printed at least once, and finally the test when `x` is less than 10 and is odd, in which case you would expect `The number is odd` to be printed at least once.

Forgetting one of these cases will mean that you have not tested part of the code; this may be the piece of code which blows up, or wipes the hard disk, or .... Would you expect any of the logical paths in the program to reveal an error in the above code?

Range of inputs

Another way to test the example program would have been to test the range of inputs. If we can be sure that the program produces the right output for each valid (and even invalid) input, then we can be a bit more sure that it does what we expect. We may for example have tested the program with a negative value, a positive value and the value 0.

Boundary cases are also important. You may want to check that the computer deals correctly with the highest possible number and the lowest possible number. Finally, you may want to put some spurious values into the program – what happens when you type in a character for example, or if you just press the enter key, or if you just sit on the keyboard?!

Of course you have to select your range of inputs carefully. Selecting the numbers `137645813451875`, `0.14643528745`, `-23` and `19`, say, would not have found the infinite loop in the program.

Systematic tests

It is all very well to test the logical paths of the program and the ranges of input, but it is sometimes the sequence of operations in a program which causes it to break. For example, the 'landing-gear down' and 'increase throttle' routines may both work exceptionally well by themselves, but putting the landing-gear down and then increasing the throttle may cause the plane to head towards the ground at a rapid speed. This is probably not what you want.

It may be worth testing a sequence of operations in your program, testing all the permutations of the routines a, b and c for example, to make sure that one does not exhibit any unexpected behaviour.

Random tests

Random testing is a perfectly legitimate activity, but do not expect it to consistently come up with all the errors which may be detected by a logical or systematic approach.

A true random test of a program is actually quite difficult to achieve. It would probably require a random number generator to choose between the routines in the program which were to be tested. A random test would also require a similar random selection activity to choose random input data to the program; of course the amount of data itself must also be randomly chosen. So be careful when you use the term 'random testing'.

Intuitive tests

The process that people often think of as random testing is actually called *intuitive testing.*

After you have spent some time programming you may become aware of common errors which appear in programs. A program which stores and deletes a collection of names will often be fooled if the first thing you ask it to do is to delete. Programs which accept characters as input will often break if you feed in a control character.

Choosing cases like this to test your program is not a random activity - you are usually selecting the tests based on your intuition as a programmer. So when you run a program for the first time and select a number of seemingly random operations, you will probably find yourself going through a number of cases which you expect to work, followed by one or two cases in which you think the program may break.

These tests usually require a bit of thought, but you can come up with some interesting results quite quickly.

Test rigs

A *test rig* is a piece of software which will run alongside the program to be tested. The test rig may generate test data, supply tests, and record and calibrate the results as the testing takes place. Test rigs are useful as they automate the testing process, removing any possibility of human error. They also allow a large number of tests to be carried out automatically; you may for example run the test rig over night, checking the results the following morning.

Test rigs also allow large systems to be tested with relative ease. Programmers of large systems, those used by banks for example, often use test rigs when they are modifying the system. This means that the results before and after the modification can be compared to make sure that the system is still operating correctly.

One thing which is slightly ironic about test rigs is that they themselves need testing, perhaps with test rigs, which themselves...

You might try some of these test methods later in the course.

The method of testing you use will often be dictated by a number of factors. You may not have time to carry out a logical or systematic test and an intuitive test will have to do; it may be essential that you identify as many errors as possible, in which case random and range testing might not be good enough. It is up to you as a programmer to weigh up these factors to determine which method is appropriate given the situation.

Chapter 4

Introduction to the Robot-maze Environment

The coursework exercises for this year's CS118 course will be based on a simulated 'robot-maze' environment. A small robot has been designed to be able to navigate its way through mazes to find a target at some given location. This task resembles those used in the classic learning experiments of the 1960s which included laboratory mice (and cheese, mild electric shocks, mice of the opposite sex, etc). The objective of the robot (or mouse as it was then) is to find the given target as rapidly and efficiently as possible, learning the maze over several runs and so on.

Building a real robot and a real maze requires a combination of efficient sensors and mechanics, sophisticated steering and speed control, clever maze exploration and navigation procedures and, no doubt, a good deal of glue. For the purpose of these exercises we focus entirely on designing the maze exploration and navigation procedures. We make no attempt to model the physics of a moving wheeled (or legged!) robot and concentrate solely on the part of the problem which can best be solved with software.

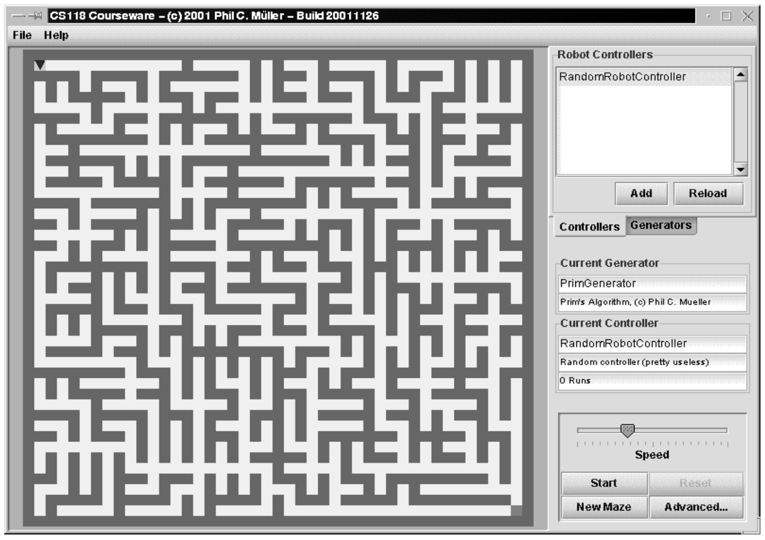
4.1 The robot-maze environment

The simulated Robot-maze environment has the following characteristics:

- The robot moves through a simple square-block maze of the type illustrated in Figure 4.1. The floor space of the maze is divided into squares of uniform size. Each square is either occupied by a wall or is empty.

- The robot occupies exactly one non-wall square and moves in discrete steps, one square at a time, north, south, east, or west. The robot cannot move diagonally. If the robot attempts to move outside the boundary of the maze or into squares occupied by walls it suffers a harmless collision (indicated by flashing red in the simulation) and stays in the same square.

- The direction the robot moves in is determined by the way it is facing (its heading, indicated by an arrow in the simulation). The robot can change the direction it is facing by rotating on the spot. Each of these rotations are directed by the robot's



RandomRobotController

Current Generator
PrimGenerator
Prim's Algorithm, (c) Phil C. Mueller

Current Controller
RandomRobotController
Random controller (pretty useless)
0 Runs

Figure 4.1: The Robot-maze environment

control procedure - a method called `controlRobot` - which is run once before the robot makes a move.

- A robot run begins at the top left-hand square of the maze. The run ends when either the robot reaches the target or the user loses patience and stops the robot manually (by pressing **Reset**). The target square is usually the bottom right-hand corner of the maze, but this along with the robot start position can be modified by the user.

During its execution the robot's control program (which you are required to write) has access to the following information:

- The direction the robot is currently facing;
- The status of the squares ahead, behind, left and to the right of the robot. Squares are either walls, empty, or beenbefore squares which are empty squares the robot has previously occupied during its current run through the maze. The boundaries of the maze are treated as walls;
- The x and y co-ordinates of the square the robot is currently occupying, and those of the target square;
- How many attempts the robot has made at solving the given maze.

4.2 Programming robot control programs

The simulated robot-maze environment is written in Java. The programs which you are required to write for this course are also Java based which means that you will be writing code which directly hooks into this robot-maze environment.

To allow this hook-up, there needs to be a common interface between the robot-maze Java code and your own Java code. Essentially this means that you need to be talking the same language; we define this language below.

The information listed below is important and you should make sure that you understand what it all means. If you are not clear on anything then you might like to talk about it between yourselves. Understanding *program interfaces* like this is very important, particularly if you are to use it to write your own program code.

4.2.1 Specifying headings in the maze

Four pre-defined constants are used to specify directions in the maze. These are

```
NORTH, EAST, SOUTH, WEST
```

where the maze follows the usual mapping convention of having `NORTH` upwards and `EAST` to the right etc.

These elements of the interface language are concretely represented as Java `int` values. This will be useful to know when you start referring to the types of these values in your programs. One advantage of this scheme is that

`NORTH+1 = EAST`; `EAST+1 = SOUTH`; `SOUTH+1 = WEST`.

4.2.2 Specifying directions relative to the robot heading

Four pre-defined constants are also used to specify directions relative to the robot's heading. These are

```
LEFT, RIGHT, AHEAD, BEHIND
```

As with headings these are also of the type `int` and therefore

`AHEAD+1 = RIGHT`; `RIGHT+1 = BEHIND`; `BEHIND+1 = LEFT`

A fifth constant `CENTRE` is also defined, which can be useful as a 'null' or 'give-up' value when communicating values between parts of complex control programs.

Do not be put off by the fact that these values have an `int` type. As far as the control programmer (this is you) is concerned, all references to headings and directions are done using the constant *name* (ie. `RIGHT`, `NORTH` etc.) and not the constant *value* used to represent it. This is our first encounter with *program abstraction.*

As the values are defined as part of a Java interface, we need to prefix these values with the name of the interface when they are used in the actual program code. The interface is called `IRobot` and therefore any reference to the constant `AHEAD` in the code is in fact done using `IRobot.AHEAD`. This might seem a bit quirky but you will soon get used to it.

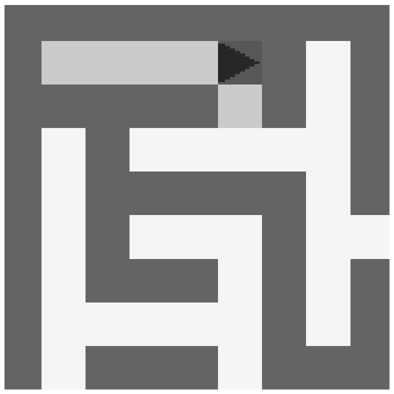


Figure 4.2: Example of sensing robot surroundings

4.2.3 Sensing the squares around the robot

The method `robot.look(`*direction*`)` takes a value specifying a direction relative to the robot (e.g. `IRobot.AHEAD` or `IRobot.LEFT` etc.) and returns a value which indicates the state of the corresponding square neighbouring the robot. The possible states are

```
IRobot.PASSAGE, IRobot.WALL, IRobot.BEENBEFORE
```

`IRobot.PASSAGE` indicates an empty square that has not yet been visited on the current run through the maze. `IRobot.BEENBEFORE` indicates an empty square that has already been visited during the current run through the maze. `IRobot.WALL` indicates a wall or the edge of the maze.

Figure 4.2 shows a typical situation that might arise during a robot run. The robot is located in the arrowed square, facing in the direction of the arrow, with squares visited previously during the same run shaded in grey. The walls are in black. In this situation `robot.look` would return the following values:

Function call	Result
`robot.look(IRobot.AHEAD)`	`IRobot.WALL`
`robot.look(IRobot.BEHIND)`	`IRobot.BEENBEFORE`
`robot.look(IRobot.LEFT)`	`IRobot.WALL`
`robot.look(IRobot.RIGHT)`	`IRobot.BEENBEFORE`

If the robot chooses to turn right and then move forward one square, then a call to the method `robot.look(IRobot.AHEAD)` would return `IRobot.PASSAGE`.

4.2.4 Sensing and setting the robot's heading

The method `robot.getHeading()` returns the robot's current heading in the maze. That is either `IRobot.NORTH`, `IRobot.SOUTH`, `IRobot.EAST` or `IRobot.WEST`. In the example in Figure 4.2 a call to the method `robot.getHeading()` would return the value `IRobot.EAST`. There is a sister method called `robot.setHeading(x)`, which can be used to set the robot's heading (where the parameter `x` is one of `IRobot.NORTH`, `IRobot.SOUTH`, `IRobot.EAST` or `IRobot.WEST`).

4.2.5 Sensing the location of the robot and target

The methods `robot.getLocationX()` and `robot.getLocationY()` return the x and y co-ordinates of the robot in the maze. The top left square in the maze is square (1,1).

The methods `robot.getTargetLocation().x` and `robot.getTargetLocation().y` return the x and y co-ordinates of the robot's target. Note that these methods look slightly different to those that sense the robot's x and y position. This subtlety will be explained later in the course.

4.2.6 Specifying turns

The method `robot.face(`*direction*`)` makes the robot turn in the *direction* specified (one of `IRobot.AHEAD`, `IRobot.BEHIND`, `IRobot.LEFT`, or `IRobot.RIGHT`) relative to its current heading. The turn is performed immediately and will be reflected in the results of subsequent calls to `robot.getHeading()`.

4.2.7 Moving the robot

The control software which you will build is polled. This means that the code you write will be called by the robot-maze environment each time it is ready to move the robot. This effectively switches control between the environment and your controller, and the environment and your controller, and the environment and your controller etc.

The code which you write should therefore first point the robot in a suitable direction and then signal to the environment for the robot to be moved. To move the robot you make a call to the `robot.advance()` method.

4.2.8 Generating random numbers

The Java method `Math.random()` returns a random floating point number greater than or equal to 0.0 and less than 1.0. The number returned is computed so as to ensure that every number appears with equal probability.

To generate random numbers uniformly distributed between 0 and n you will need to take the result, multiply it by n, round it to the nearest integer (using `Math.round()`), and then cast the result to an `int` value, e.g.

```
int result = (int) Math.round(Math.random()*n);
```

So, the code

```
randno = (int) Math.round(Math.random()*3);
```

for example, will assign a random integer value between 0 and 3 inclusive (that is one of four distinct possibilities) to the variable `randno`.

4.2.9 Detecting the start of a run and a change of maze

The method `robot.getRuns()` returns a number (`int`) which corresponds to the the number of previous runs which the robot has made on a given maze. After you have run a robot through a maze you will notice that the current controller screen to the right of the robot-maze environment displays `1 Run`. This is the result of the `robot.getRuns()` method. You will find that this method is useful in the second coursework.

Chapter 5

Coursework 1 (Part 1): Simple Robots

The first coursework for CS118 consists of two parts. You will need to complete both parts if you are aiming for a top grade.

Your answers to this coursework must be completed by **Wednesday 26 October (Week 5)**. You will be asked to come in on that day and demonstrate that your code works and that you have understood the material for each of the exercises. If we are not able to mark your work on that day, for whatever reason, then you will receive no marks.

Your work will of course be marked for functionality, that is the program does what it is supposed to do. It is also useful to remember that the work will be marked for programming style, clarity, re-usability and use of techniques taught throughout the course. This means that even if your code works wonderfully, you may not get fantastic marks if it looks like a dog's dinner. Likewise, if you do not finish all the exercises, but your solutions look like a masterpiece, then you are likely to do well.

Cooperation, Collaboration and Cheating

If a submitted program is not entirely your own work, you will be required to state this when the work is marked. Any and all collaboration between students must be acknowledged, and may result in stricter marking of the work. Consultation of textbooks is encouraged, but programs described elsewhere should not be submitted as your own, even if alterations are made. It will be useful to quote here the University's regulations on the subject:

> *. . . 'cheating' means an attempt to benefit oneself, or another, by deceit or fraud. This shall include deliberately reproducing the work of another person or persons without acknowledgement. A significant amount of unacknowledged copying shall be deemed to constitute prima facie evidence of deliberation, and in such cases the burden of establishing otherwise shall rest with the candidate against whom the allegation is made.*

Therefore, it is as serious for a student to permit work to be copied as it is to copy work. Any assistance you receive must be acknowledged. If in doubt, ask.

5.1 Exercise 1

To begin you need to copy the mouse-maze environment and the controller software to your home directory. I suggest that you first create a `cs118` directory. Invoke a terminal window and type in this window the UNIX command

```
mkdir cs118
```

and then change to this directory using the command `cd cs118`.

You should now use a web browser to go to the CS118 course web page (which should be in your bookmarks if you have followed all the instructions in problem sheet 1.)

Under the **Coursework** section of this web page you will see that there are four links, one of which says **Maze software** and one of which says **Dumbo controller**. Click on these with your right mouse button and select the *Save Link Target As* option from the menu. This will allow you to save the file. When you save these files, make sure you double-click on the `cs118` directory so that the file is saved to the appropriate place. The **Download Manager** will confirm that the file has been saved and once you have done this for both files, you can check that the files have been saved by typing `ls -al` in your terminal window.

You should find two new files in this directory. One of these files is a `.jar` file; this postfix means that the file is a Java archive, an efficient ZIP-like file format which allows all the component parts of the robot-maze environment (stored in this file) to take up as little space as possible in your home directory. The other file is a `.java` file like those you created in your first seminar. This `.java` file is the robot controller which interfaces with the robot-maze environment software.

The result of the `ls -al` command should look something like this[1]

```
-rw-------   1 saj      dcsstaff     697 Aug 11 10:32 DumboController.java
-rw-------   1 saj      dcsstaff   99539 Aug 11 10:32 maze-environment.jar
```

If you find that the file size of either of these files is zero (this is the number in the fifth column), then something has gone wrong during the downloading. In this case you should try downloading them once again.

You need to compile the `.java` file if you are to run it and in order for the controller software to run along side the mouse-maze environment the two programs need to be compiled together. This requires a small addition to the `javac` command which you used in compiling your first Java programs. Type into you terminal window the command

```
javac -classpath maze-environment.jar DumboController.java
```

[1]If you are doing this under Windoze then you may find that the maze environment file is saved as a zip file. If this is the case then the best thing to do would be to rename this as a `.jar` file, i.e. `maze-environment.jar`.

This compiles the controller program `DumboController.java` into a corresponding class file (`DumboController.class`). The compilation is performed in the context of the robot-maze environment (through the `-classpath maze-environment.jar` extension) which is just what we want.
You can now run the robot-maze environment by typing

```
java -jar maze-environment.jar &
```

The `&` by the way, frees the window so that you can still use it for other business. Admire the baroque elegance of the highly sophisticated computer graphics in the robot-maze environment program. To make it look more maze-like, click on the `Generators` button and then on the **PrimGenerator** in the window above. You will see that this fills the *Current Generator* information panel. If you now click on the **New Maze** button at the bottom right you will get a new maze (generated through an application of Prim's algorithm).

Now that you have a maze you need a robot. Click on the **Controllers** button and select the **RandomRobotController**. You will see that this configures the *Current Controller* information panel.

The **RandomRobotController** is a pre-installed piece of controller software which drives the direction-choosing capabilities of a basic robot. Before clicking on the **Start** button to test the robot, set the **Speed** gauge to the far right of the screen.

When the robot is running you will see that it exhibits some unusual behaviour. First you will see the direction change, as indicated by the blue arrow. You will also notice that it leaves a trail (of been-before squares) as it moves through the maze. Every now and then the robot crashes into a wall (indicated in red), this is because the controller which is being used to drive the robot makes no allowance for where the walls are in the maze.

You should familiarise yourself with this environment. See what happens when you increase the speed, try generating new mazes, try editing mazes with your mouse, try moving the location of the target, try changing the dimensions of the maze.

5.2 Exercise 2

One of the nice features of the robot-maze environment is the ability to experiment with different controller software. The `DumboController` which you compiled earlier can be loaded into the environment by clicking **Controllers** followed by **Add**.

This will provide you with a directory menu from which you should double click on your `cs118` directory. Here you will find your `DumboController.class` file which you can then highlight and **Open**.

You will see that this adds the `DumboController` to your list of robot controllers. You will use this same process to load all the robot controllers which you write during this first piece of coursework.

If you click on the **DumboController** in the robot controllers menu, the environment will switch between controllers. Run the new robot controller to see if you can work out where **RandomRobotController** and **DumboController** differ. Try to characterise the strategies which they appear to follow. You may find find it helpful to test the robots on different mazes.

It may become obvious that it is not always easy to see exactly what strategies these robot controllers appear to follow. It is often quite difficult to reverse engineer from the *behaviour* to the *specification*. Similarly, it is not good practice to have a specification-less program, as if the user is in any doubt as to the behaviour of part of the program, this problem can be easily resolved by consulting the specification.

5.3 Exercise 3

Use a text editor to look at the source code in the file `DumboController.java`. The method `controlRobot` is the controller part of the code. If you have not already worked it out, study the code and see if you can detect what strategy this control program implements.

Note the use of the `import` statement at the top of the program. This is the statement which connects the robot-maze environment with the controller code. The behaviour of this interface was described in Chapter 4 and you might like to go back to this chapter and just check what each of the robot methods and constants do.

Talk with your friends about this code. Is it clear to you which parts of the code are 'pure' Java and which parts come from the interface to the robot-maze environment?

5.4 Exercise 4

♡ An order is received from an existing customer for a modified dumbo robot:

> 'Could we have a modified robot controller that still chooses directions randomly, but avoids crashing into walls.'

This description can be identified as the *specification of requirements* for the new robot which the customer requires.

A good software developer will set about solving this problem in a systematic and logical fashion; for example, using the processes of *Design*, *Build* and *Test* which were described in Chapter 3.

Designing a new piece of software requires a complete understanding of the problem to hand; you cannot write a program for a problem which you do not understand. To help work out what the specification states, we will break the description up into its constituent parts.

- The text describes the modified robot as 'still choosing directions randomly'. This would suggest that the part of the robot control program which chooses a random number and then converts this to a direction should stay as it is.

- The text also states that the robot should 'avoid crashing into walls'. Sometimes the existing robot controller chooses a random direction which will point the robot towards a wall. What you need to do is filter out these occurrences. This will involve looking to see if the direction chosen does point the robot towards a wall, and if so, choosing another direction.

A software developer would be right in thinking that the existing `controlRobot` method in the `DumboController.java` file can be reused; there are many similarities between the existing robot controller which you studied in exercise 3 and the new robot controller. Your answer to this exercise should therefore be based on the code found in `DumboController.java`.

The main difference between the old controller and the new one is that the new controller will prevent the robot from crashing into walls.

Design question 1: How do you think you can detect if the robot is about to crash into a wall? Hint: look at section 4.2.3 of *The Guide*.

Once you have discovered how to detect for collisions, you will need to ensure that the robot controller keeps choosing directions for the robot until a non-wall direction is found.

Design question 2: How do you plan to do this? Hint: look at the lecture notes and any Java books you have to find out more about loops.

Once you have designed the program you are ready to *build* the new robot controller. The new robot controller can be built by making modifications to the existing `controlRobot` method in the `DumboController.java` file. The changes require about two lines of code, so there is no need to get carried away, or indeed too daunted by the programming task ahead!

After saving the `DumboController.java` file, you should compile the code using the command

```
javac -classpath maze-environment.jar DumboController.java
```

to generate a new `DumboController.class` file. Once you have eliminated any fatal, compiler-detectable errors, a new class file will be created. The new class file will be detected by the robot-maze software and it will ask you whether you would like to reload this new class; you should press **Yes**; you can now test your new solution.

Before you finish this exercise, consider how you would convince the customer that you have tested the program and that it fully meets the customer's requirements. Document your test strategy (in a paragraph or two) so that we can discuss your approach with you.

When you have the program working, copy your `DumboController.java` file to the file `Ex4.java`. This will ensure that you have a copy of the file which can be marked on Wednesday 26 October. If you fail to to this, for whatever reason, you will not gain marks for this exercise.

5.5 Exercise 5

Now that the robot no longer crashes into walls, it is easier for your customer to monitor the robot's behaviour. As a result, you receive an email stating that they have noticed some rather unexpected behaviour.

While testing the current robot your customer noticed that although it seems to choose directions randomly, some directions appear to be selected more often than others.

This is a difficult trait to investigate by hand. You could try running your robot slowly and making a note of the directions it chooses, but this is rather painful (and life is too short for such tedium[2]). An alternative approach is to add a logging mechanism, so that the robot keeps a record of which direction is selected each time it moves. The idea is that from this log of movements you will be able to analyse the behaviour of the robot for a particular maze.

The following output is taken from a working solution to this exercise:

```
I'm going forward at a deadend
I'm going forward down a corridor
I'm going forward down a corridor
I'm going forward at a junction
I'm going backwards at a deadend
I'm going backwards at a junction
I'm going backwards at a deadend
I'm going forward at a junction
I'm going backwards down a corridor
I'm going forward at a junction
I'm going backwards at a deadend
I'm going forward at a junction
I'm going forward down a corridor
I'm going forward down a corridor
I'm going backwards at a deadend
I'm going forward down a corridor
I'm going forward down a corridor
I'm going forward at a junction
I'm going backwards at a deadend
I'm going right at a junction
I'm going forward down a corridor
I'm going right at a junction
```

You will see that the first thing that the robot detects (as it begins the exploration of the maze) is that it is at a dead-end and therefore the only thing that it can do in this case is move forward (see figure 5.1).

Once it has done this, it then detects that it is in a corridor and therefore it decides to move forward. The same thing occurs for the third move.

After three moves the robot finds itself at a junction, here it decides to go forward once more, at which point it reaches a dead-end and can go nowhere but backwards.

[2]Unless of course you are an Arsenal supporter in which case you are used to this.

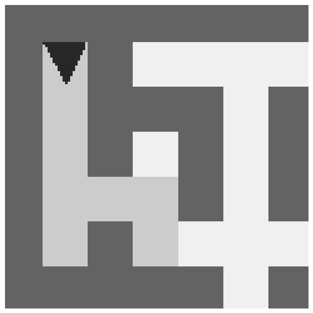


Figure 5.1: Coverage through an example maze.

You can follow the route of the robot until the reset button was pressed. You will see that the robot is not particularly efficient (as it is operating in just the same way as the robot in exercise 4) but it does print out an accurate log which itself might be useful at a later date.

It is always simpler to write more complicated software such as this as a series of smaller tasks. We will be building on the results of our previous exercises, so make sure that any programming which you do is done in the file `DumboController.java`.

Consider the first problem of outputting the text which identifies the direction in which the robot is heading, the

```
I'm going forward
```

part of the output.

For the robot controller to print a log of this direction chosen, you need to include an instruction to output text. You have already seen such an instruction in the exercises in your first problem sheet. You might want to remind yourself how the `System.out.println` instruction works. Try adding a simple `System.out.println` instruction to your robot controller, one that says "`I'm going `", or something similar. Compile and run the new robot controller to observe its effect.

Now that your controller outputs some text, you are ready to modify the program so that it outputs the `forward`, `backwards`, `left` or `right` as required. Producing this output is quite simple using the `System.out.println` instruction; the trick is deciding which of `System.out.println("forward");` or `System.out.println("backwards");` etc. is required.

Let's consider the sub-problem of recording the direction chosen. One way to do this is to is to add an extra variable to the program, `heading` say, in which we store an `L` when the direction chosen is `LEFT` and an `R` when the direction chosen is `RIGHT` etc. As the value we want to store is a character, the `heading` variable is declared as

```
char heading;
```

Once the variable has been declared, we need to ensure that the correct value is assigned to the variable. When the `LEFT` direction is chosen we can assign the character `L`, e.g.

```
heading = 'L';
```

The same statement applies - with slight modification - to the three other possible directions.

Now we have a record of the direction selected, to convert this to the corresponding output we simply inspect the variable at a suitable place in the program and execute a corresponding `System.out.println` statement as required[3].

Design a program which prints the direction the robot is going in. Make sure that you answer the following questions in your design:

Design question 1: Where in the control program do you want to declare the variable '`heading`'?

Design question 2: What value is this variable initialised to?

Design question 3: Where in the code is this variable assigned a value?

Design question 4: Where in the code is this value inspected and a corresponding piece of text printed out?

These may seem like obvious questions to ask (and the answers may be surprisingly trivial), but asking such questions does encourage you to write correct programs. Once you have designed this part of the program, it may be worth building it and running some tests to make sure that things are working correctly.

Now you are ready to work on the final part of the software; detecting whether the robot is at a dead-end, in a corridor, at a junction or at a crossroads. Answer the following questions as part of your design:

Design question 5: How is it possible for the robot to detect whether it is at a dead-end, in a corridor, at a junction or at a crossroads?

Design question 6: Is there a corresponding method in the maze-environment package which allows this detection to take place? Hint: see section 4.2.3.

Design hint 1: You might like to add an additional variable to you code, `walls` say, in which you store the number of walls surrounding the robot. Once you have done this you will want to answer the following design questions:

[3]This could be done using four `if (heading==...) System...` but it would be tidier to use a `switch` statement in this part of the code.

Design question 7: Where in the control program do you want to declare this variable?

Design question 8: Where in the control program do you initialise this variable and to what value is it initially set?

Design question 9: What criteria must be fulfilled if values are to be assigned to this variable?

Design question 10: Where in the code is this value inspected and what is the result of this?

This should allow you to establish a design for the second part of this program. Build the code and design some test criteria. You might want to look at how you would test the logical paths of your code - this was previously discussed in section 3.3 of *The Guide*.

It is worth noting that the additional code should be about twenty lines long.

Once you have thoroughly tested your solution so that you are happy that it meets the required behaviour[4], copy your solution to `Ex5.java`.

5.6 Exercise 6

♡ Using the logging from the previous exercise, determine whether the customer was correct in stating that the robot chooses some directions more often than others.

Hint: Notice how often the robot goes `LEFT` compared to how often it goes `RIGHT`, and how often the robot chooses to go `AHEAD` compared to how often it goes `BEHIND`. You may find it helpful to run the controller on different mazes.

Investigating the bias in choice of directions is difficult by hand. However, what we can do is use some additional code to analyse the log which the robot prints out.

We have been supplied with some analysis software (by our sceptical customer). Provided that your logging output from the previous exercise is correct, this software will count the number of times the robot heads in each of the four directions.

You should download this analysis software from the course web page. Here you will find a link to the file called `count`. Right click on this file and *Save Link Target As* to save the file in your `cs118` directory, where you have stored your answers to the exercises and the maze environment. Check that the file has been downloaded properly by typing the command `ls -al` in a terminal window and inspecting the size of the file[5].

The `count` program will examine your output from Exercise 5 and produce a summary of the moves of the robot. For this reason, your output from Exercise 5 must be precisely formatted, otherwise the analysis will not work properly. To run the maze environment

[4]paying careful attention to the layout of the logging output

[5]it should be 337 bytes

alongside the `count` program you should type

```
java -jar maze-environment.jar | ./count
```

The `| ./count` part of the command tells the computer to send your output to the `count` program to be analysed[6].

If you have done everything correctly then the robot should exhibit the same behaviour as before, but will output some different information on the console. This will be the analyser's output, and will look something like:

```
Summary of moves: Forward=1 Left=0 Right=0 Backwards=0
Summary of moves: Forward=2 Left=0 Right=0 Backwards=0
Summary of moves: Forward=3 Left=0 Right=0 Backwards=0
Summary of moves: Forward=4 Left=0 Right=0 Backwards=0
Summary of moves: Forward=4 Left=0 Right=0 Backwards=1
Summary of moves: Forward=4 Left=0 Right=0 Backwards=2
Summary of moves: Forward=4 Left=0 Right=0 Backwards=3
Summary of moves: Forward=5 Left=0 Right=0 Backwards=3
Summary of moves: Forward=5 Left=0 Right=0 Backwards=4
Summary of moves: Forward=6 Left=0 Right=0 Backwards=4
Summary of moves: Forward=7 Left=0 Right=0 Backwards=4
Summary of moves: Forward=8 Left=0 Right=0 Backwards=4
Summary of moves: Forward=9 Left=0 Right=0 Backwards=4
```

Examine whether the summary is actually consistent with the robot's moves. If you find that the output seems wrong, it is most likely that the log of movements you printed for the previous exercise has not been formatted correctly. Perhaps you have forgotten to add a space character, or your output has a typing error...

There is a lesson here in always paying careful attention to the *program specification*. It might appear a minor problem to forget a space in a log of robot movements, or you might decide that the output looks nicer formatted slightly differently. Either way you are dicing with death. Of course when a human looks at the output of your program then they can make allowances for slight variations in output. When the output is read by another computer program however, then we might not have this same flexibility. And of course how are we to know who is going to process the output, it could be a human today and another computer program tomorrow. So rather than risk things going wrong we just stick to the specification. This way we have met our side of the agreement, and if something screws up then this is (hopefully) someone else's problem (legal case, prison sentence or whatever).

If you run tests on a number of mazes for a sufficiently long period of time, then you might notice some pattern in the directions the robot chooses. You should find that the directions chosen by the **DumboController** are indeed random, but that the directions

[6]For more information about this see the excellent *Introducing UNIX and Linux* by some very well known authors; did I say that you should buy a copy of this book...?

are not chosen with the same probability[7].

Investigate the definitions of the `Math.random()` and `Math.round()` methods from the Java API[8]. Using this information state the probability of the robot choosing the directions *left*, *right*, *ahead* and *behind*.

Hint: The Java APIs are an excellent source of code that tens of thousands of programmers draw upon every day. They provide a repository of program code that you can use to construct more complex code. A word of warning though; just because someone else has written the code, doesn't mean that when you employ it you are excused the task of understanding exactly what that code does. Using the `Math.random()` and `Math.round()` methods does indeed allow the robot to generate random directions, but probably not in the same way that you first thought.

5.7 Exercise 7

♡ After further discussion with your customer they submit a revised specification.

> 'Could we have a modified robot controller that chooses directions randomly with equal probability? The robot should still avoid crashing into walls and should still print a log of its movements.'

You should base your solution to this exercise on your answer to Exercise 5. So make sure that you continue working on the `DumboController.java` file and copy your solution to `Ex7.java` when you have finished.

Hint: Review your answer to exercise 6. Consider removing the `Math.round()` call and using the output of the random number generator directly in your selection statements.

5.8 Exercise 8

♡ Use the same method of *design*, *build* and *test* which you followed in Exercise 7 to further modify `DumboController` so that it meets the following customer requirements:

> 'The robot should only change direction if to carry on ahead would cause a collision. As before, when `DumboController` chooses a direction, it should select randomly from all directions which won't cause a collision. Directions should be chosen with equal probability and a log should be kept of the robot's movements.'

[7]If you did not spot this in Exercise 2 then this is the difference between the **DumboController** and the **RandomRobotController**.

[8]For details see `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Math.html#random()` and `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Math.html#round(double)`.

You should base your solution on your answer to exercise 7, so again modify the code in the file `DumboController.java`. The modification you are required to make is again very small. These are the design questions which you need to consider:

Design question 1: How will you instruct the robot controller to check `if` there is a wall ahead before it decides to change direction?

Design question 2: What should the controller do if there is a wall ahead of the robot?

Design question 3: What should the controller do if there is not a wall ahead?

Design question 4: Does your controller program work the very first time it is run? What is the initial value of the variable `direction`?

Copy your answer from `DumboController.java` to the file `Ex8.java`. You will notice that you are building and testing your code using the file `DumboController.java`. This is intended as this allows you to develop code in one file and then use the files `Ex*.java` as backups, for marking purposes, and as the code which you finally ship to the customer.

5.9 Exercise 9

♡ Using your answer to Exercise 8, further modify the controller in `DumboController.java`. This time, as well as displaying all the previous characteristics, the robot controller should also choose a new direction randomly, on average every 1 in 8 moves, irrespective of whether there is a wall ahead or not.

There are two design questions which need to be considered:

Design question 1: How will you get your robot controller to choose a new direction on average every 1 in 8 moves? Hint: you have given this some thought in Exercise 6.

Design question 2: How will you incorporate this into the existing code? Can you combine this new code with your existing `if` statement by employing some logic? Hint: look at the lecture notes and your textbooks for information on logical operators in Java.

Once you have established your design, build the new robot controller. Again, your solution should modify no more than two lines of the controller program, so do not get too carried away.

One example of the resulting behaviour will be that a robot which is travelling forwards in a straight line will occasionally change direction. It may choose to go backwards, continue on its original path forwards, or choose a left or a right turn if these are available. Can this version of **DumboController** be expected to reach the target if given enough time? Test your solution on some suitable examples.

Make a copy of your solution to this exercise by typing

```
cp DumboController.java Ex9.java
```

5.10 Exercise 10

You may have decided while doing the previous exercises that there are many alternative solutions to these problems; you would of course be right. Which solution you ultimately choose is a matter of taste, though later in your programming career you may well find that *in-house styles* or *programming templates* dictate the approach which you use.

Think about an alternative approach to solving one of the previous exercises. Why is your alternative solution better and why?

Chapter 6

Coursework 1 (Part 2): A Homing Robot

In this part of the coursework we get to grips with some slightly more difficult programming tasks, consider the problem of ensuring that software is working correctly, and experiment with additional elements of the Java language.

The main aim of the exercises in this chapter is to guide you through building a robot controller which will make the robot home in on the target. The robot controllers which we have built so far will ensure that the robot eventually reaches the target; however, they will not direct the robot in any meaningful way. The target is reached because the robot chooses directions randomly and if enough random moves are made then the robot will eventually find the target. The trouble with this method is that it may take a long time for the robot to reach the target, particularly if the maze is very large.

If the robot controller is able to sense where the target is located, then a better search technique can be applied. Rather than the robot moving randomly, it could attempt to move closer to the target. This is roughly this technique which we will follow in this chapter.

6.1 Exercise 11

Before you begin building the homing robot, it is worth noting some of the programming errors which you may encounter.

Use your text editor to study the robot controller in the file `Broken.java` which can be down-loaded from the course web page. This robot controller, submitted as an answer to Exercise 4 in Part 1, has two programming errors that the compiler cannot detect. You can compile the code using the command

```
javac -classpath maze-environment.jar Broken.java
```

and then load the **Broken** controller into the robot-maze environment in the usual way. When you try and run the robot you will find that it stalls; it seems not to move and when you press the **Reset** button this is confirmed when it reports that no moves have been made. It is clear that in this case the robot does not meet the customers requirements. We will work through the problems together. First look at the line

```
direction = robot.look(IRobot.EAST);
```

If you study the details of the method `robot.look` in Section 4.2.3 of *The Guide*, then you will find that it returns a result `IRobot.WALL`, `IRobot.BEENBEFORE`, etc. The variable `direction` will be assigned that value.

The first question to ask yourself is '*is this correct?*' Is it right to set the `direction` variable to `IRobot.WALL` or `IRobot.BEENBEFORE`, etc? The answer really depends on what you want to do with that variable. We are given a clue as to its use later on in the program

```
robot.face(direction);
```

This seems to suggest that once the direction is chosen, the robot is then faced in that direction before the robot is finally moved. So now you should ask yourself what happens when the robot tries to face `IRobot.WALL` or `IRobot.BEENBEFORE`?

Whatever the answer is, and I am not really sure, the programmer is using the methods `robot.look` and `robot.face` in a way which makes no sense according to the *programming interface*. What this means is that these methods are strictly defined and serve the purpose of interfacing between the controller software and the robot-maze environment. If these methods are used differently than intended then we can not be sure how these methods will behave.

This might not seem particularly interesting, or indeed important, but adhering to the *interface specification* is crucial if your programs are to work correctly. Even if you have a good feeling about the way in which a method works outside of this specification - and this is justified by a few test calls - if you are using a method differently from the way in which it is specified then you are playing with fire. The method may work well for the first 100 calls and then blow up on the 101st call. Then you are in trouble.

In this example I think the programmer just read the interface specification wrong and decided that a call to `robot.look(IRobot.EAST)` was probably a reasonable thing to do. This is an error which is going to occur a lot in these exercises and you may well be the one who falls into this trap.

You might (reasonably) have expected the compiler to signal an error in this example. After all, the method `robot.look` is expecting something of type `IRobot.AHEAD`, `IRobot.LEFT`, etc. and is passed something of type `IRobot.NORTH`, `IRobot.SOUTH`, etc. But the Java compiler is oblivious to the problem. As far as the compiler is concerned, both these *abstract types* look the same. Both are represented in the program as integer values, and so any type-checking that the compiler performs will just ensure that the value assigned to the variable `direction` and passed to the function `robot.look` is an `int` - which it is.

This problem is an interesting example of the difference between a syntax error (which the Java compiler can detect) and a semantic error (which the Java compiler cannot).

There is one other semantic error in the code. See if you can spot where it is and once you have detected it, correct the program so that it runs in accordance with the specification of Exercise 4.

To make sure that you have a corrected copy of the robot controller as a back-up, copy the file `Broken.java` to `Ex11.java`. Now let's go back to the task of building a homing robot...

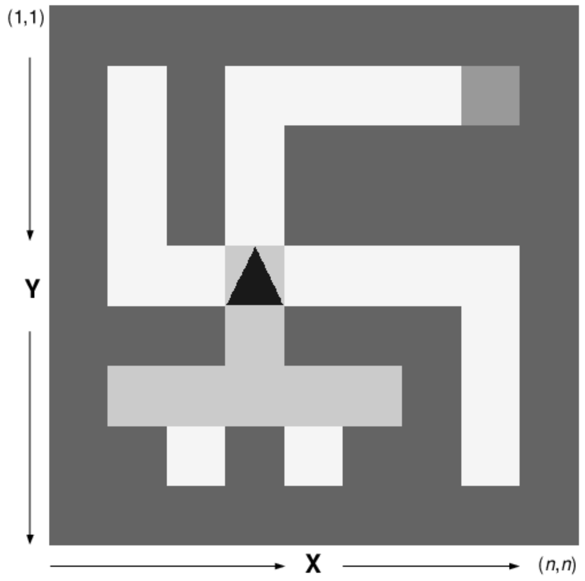


Figure 6.1: The robot homing towards a target that is north-east would choose to go ahead or right as opposed to behind or left. Note the relationship between the x- and y-coordinates of the robot and the target.

6.2 Exercise 12

The controller of the homing robot which we are going to build is based on four separate units of additional controller code: we will call these `northController`, `southController`, `eastController` and `westController`. The names of these control units refer to the heading of the robot at any given step. So, if for example the robot is heading `NORTH`, it will use the `northController` to decide on the next move to make, and so on.

Our new homing robot will choose a direction based on its current heading and the location of the target relative to its current position. For instance, if the robot is heading `NORTH` and the target is to the north of it, as in Figure 6.1, then it makes sense for the robot to select `AHEAD` in preference to `BEHIND`. Based on this assumption, you will construct controller code to determine whether the target is to the *north*, *south*, *east* or *west* of the robot, and then build four additional controllers based on the robot heading that will guide the robot closer to the target. Essentially what the robot is trying to do is 'sense' the target and move towards it if at all possible. A scheme that seems inherently sensible, at least at the outset.

It is possible to decide whether the target is to the north of the robot by examining the y-coordinate of the robot and the target[1]. If the robot's y-coordinate is greater than that

[1]The coordinates begin (1,1) at the top left-hand corner of the maze and increase to (n,n) at the bottom right-hand corner of the maze (the default maze is 15 by 15).

of the target, then the target is north of the robot. Similarly, if the robot's y-coordinate is less than that of the target, then the target is to the south. If the y-coordinates of both the robot and target are the same, then you will find that both the robot and target are on the same latitude.

♡ Add a new method called `isTargetNorth` to the file `Broken.java`. The method should take one parameter (the robot itself[2]) and should return `1` if the target is north of the robot, `-1` if the target is south of the robot and `0` otherwise.

The method should look something like

```
private byte isTargetNorth(IRobot robot)
{
  byte result ...
  // returning 1 for 'yes', -1 for 'no' and 0 for 'same latitude'

  ...

  return result;
}
```

and should not be more than about five lines long.

First sketch your solution on paper, answering the following design questions as you go along:

Design question 1: Where in the `Broken.java` file should you locate this new method?

Design question 2: How can you determine the relative positions of the robot and the target?

Design question 3: What parts of the robot interface can help you in this calculation?

Once your code compiles correctly, you should consider how to go about testing it. Is it possible to develop some exhaustive tests to cover all eventualities? You might want to look back to section 3.3 of *The Guide* to see what testing technique would me more appropriate.

Consider adding some appropriate `System.out.println` statements, and running the robot slowly to examine whether the output makes sense. You might find that moving the target and the robot will help you test more cases. Be prepared to talk about how you tested your solution and why you tested it in that way.

Save your solution as `Ex12.java`.

[2]You will need this parameter if you are to check the y-coordinate of both the robot and target.

6.3 Exercise 13

♡ Use your `isTargetNorth` method as a basis for a second method called `isTargetEast`. This should return `1` if the target is to the east of the robot, `-1` if the target is to the west of the target, and `0` otherwise.

Once you have thoroughly tested your solution, save it to `Ex13.java`.

6.4 Exercise 14

♡ We will now consider the case when the homing robot is heading `NORTH`.

Using the two methods that you developed in Exercises 12 and 13, it is possible to calculate where the target is relative to the current position of the robot. As in Figure 6.1, if we detect that the target is to the north-east of the robot, it makes sense to direct it either ahead or right. That is, as long as there is not a wall in either (or both) of these directions.

Create a `northController` method that exhibits the following behaviour:

> Whenever the robot is heading `NORTH`, the controller should return a direction that will head the robot towards the target if at all possible. This means that: *i)* if it can select direction that will move the robot closer to the target then it should do so; *ii)* it should not lead the robot into a wall; *iii)* if the robot has the choice of more than one route, then it should randomly choose between them.

Before trying to write the software, you must have a clear understanding as to what this specification means exactly.

Design question 1: What should the robot controller do if travelling `AHEAD` or `RIGHT` will move the robot towards the target, and these passages are not blocked by walls?

Design question 2: What should the robot controller do if travelling `AHEAD` or `RIGHT` will move the robot towards the target, and there is a wall ahead of the robot but not to the right?

Design question 3: What should the robot controller do if travelling `AHEAD` or `RIGHT` will move the robot towards the target, and there is a wall to the right of the robot but not ahead?

Try to design your code on paper first. You might find it useful to create a table of the scenarios that the robot might encounter and use this when designing your code.

Save your solution as `Ex14.java`. Remember, no file, no marks.

6.5 Exercise 15

♡ Now that the controller code is becoming more complex, it is important that we test it to see that it meets the desired functionality.

To help with testing, we have constructed a *test harness* that you can use to test your `northController`[3] code.

To access the test harness you need to first download it from the course web page; you will see that it is named `ControlTest.class` and it should be saved to the same directory as your source code.

To call this test harness you need to add a couple of lines of code to your program. The first thing you should do is add a call to the test harness (`ControlTest.test`) just before your robot faces its newly chosen direction and advances, i.e.

```
ControlTest.test(direction,ControlTest.NORTHCONTROLLER,robot);
robot.face(direction);
robot.advance();
```

This test-calling code will check each direction that your robot selects and compare it against a working solution.

Next you need to add some code that will print the log of test results at the end of the robot's run. You should include the code

```
public void reset()
{
  ControlTest.printResults();
}
```

to your `Explorer` class (outside the definition of the `controlRobot` method, yet within the brackets of the `Explorer` code).

These modifications will allow you to test the behaviour of your `northController` method. You should ensure that your robot passes these tests (indicated by a status report of **ok**) before you move on to the next question. Example test reports can be found on the course web page.

Don't take this testing too lightly. If you were a customer buying the controller code then you would be pretty careful to check that it works, particularly if you are paying good money for it. To make the exercise more realistic there will be a pint penalty[4] for any code that it found to be faulty.

6.6 Exercise 16

♡ Complete the corresponding `eastController`, `westController`, and `southController` methods for the cases when the robot is facing in these respective directions.

[3] and also the `eastController`, `southController` and `westController`

[4] To be redeemed at the bar by Dr Jarvis at a time of his pleasing.

You will notice some obvious similarities between the code that you wrote for Exercise 14 and the code needed for these three new methods. There are alternative ways of tackling this exercise: you might decide to cut-and-paste your `northController` three times and make appropriate modifications; a more sophisticated solution would be to extract the common components and encapsulate these as separate utility functions. This second approach is usually a better way of proceeding as it makes the modification of the code (for an upgrade at a later date, for example) easier to achieve. To recognise this fact, more marks will be awarded to solutions that create utility methods for common code blocks.

Make sure that you test[5] your code before saving your solution to `Ex16.java`.

6.7 Exercise 17

♡ This is the cool bit of the first coursework[6]. You should now modify the robot controller so that it uses the four `northController`, `eastController`, `southController` and `westController` methods that you constructed for Exercises 14 and 16.

The robot should use the method appropriate to the direction in which it is currently heading - so for example, if the robot it heading `EAST` then the `eastController` method should be called. These methods should then return the *optimal* direction, which the robot should then take.

This exercise should not be that difficult as you have done most of the hard work already. Save your solution as `Ex17.java`.

6.8 Exercise 18

♡ Can the homing robot always be expected to find the target? Carefully justify your answer.

It is interesting that developing a smarter control algorithm does not actually provide us with a better robot. The random robot is preferable to the homing robot in the sense that it will eventually reach the target, albeit after a very long wait.

Also of interest is the fact that specifying and ordering a homing robot seemed sensible. It is quite possible that a customer, wanting a more sophisticated robot, would request such behaviour, unaware that the resulting robot would not reach the target in some cases.

An answer to this sort of problem is to build a *prototype*. Software developers will often produce some small cheap code to model a potential solution to a coding problem. The code does not need to be shown to the customer, as in itself it is not important. What is important is the input and output behaviour that the code exhibits. A customer

[5]Yeah, yeah, testing bloody testing! You are probably bored to death with this testing mantra by now. However, I predict that at least 50% of people's code will not work at this stage in this coursework. You will not know this yet, because you can't be bothered to **really** test your code, but you will find out later when you try and put the methods together and find that your robot barfs.

[6]cool my arse, I can hear you thinking...

will be shown the prototype and asked to *inspect* the behaviour. It is at this point that the customer can say, 'hang on, this was not what I thought it would do!'

Prototypes are an excellent way of developing potentially expensive software. They ensure that when a customer pays twenty million pounds for some code, it turns out to be what they wanted.

It is quite possible to write prototypes in the Java programming language, but it is often argued that other languages are better suited to the task. For example, *functional* programming languages are favoured for the speed at which software can be developed and the size of the resulting code. They do not produce particularly fast code, but then again at the prototyping stage this probably does not matter. If you are doing the CS course you will be introduced to functional programming languages shortly.

This is the end of the first coursework. This work will be tested on **Wednesday 26 October (Week 5)** in the IBM Lab of the Computer Science building. If you have time, you might like to look through chapters 5 and 6 again just to make sure that you have fulfilled the requirements. Once you have done this you will be required to formally submit your work. Please read the next chapter to see how this is done.

Chapter 7

BOSS: Submitting your coursework

The Department of Computer Science at Warwick has a very sophisticated tool which is used for handling the submission of practical work. There are a number of reasons for using this tool:

- it avoids the impracticality of three hundred people all trying to print out and hand in their coursework at exactly the same time;
- it provides receipts for submitted coursework so we (and you) can be sure that your work has been handed in;
- it allows us to check whether the work has been handed in on time;
- it provides a way of archiving your work;
- it allows post-submission tests to be run on your solutions so that we can detect plagiarism.

You will find that this tool is used for most of the large courses in Computer Science. In order to use the system you must first register.

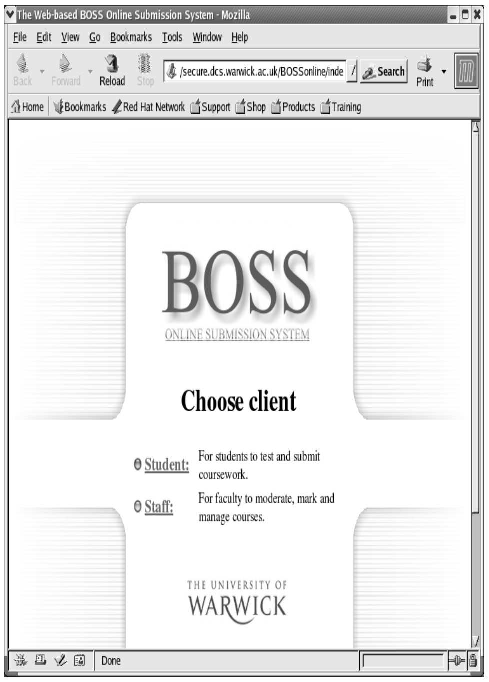
7.1 Registering

You must register with the BOSS system before you can submit your work. Ideally you should try and do this a couple of days before the first deadline, this will ensure that if there are any problems with your registration then they can be sorted out in time. When your work is submitted it is time-stamped and marks automatically deducted if it is late – so if it is late because you did not do your BOSS registration in time, then tough.

You run the BOSS submission system through a web browser. In my experience it works best when you are using Mozilla, but you might want to try using another browser, particularly if you are submitting your coursework from home.

The Web address for the BOSS system is

```
https://secure.dcs.warwick.ac.uk/BOSSonline
```


The Web-based BOSS Online Submission System - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
/secure.dcs.warwick.ac.uk/BOSSonline/inde
Search
Print
Home Bookmarks Red Hat Network Support Shop Products Training
BOSS
ONLINE SUBMISSION SYSTEM
THE UNIVERSITY OF WARWICK
Done

Choose client

Student: For students to test and submit coursework.

Staff: For faculty to moderate, mark and manage courses.

Figure 7.1: Boss on-line.

which should take you to a screen like that in Figure 7.1.

To register you first need to click on the **Student** button which will take you to the *Student client login* screen. To the left you will see the **New user or forgot password?** button which you should press.

Now type in your *University ID* and *Surname* in the appropriate text boxes and press **Submit**. Your submit password will (rather confusingly) be mailed to your IT services account. So once you have registered with the BOSS system you will have to log in to your IT services account to collect it.

Once you have this password you can log in to the submission system. I suggest that the first thing you do is change your password to something you are likely to remember. This can be done using the **Change password** button to the left of the BOSS screen. If you have any problems with registration or the system itself then there are some help pages which you can consult. If the answer to your question is not there then please mail `boss@dcs.warwick.ac.uk` for assistance.

7.2 Submitting your coursework

For the first piece of CS118 coursework you are required to submit a number of files. If you have followed the instructions carefully then you should have files `Ex4.java`, `Ex5.java`, `Ex7.java`, `Ex8.java` and `Ex9.java` for part 1, and files `Ex12.java`, `Ex13.java`, `Ex14.java`, `Ex15.java`, `Ex16.java` and `Ex17.java` for part 2.

Before you submit these files you should edit them to ensure that the class name of each corresponds to the file name; that is, the file `Ex4.java` has the line

```
   public class Ex4
```

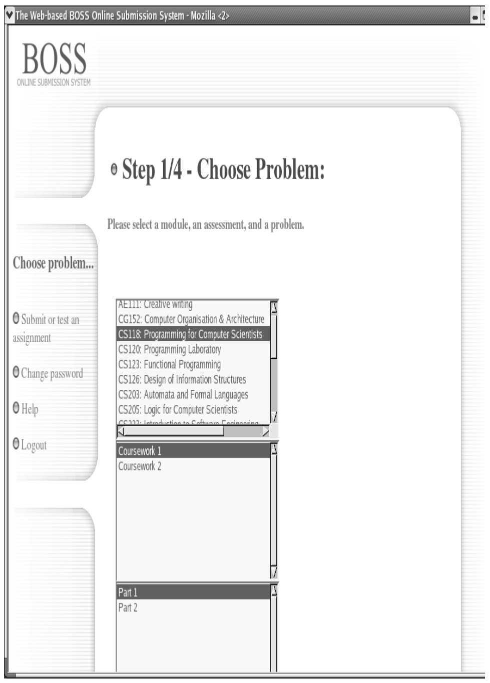
and similarly for the rest.

It is easy to check whether you have done this right. Try compiling the files, e.g.

```
javac -classpath maze-environment.jar Ex4.java
...
javac -classpath maze-environment.jar Ex17.java
```

and if they compile successfully then the files are ready to submit[1].

To submit the files you need to be running the on-line submission system (see above), in which you select the **Submit or test and assignment** option. The submission process is made up of four easy steps. First select the *CS118* course code, then *Coursework 1*, then *Part 1* (see Figure 7.2).


The Web-based BOSS Online Submission System - Mozilla <2>
BOSS
ONLINE SUBMISSION SYSTEM
Choose problem...
Submit or test an assignment
Change password
Help
Logout

Step 1/4 - Choose Problem:

Please select a module, an assessment, and a problem.

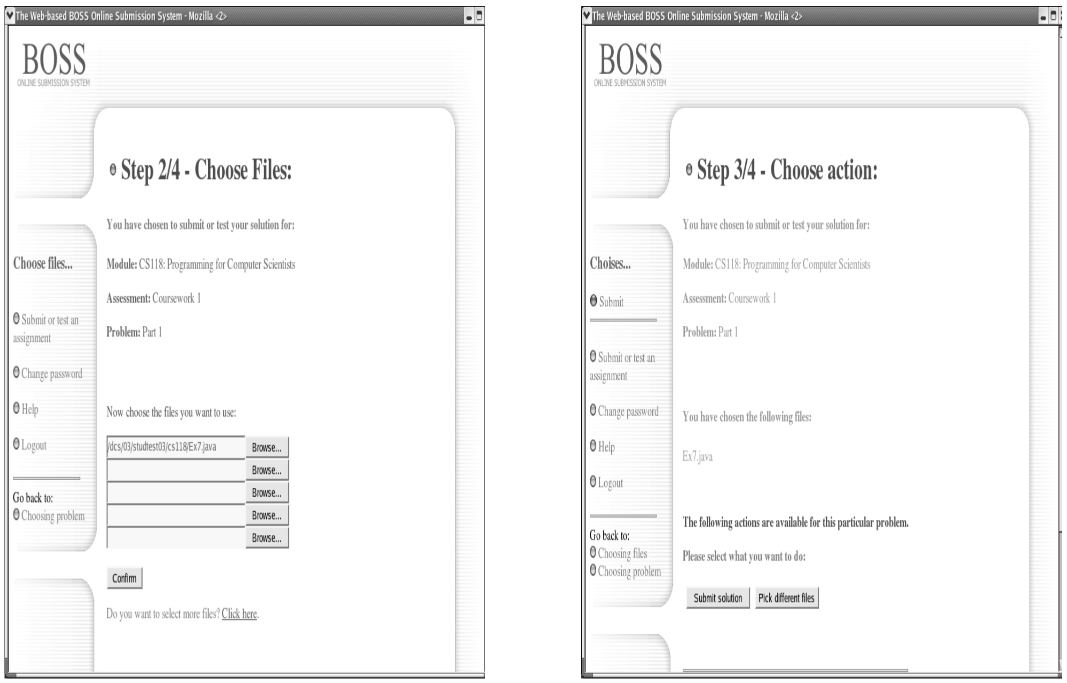
```
AE111: Creative writing
CG152: Computer Organisation & Architecture
CS118: Programming for Computer Scientists
CS120: Programming Laboratory
CS123: Functional Programming
CS126: Design of Information Structures
CS203: Automata and Formal Languages
CS205: Logic for Computer Scientists

Coursework 1
Coursework 2

Part 1
Part 2
```

Figure 7.2: Boss on-line: submission step 1.

At the bottom of the screen you will find a **Confirm** button, press this to go on to the next step. Next you need to choose the files that you want to submit. For this example we choose file `Ex7.java` (see Figure 7.3 – left).

A confirmation screen will give you the details of the file/s that you wish to submit (see Figure 7.3 – right).

[1]It is important to get this right as when you come to get your work tested on the 26 October you will be required to run these files.



Step 2/4 - Choose Files:

You have chosen to submit or test your solution for:

Module: CS118: Programming for Computer Scientists

Assessment: Coursework 1

Problem: Part 1

Now choose the files you want to use:

/dcs/03/studtest03/cs118/Ex7.java

Do you want to select more files? Click here.

Step 3/4 - Choose action:

You have chosen to submit or test your solution for:

Module: CS118: Programming for Computer Scientists

Assessment: Coursework 1

Problem: Part 1

You have chosen the following files:

Ex7.java

The following actions are available for this particular problem.

Please select what you want to do:

Figure 7.3: Boss on-line: submission steps 2 and 3.

If at any stage you get a warning message, for example:

WARNING: According to our records you are not registered for this module

then my advice is just to ignore it.

The final stages of submission require you to formally submit your solution (see Figure 7.4 – left) – at which point you will be informed as to whether your work is late or not – after which your file will be copied to the BOSS file server (see Figure 7.4 – right).

Remember, you must be sure to submit all the files.

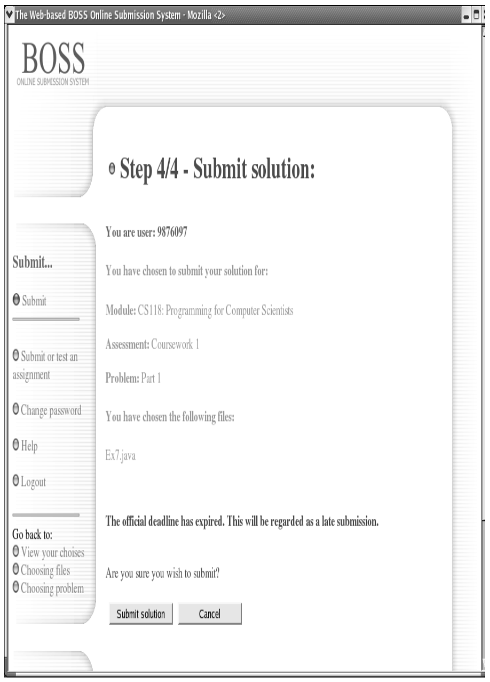
7.3 Receipts of submission

Once you have submitted your coursework you will be emailed (to your IT services account...) receipts of submission, one for each of the submitted files. Do not delete these as they are your only evidence that you have submitted the work.

7.4 The marking process

On Wednesday 26 October (and again later in term) you will be invited into the department to have your coursework marked by one of the seminar tutors. The marking times will be announced nearer the time but the procedure is this:

> The seminar group which you are in will be allocated a 1 hour marking slot. You must get your work marked during this hour otherwise it will not get



Step 4/4 - Submit solution:

You are user: 9876097

You have chosen to submit your solution for:

Module: CS118: Programming for Computer Scientists

Assessment: Coursework 1

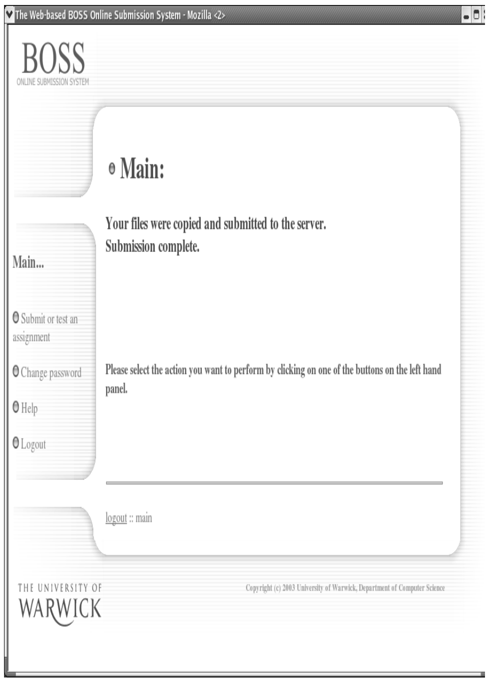
Problem: Part 1

You have chosen the following files:

Ex7.java

The official deadline has expired. This will be regarded as a late submission.

Are you sure you wish to submit?


The Web-based BOSS Online Submission System - Mozilla <2>
BOSS
ONLINE SUBMISSION SYSTEM
Main...
Submit or test an assignment
Change password
Help
Logout
logout :: main
THE UNIVERSITY OF WARWICK

Main:

Your files were copied and submitted to the server.
Submission complete.

Please select the action you want to perform by clicking on one of the buttons on the left hand panel.

Copyright (c) 2003 University of Warwick, Department of Computer Science

Figure 7.4: Boss on-line: submission steps 4 and finish.

marked at all[2]. In order to avoid a crush at the beginning of each hour you should turn up at a time during that hour that corresponds to your surname - *Abbott* in the first five minutes, *Jarvis* about 20 minutes through the hour, *Wenger* at the end of the hour, etc.

This might seem like a completely anal procedure but if it goes right it means that your coursework is marked within about 15 minutes. If it does not go right - because people turn up early or late - it means that you could be in a queue for up to three hours. This did happen a few years ago and since then we have tried to be a bit more organised about things.

The marking process itself is simple. You will be asked to explain your answers to some carefully selected questions; you will be asked to talk us through some of the code; you will also need to run some of your code to prove that it works.

As this takes place the person marking your work will be noting down your marks. This is a completely transparent procedure, so if you are in any doubt as to why a mark was awarded you can just ask. At the end the marks are tallied up and a preliminary mark awarded. You are asked to sign the mark sheet to indicate that you are happy with this preliminary mark. Then you are free to go.

You should take this opportunity to get some direct feedback on your work. If you have lost marks, make sure you find out why. Talk to the marker and see what he/she thinks of the work, where improvements can be made etc. All this will be very useful when it comes to you starting the second coursework.

[2]A seminar tutor will be at the door ensuring that all marking is done at the appropriate time; if, for whatever reason, you feel you can not make your allocated time, you must let me know beforehand.

Provided there are no problems with your work - like we find out you copied someone - then this preliminary mark will be converted to your final mark. If you do not hear anything within three weeks of this marking session you can assume that this is the final mark for this piece of work.

I hope you will agree when the time comes that this is a fair and transparent way of marking your work. You are directly involved in the marking process, you have a chance to discuss the marking with the examiner and are completely free to question any marks awarded. It is only once you and the marker are in agreement on a representative mark that you need leave.

Chapter 8

Coursework 2 (Part 1): Smarter Robot Controllers

In the second coursework you are required to develop sophisticated robot controllers which adopt a more systematic approach to exploring a maze and that learn. In the first part of this coursework we will initially build a controller that systematically searches a dense maze (one that does not contain loops). In such mazes the empty squares form a network (or tree) of corridors one square wide. There are additional exercises at the end of Part 1 whereby we extend this controller so that it is able to deal with loopy mazes[1].

For the second part of the coursework you will be asked to create a robot controller that learns from its previous runs. This means that the more a robot explores (the same maze), the quicker it gets at finding the target.

To complete these exercises you will need to be familiar with the use of arrays in Java. You will find plenty of examples dedicated to the use of arrays in the lecture notes; you will also find good coverage of arrays in the recommended course textbooks.

The control programs which you will write will be more complicated than those from the earlier chapters. As such, you will find that they are much more manageable if you split them up by writing components as separate methods. As a rough guide you should aim to keep each individual method below 30 lines in length. Similarly, you will almost certainly find it worth your while packaging useful bits of program from earlier questions (e.g. choosing a random direction etc.) into methods for later re-use.

In these exercises you are also required to develop your own Java class(es) from scratch. This is an important part of code design and you will find that you can reuse some of these classes in the solutions to the questions in Chapter 9. It is important not to shy away from the inclusion of your own classes. Apart from anything else, it will ensure that you gain a better grade when your solutions are marked.

This coursework, like the first, is split into two parts. You will find that you are guided through the first part of the coursework but not the second. The reason for this is that Part 2 is intended to be tricky and I am not expecting everyone to complete the work. If you finish up to and including Exercise 21 of Part 1, then you will not fail the coursework component of the CS118 course. If your aspirations are to get one of the top grades in this course then you will want to try and complete the latter stages of Part 1 and also

[1]Mathematically this means extending our robot from one that explores *trees* to one that explores *graphs*.

Part 2. Even if you do not complete these remaining exercises, you may find that you pick up marks for a partial solution (or even a non-working solution), so it is worth having a go at these questions if you have time.

All your solutions must be complete by **Friday 2 December (Week 10)**. You should remind yourself of the rules for coursework by taking a quick look at the first page of Chapter 5.

8.1 Exercise 19

♡ An entirely new `Explorer` robot controller is going to be built; this should be done in a file called `Explorer.java`. This new controller should ensure that the robot meets the following specification:

- The robot should never reverse direction except at dead ends.
- At corners it should turn left or right so as to avoid collisions.
- At junctions it should, if possible, turn so as to move into a square that it has not previously explored, choosing randomly if there are more than one. If this is not possible it should randomly choose a direction that doesn't cause a collision.
- Similar behaviour to junctions should be exhibited at crossroads: the robot should select an unexplored exit if possible, selecting randomly between the exits if more than one is possible. If there are no unexplored exits then the robot should randomly choose a direction that doesn't cause a collision.

As the specification suggests, there are four cases to consider, the direction chosen if the robot is: at a dead end; travelling down a corridor; at a junction; and at a crossroads.

We can tell which of these cases we need to consider at any one time by developing a method called `nonwallExits`, running it and observing the result.

Design step 1: Add a method `nonwallExits` to your `Explorer.java` file which returns the number of non-`WALL` squares (exits) adjacent to the square currently occupied by the robot. You will need to use the `robot.look` method and check all four directions. As a guide, your solution should not be more than ten lines of Java code.

You should check that you have not made any syntax errors by compiling your solution. You will need to make sure that you have incorporated the `import` statement at the beginning of the file, you will also need an `Explorer` class which includes an empty `controlRobot` method as well as the new `nonwallExits` method. E.g.

```
import uk.ac...

public class Explorer
{
  public void controlRobot(IRobot robot) {}

  private int nonwallExits (IRobot robot)   // Your new method
  {
    ...
  }
}
```

After some debugging you will find that the compiler no longer complains. Of course you cannot run this program yet as the controller code does not do anything.

If the `nonwallExits` method returns a result that is less than two, then the robot is at a dead end; if the robot is travelling down a corridor, then the number of non-wall exits will be exactly two; if the number of non-wall exits is three then the controller has detected that the robot is at a junction; and finally, if the number of non-wall exits is four then the robot is at a crossroads. See Figure 8.1 for details.

A sensible way to proceed with the development of the explorer robot is to design the method `controlRobot` so that it detects which of these four cases it is dealing with. If the robot is travelling along a corridor, then the control method can pass control to a subsidiary method which determines what to do in this case. Likewise, the dead-end, junction and crossroad cases can be developed in the same way.

Design step 2: Modify the `controlRobot` method so that it records the result of calling the `nonwallExits` method. Store the result in a variable called `exits`.

Design step 3: Now extend the `controlRobot` method so that it passes control to four sensibly named subsidiary methods depending on the value of the `exits` variable.

You will probably want these four subsidiary methods to return direction values to your controller. Your controller therefore, will need to introduce a variable `direction` and assign the result of calling the subsidiary method to that. E.g.

```
direction = deadEnd(robot);
```

The `controlRobot` method will need to execute the commands `robot.face(direction)` and `robot.advance()` before it terminates.

You can compile your changes to check for any syntax errors. You will need to include empty definitions of your subsidiary methods if the compilation is to succeed. The next task is to write the four subsidiary methods. We will look at each of these in turn.

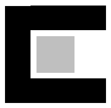
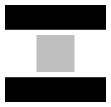
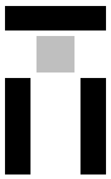
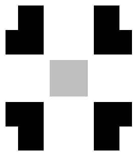
nonwallExits	Maze type	
== 1	Deadend	
== 2	Corridor	
== 3	Junction	
== 4	Crossroad	

Figure 8.1: There is a clear relationship between the number of non-wall exits and the situation which the `Explorer` robot finds itself in; this is demonstrated in the above figure.

Design step 4: *Dead ends* – What should the robot do if it is at a dead end? Well, you are nearly right. It should turn round and head back in the direction it came from in all but one case - when it is at the start. Write the first of the four subsidiary methods to deal with the case when the robot is at a dead end. This will not require too much code as all you need to do is get the robot to find the one and only non-wall route.

Design step 5: *Corridors* – If the robot is travelling down a corridor, or is at a corner, then control should be passed to the corridor subsidiary method. This method will ensure that when in a corridor the robot will not crash into walls (of course) and that it will not reverse direction and go back on itself, since it only does this at dead ends. Write this second subsidiary method; again it should not be longer than about ten lines of code.

Design step 6: *Junctions* – At a junction the robot controller should select a `PASSAGE` exit if one exists. This ensures that the robot explores new parts of the maze in preference to exploring parts of the maze which it has already visited. If there are no passage exits the robot should choose randomly between all non-wall exits.

Design step 7: *Crossroads* – The final subsidiary method is the control code for crossroads. This should exhibit similar behaviour to that of the junction controlling method: selecting an unexplored exit if possible, selecting randomly between these unexplored exits if more than one is possible and, if there are no unexplored exits, randomly selecting a direction that doesn't cause a collision.

You might find it useful to define an additional `passageExits` method. This will be similar to your `nonwallExits` method but instead it will return the number of passage exits in relation to the robot position.

Implementing the junction and crossroad control methods then becomes simple. If there are one or more `PASSAGE` exits then the controller should choose one of the passages randomly; if there are no `PASSAGE` exits then the controller should choose randomly between all non-wall exits.

When you have completed the code you should compile it to remove all the errors. When you have finished this you should have a new `Explorer.class` file which can be loaded into the robot-maze environment. Test your robot controller carefully to ensure that it meets the specification.

When you are satisfied that it works correctly, be sure to save the `Explorer.java` file in `Ex19.java` otherwise you will receive no marks for this exercise.

8.2 Exercise 20

♡ You will notice that the explorer robot is very good when it comes to searching areas of the maze which it has not been to before. However, when part of the maze is thoroughly searched it is unfortunate that the robot goes into *random* mode. It would be better if the robot were able to follow its path back to the point at which it chose between one unexplored path or another. This would enable the robot to backtrack to a previously encountered junction and follow any previously unexplored exits.

This is the behaviour of the robot controller which will be built in the next two exercises of this chapter. In order to do this, the controller `Explorer.java` will be modified so that, whenever a junction is encountered which the robot has not previously encountered in its current run, its location and the direction the robot arrived from are recorded. This information will then be used in the implementation of a *backtracking* routine.

Design step 1: You can easily detect whether a junction or crossroads has already been visited during a robot run by counting the number of adjacent `BEENBEFORE` squares. If there are more than one, the robot Explorer must have visited the junction or crossroads at least once before.

Write a method called `beenbeforeExits` that is similar to the method `passageExits` which you defined in exercise 19. This method will return the number of `BEENBEFORE` squares adjacent to the robot.

Design step 2: The recording of junction and crossroad information[2] will be implemented in a separate class which should be named `RobotData`. This new class can be included as part of the `Explorer.java` file and should contain local state information for each junction the robot encounters.

When a junction is reached your robot should store the x and y-coordinates (to uniquely identify it) and the direction which the robot arrived from when it first encountered this junction. This information can be stored in three arrays

```
private static int maxJunctions = 10000;  // Max. number likely to occur

private static int junctionCounter;  // No. of junctions/crossroads stored

private int[] juncX;          // X-coordinates of the junctions/crossroads
private int[] juncY;          // Y-coordinates of the junctions/crossroads
private int[] arrived;        // Direction the robot first arrived from
```

and an implementation which looked like this would be quite adequate. However, you might decide that an array of `JunctionRecorder` objects or something similar would be a better implementation (and indeed it would).

The coordinates and arrived-from direction for the i-th freshly unencountered junction will be stored in the i-th elements of the arrays. You can do this by using an integer variable (`junctionCounter`, say) to count the number of junctions for which information has been recorded.

On the first pass of a new run `junctionCounter` should be set to 0. This can be done by observing the `robot.getRuns()` method which allows the control program to detect when it is computing the first run through a maze (see section 4.2.9). This value alone is not enough as it will remain 0 throughout the robot's first run through the maze. What you need to do is include a counter which counts the number of times the controller code is polled. A combination of these values will allow you to detect the first move in a first run through a new maze.

The code for this might look something like:

[2] From here on in the text I will refer to junctions/crossroads simply as junctions. Those smart ones amongst you will have realised that there is in fact no difference in the treatment of the two.

```
public class Explorer
{
  private int pollRun = 0;     // Incremented after each pass
  private RobotData robotData; // Data store for junctions
  ...

  public void controlRobot(IRobot robot) {
    ...
    // On the first move of the first run of a new maze
    if ((robot.getRuns() == 0) && (pollRun == 0))
       robotData = new RobotData(); //reset the data store
    ...
    pollRun++; // Increment pollRun so that the data is not
               // reset each time the robot moves
  }
...
```

where the constructor code for `RobotData` does something sensible – such as setting `junctionCounter` to zero, for example.

Complete and insert this code into the `Explorer.java` file.

This is a tricky part of the Explorer code as you are having to manage the introduction of your new `RobotData` class as well as interfacing with the maze-environment itself. In order to pull this off you need to add the method

```
public void reset() { robotData.resetJunctionCounter(); }
```

to the `Explorer` class in which you are developing your controller code, and then add to your `RobotData` class the method

```
public void resetJunctionCounter() { junctionCounter = 0; }
```

What this does is ensure that when you press the **Reset** button in the maze-environment your `junctionCounter` variable will be reset[3].

Design step 3: Now modify the `controlRobot` method so that each time the robot arrives at a previously unencountered junction the coordinates and the direction the robot arrived from are stored in the `junctionCounter` elements of each array. Remember to increase the `junctionCounter` by 1.

A nice way to carry out this recording of junction information is to extend the `RobotData` class so that it includes a `recordJunction` method. This method might take three parameters: the x-coordinate of the junction (which you can access by making a call to

[3]You must follow these instructions otherwise you will get into trouble in the next exercise.

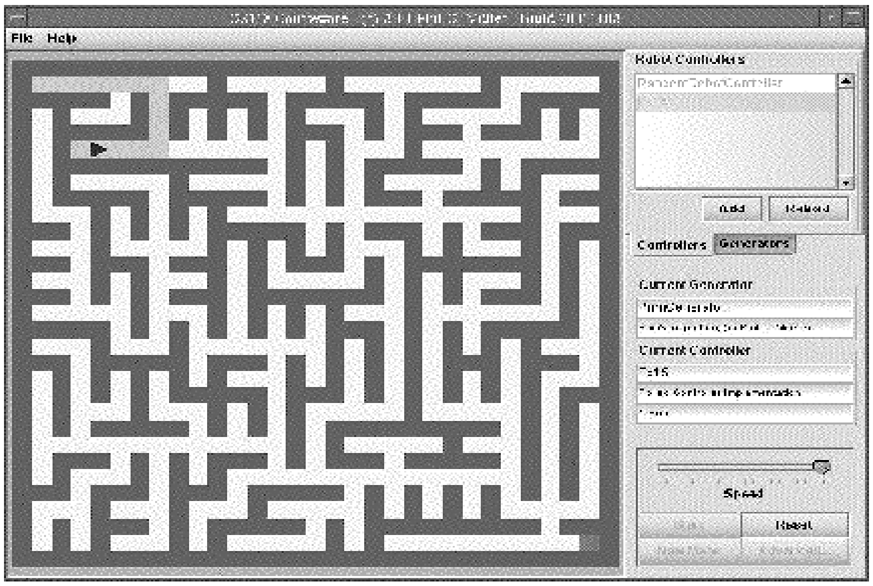


Figure 8.2: The robot has passed through three junctions. The information which is stored in your new `RobotData` class includes the x- and y-coordinates of these junctions as well as the directions the robot arrived at these junctions from.

`robot.getLocationX()`); the y-coordinate of the junction (which you can access by making a call to `robot.getLocationY()`); and the robot heading (which you can access by calling `robot.getHeading()`).

Design step 4: When you have completed these modifications, test that the information recorded is correct by printing it out on the screen (using a `printJunction()` method) and comparing it with the simulation display.

If you are in any doubt as to what the result of this exercise is then consider the following scenario: In figure 8.2 we see that the robot has passed through three junctions. In this case one would expect the robot to record (and print using the `printJunction()` method) the following information:

```
Junction 1 (x=5,y=1) heading EAST
Junction 2 (x=7,y=1) heading EAST
Junction 3 (x=7,y=5) heading SOUTH
```

In order to verify that the output of your program is correct in relation to the movement of the robot, you will have to run your robot very slowly. Reset your robot every now and then and trace through the route of the robot and the output you get from `printJunction()`, just to make sure that the information which you record is correct.

Save your answer to this exercise as `Ex20.java`. Remember, no file, no marks.

8.3 Exercise 21

♡ Modify the **Explorer** robot so that it uses the information it records to perform a systematic search for the target. The specification for the new robot is as follows:

- Initially the robot will *explore*.
- When the robot is *explore*-ing it behaves like the original `Explorer` robot except that when it reaches a dead end or a junction that it has already encountered in the same run, it should reverse direction and *backtrack*.
- If the robot encounters a junction with unexplored exits while *backtrack*-ing it should choose one of these exits randomly and *explore* down it.
- If the robot encounters a junction with no unexplored exits while *backtrack*-ing it should *backtrack* in the direction from which it came when it first reached the junction. This behaviour is termed ‘backtracking through’ a junction.

All this may sound complicated but it is not. What the robot controller is really doing here is switching between two states, the *exploring* state and the *backtracking* state. This switch can be implemented as a state variable of the `Explorer` class,

```
private int explorerMode;  // 1 = explore, 0 = backtrack
```

for example.

Design step 1: Add the `explorerMode` variable to your code and set it in your control code so that when the robot begins a new run the mode is appropriately initialised[4].

We now need to implement two controllers, `exploreControl` for the exploring and `backtrackControl` for the backtracking. The robot will be able to switch between these states depending on the situation.

Design step 2: The method `exploreControl` is pretty much the same code as you used in the previous exercise. You should define a new method called `exploreControl` in your `Explorer` class. When you have done this, cut the explorer code out of `controlRobot` and paste it into `exploreControl`. You will find that the `controlRobot` method then contains just the basic control code which detects if it is a new run etc. and of course, a call to the new `exploreControl` method.

To complete the `exploreControl` method you also need to add the code which sets the `explorerMode` switch to zero when you reach a dead end.

Design step 3: The `backtrackControl` method will require a call to a `searchJunction` method (which should also be defined as part of your `RobotData` class) which is used to

[4]The robot should start off in explorer mode. You can ensure that this is the case by adding an appropriate line of code just after your call to `new RobotData();`. To ensure that you get the same effect when the **Reset** button is pressed, you should also add this line of code to the `reset()` method which you introduced in the previous exercise.

search the RobotData for a junction which has already been encountered. This will return the direction which the robot was travelling when it originally arrived at the junction.

Write the method searchJunction which, when given the x- and y-coordinates of the robot, will return the robot's heading when it first encountered this particular junction. What will this method return if it is called when the robot is at a junction which it has not previously encountered?

Design step 4: Your backtracking control method can now be written. Start by introducing a new method backtrackControl, below your exploreControl method, in the Explorer class. Design your backtrack control method so that it calculates the number of non-wall exits in relation to the robot's position - this is a similar framework to the exploreControl function.

If the number of non-wall exits is greater than two, then the robot is at a junction/crossroads. The backtracking control method then needs to detect if there are any passageExits at this junction: if there are, then the robot must switch back into explorer mode and then proceed down one of these unexplored paths (choosing randomly between them if there are more than one); if there are no passage exits then the robot must exit the junction the opposite way to which it FIRST entered the junction. You can use the searchJunction method to determine the initial heading of the robot when it first entered the junction – the controller should calculate the reverse of this and head the robot in that direction[5].

If the number of non-wall exits is two or less, then the backtracking method should use the existing methods which select a direction at a corridor and select a direction at a dead end respectively.

Design step 5: There is one further design step which you need to make and that is to consider what the controller should do when the robot is at the very first square.

If you have any doubts as to what all this means then talk to your seminar tutor. Time will be set aside in the seminars to discuss these set of problems.

Save your answer to this exercise as Ex21.java. No file; no marks.

8.4 Exercise 22

♡ Will the robot **Explorer** always find the target using this strategy? Can you place a limit on the length of time it will take **Explorer** to find the target? Explain your answers.

The last three questions are intended to be difficult and therefore allow the best programmers to shine. You therefore have a number of options: you could ignore Exercises 23, 24 and 25 and move on to Part 2, this is perfectly acceptable; you might decide to try and answer these remaining questions from Part 1, this is fine, but make sure you

[5]See Section 4.2.4.

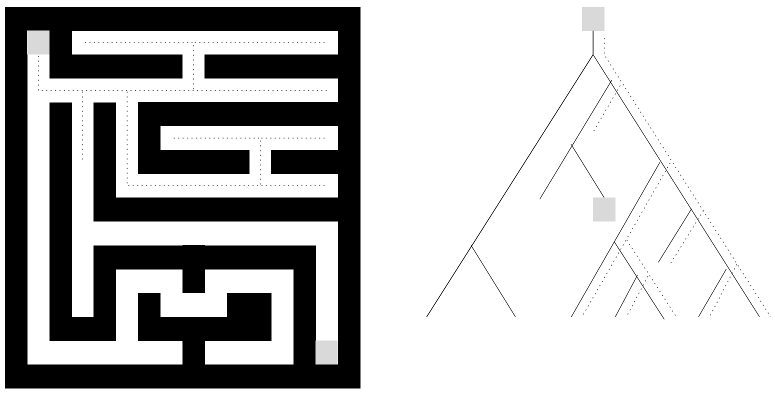


Figure 8.3: Representing the maze as a search tree

leave some time for Part 2 as this will be worth more marks in the long run; you might decide to retire gracefully, this is also perfectly acceptable, although you might want to go back and make sure your code works correctly and is well commented etc. Whatever you decide, the remaining exercises are for the brave. Good luck.

8.5 Exercise 23

♡ In a ‘real life’ situation it may be highly desirable to minimise the amount of data storage required by the control program. Re-implement your robot controller so that the systematic search strategy of the **Explorer** robot does not require the location of each junction to be recorded.

Save your solution to `Ex23.java`.

8.6 Depth-first search in path finding

Throughout this chapter you have been working on the solution to a well-documented *search problem*. These sorts of problems are ubiquitous, cropping up everywhere in Artificial Intelligence and in other areas of Computer Science.

Imagine taking our maze and picking it up from the robot’s start position. You can lift the maze up so that it hangs like a mobile; it will look like an inverted tree (see Figure 8.3). You will notice that each path through the maze becomes a branch in the tree, terminating at a leaf when a dead end is reached. The target will appear on one of the branches in the tree.

Consider what happens when the explorer robot searches the maze. First it will choose one path of the maze. The robot will thoroughly search the part of the maze which this path leads to; any unexplored exits will be searched until, if the target is not detected, the robot backtracks to the junction at which the initial choice was made.

This procedure is analogous to searching one part of the maze-tree. Given that one initial path is as likely to be as good as any other, searching the tree requires picking an alternative at every node in the tree and working forward from that alternative. Other alternatives at the same level are ignored as long as there is a hope of reaching the target using the original choice. This search strategy is known as a *depth-first search.*

The search proceeds to the bottom of the tree if the target is not found; it then backs up to the nearest ancestor node with an unexplored alternative. If this path does not work out then the procedure will move still further back up the tree seeking another viable decision point to move forward from. This process continues until the target is reached or all possible paths in the tree are exhausted.

There are many other search techniques which are used to solve this and similar problems in Computer Science. Some of these search techniques will be more efficient, others will be more suited to finding the *shortest path.* Special-case procedures also exist which are appropriate when facing an adversary. These procedures use *game trees* and are common in computer programs that play board games such as chess for example.

8.7 Exercise 24

♡ Currently our **Explorer** robot uses a depth-first search. This is all very well as long as our maze is non-loopy – as soon as a loop is introduced the exploring algorithm breaks[6].

Modify your **Explorer** robot so that it is able to navigate mazes with single loops.

Save your solution to `Ex24.java`.

8.8 Exercise 25

♡ Extend your answer to the previous exercise so that your robot can navigate mazes with multiple loops.

Save your solution to `Ex25.java`.

Solving loopy mazes has been the subject of much research[7] and as you might expect, someone has already contemplated the problem of navigating around a maze with multiple loops. Indeed a nice solution to this problem was originally published in *Recreations Mathematique* (Volume 1, 1882)[8] by M. Trémaux.

[6]You can test this out for yourself.

[7]From longer ago than you might think — from the ancient Minoans in Crete in about 2000BC, to our very own upper-class and bored Royalty, with nothing better to do than frolic around in daft clothes and in houses large enough to host a maze in their outside privy.

[8]An excellent read for those lonely afternoons on the Arsenal terraces. You might try asking for it in the library.

8.9 Summing up

You have now reached the end of the first set of exercises of coursework 2. Getting this far in the exercises will probably be enough to ensure that you get a pass grade for the practical component of the Programming for Computer Scientists course (providing your code works, has comments, looks nice etc.) Before you go off and celebrate, you must ensure that you submit a copy of the file `Ex21.java`[9].

To do this you should change the class name at the top of the file from `Explorer` to `Ex21` so that it compiles correctly. Then follow the submission instructions found in chapter 7.

[9]And `Ex23.java`, `Ex24.java` and `Ex25.java` if you have done them.

Chapter 9

Coursework 2 (Part 2): The Grand Finale

The exercises in this chapter are designed to test the very best of you. Although it should be said that it is possible to get a good grade in the practical component of the CS118 course without having done the exercises in this chapter; this means that if you do not get a chance to finish this part then you should not panic. You may however find the exercises interesting and decide that you have time to attempt some or all of the questions provided. Even if you do not complete the exercises, you might get some marks for trying. So even if your robot is a bit wonky, you may still gain credit for a part solution.

In this second part of coursework 2 you are required to design, build and test a *learning robot.* You will receive much less step-by-step guidance and as a result I expect to see lots of exciting and innovative solutions. Your solutions will be marked on design, programming style and of course correctness[1].

The programs which you will write for this section are probably the most interesting in this course. You will produce some very clever robot controllers as answers to the questions. It can also be very satisfying to watch the more advanced robots in action.

As a consequence of the work being more difficult, you will receive extra credit for any work you do from Chapter 8. It is very difficult to quantify exactly what this means in terms of *your* marks, as other factors such as programming style, reusability, etc. will play a part. However, you might like to think of these exercises as constituting the difference between an A grade and a B grade in your coursework (depending on where the grade boundaries are drawn[2]).

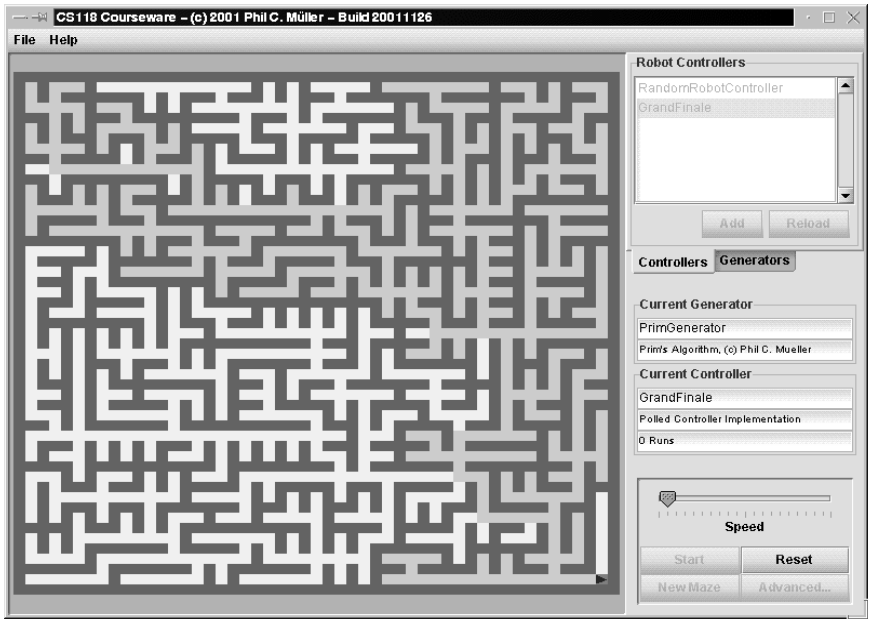
This year there will also be a prize for the person who develops the best solution[3]. While this prize is unlikely to be the model solutions to the CS118 exam, or dinner with Kylie Minogue or John Prescott, it is perhaps the only thing the Department will give you[4].

[1]If your program works then you can be assured of at least 60% of the marks, if you have a smart design then you will bank a further 20%, and if you have programmed like a coding god then you can expect the remaining 20% of the marks.

[2]and whether I have received your cheque or not.

[3]Yes I did say this last year and it is true that there was some problem with the delivery of this prize. In fact the prize got stuck at Bangkok customs.

[4]except for a small certificate and tonnes of grief over the next three years.



CS118 Courseware – (c) 2001 Phil C. Müller – Build 20011126

Robot Controllers
RandomRobotController
GrandFinale

Current Generator
PrimGenerator
Prim's Algorithm, (c) Phil C. Mueller

Current Controller
GrandFinale
Polled Controller Implementation
0 Runs

Figure 9.1: A trace of the **GrandFinale** robot the first run through the maze.

9.1 The *Grand Finale*

♡ You receive a call from NASA. They have seen your robot in action and plan to use some of your code in their next mission[5]. However, while it is clear that your robot searches in a very sophisticated way, NASA point out that it does not learn from its mistakes[6]. What they would like is a robot that learns.

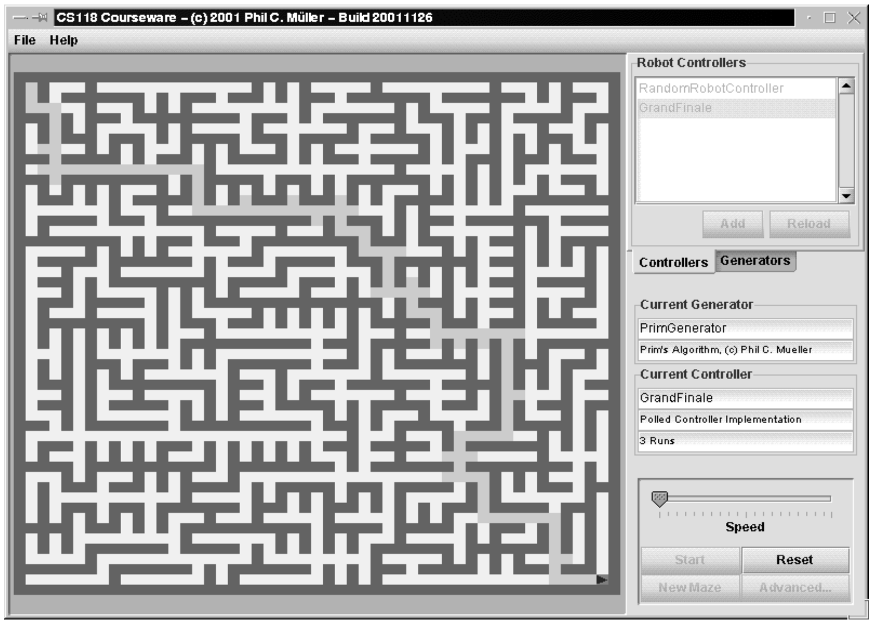
The aim of this last part is to build a **learning robot** that can use information gathered during previous runs through a maze to find a target at increasing speed.

The plan is to build on the solution to exercise 21. The robot will search the maze (in its **Explorer**-like way) the *first time* the robot is run through a maze (as it did in exercise 21) but, the *second time*[7] the robot is run, it will use its virtual map (its memory if you like) to find the target more quickly. What you would expect the robot to do the second time round is exclude the routes through the maze which went nowhere and instead select those which it knows will take it towards the target.

An example of this behaviour is demonstrated in figures 9.1 and 9.2. In figure 9.1 we see the trace of the robot the first time it is run through a fairly extensive maze. As you would expect it is fairly thorough about the areas which it explores, finally however it reaches the target.

When the robot is run again, shown in figure 9.2, the robot can use the information which it stored about the maze during the first run to direct its search for the target. As you see it needs far less exploration the second time round. The aim of the second part of coursework 2 is to model this behaviour.

[5]Heaven forbid.
[6]Unlike NASA, of course.
[7]or more



Current Generator
PrimGenerator
Prim's Algorithm, (c) Phil C. Mueller

Current Controller
GrandFinale
Polled Controller Implementation
3 Runs

Figure 9.2: A trace of the **GrandFinale** robot the second time it runs through the maze.

There is more than one way of doing this and in order to provide you with some (modest) assistance I shall describe two approaches which you might like to take. I do not mind which of these you go for, if any.

9.2 Route A

♡ Modify **Explorer** so that it records the junctions (if any) that each known junction's exit(s) lead to. Test that your program stores the correct information by temporarily modifying it to print the information on the screen and comparing the results with the layout of small test mazes. The information gathered should be retained between runs in the same maze.

You may wish to experiment with the use of complex data structures or arrays of arrays to record this information in a conveniently accessible form. Figure 9.3 shows the sort of data structure which you will need and the extra information which you are likely to store. Note that as well as storing the direction from which the robot entered a junction, the controller also stores the junction which taking a `LEFT`, `RIGHT`, `BEHIND` or `FORWARD` path would lead to.

In the figure you will see that the array index at which the information relating to a particular junction is stored, provides a very convenient means of identifying that junction. Since these identifying integers will be positive (or zero), exits which have yet to be explored can be represented as a negative number. Remember that a method is provided which tells you whether you are computing the first run in a maze, and using this you can detect whether the maze has changed since the previous runs.

The second task is to design, build and test a method which uses the information

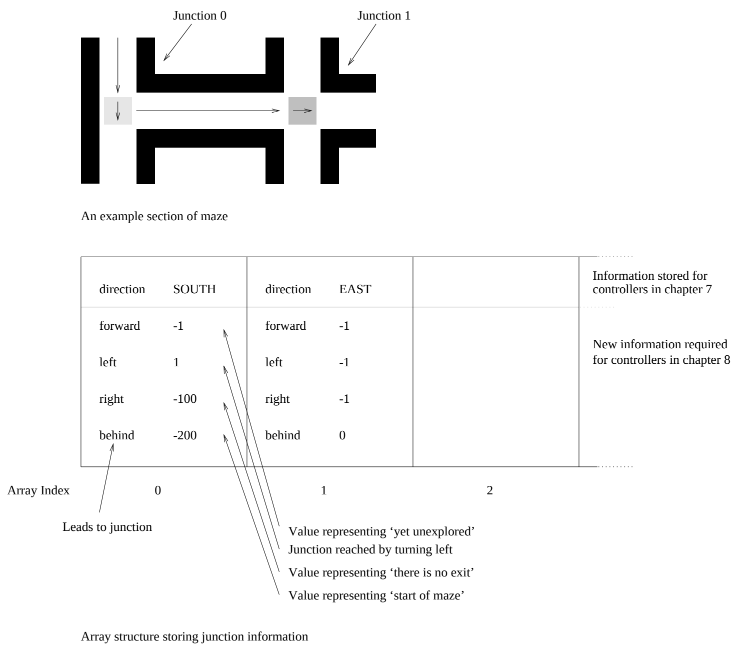

Junction 0
Junction 1
An example section of maze
direction SOUTH
forward -1
left 1
right -100
behind -200
direction EAST
forward -1
left -1
right -1
behind 0
Information stored for controllers in chapter 7
New information required for controllers in chapter 8
Array Index
0
1
2
Leads to junction
Value representing 'yet unexplored'
Junction reached by turning left
Value representing 'there is no exit'
Value representing 'start of maze'
Array structure storing junction information

Figure 9.3: Storing more junction information.

collected by the controller to compute a route between the start and end of the maze.

My advice here is to think first and implement second! Consider how you might represent and compute such a route. Give yourself time to think. Go away from the computer, with pencil and paper (or coffee, or beer, whatever...) to design a good scheme. Then try to implement and test your ideas. If inspiration fails to strike even after thinking hard, your seminar tutor will be able to offer a hint or two. There are quite a number of possible methods which you could use, and each can be implemented as a computer program in many different ways. So, don't panic if your solution sounds completely different to that offered by the seminar tutor - the chances are that you are both right.

Save your working implementation to this exercise in the file `GrandFinale.java`.

9.3 Route B

♡ There is a solution to this problem which many perceive as simpler to that presented in Route A. This route is based on your answer to exercise 23 and so you might like to remind yourself what that was about.

It is possible to store the arrived-from direction and not the x- and y-coordinates of the robot and still build a learning robot. What you need to do in this case is treat the arrived-from directions as a *stack* of values.

The analogy which is often introduced when describing stacks in Computer Science is a pile of dinner plates. If you imagine this pile, then plates can only be introduced at the top of the pile (if you want to avoid any lifting) and similarly taking a plate (in this lazy way) means taking one from the top as opposed to anywhere else in the pile. This method of putting things on and taking things off is described as *last in first out*, and Computer Scientists use the terms *push*ing items onto the stack and *pop*ing them off.

You can use a stack to record the arrived-from directions of the robot. Each time the robot arrives at a junction the arrived-from direction should be pushed onto the stack; when the robot is backtracking the arrived-from directions should be popped off the top of the stack.

If you use this approach you will find that by the time the robot reaches the target the stack will contain a route to the target. The trick is to then use this stack the next time round to direct the robot straight to the target.

Save your working implementation to this exercise in the file `GrandFinale.java`.

9.4 Submitting your coursework

The files which should be submitted for the second coursework are `Ex21.java` and also `GrandFinale.java`; if you have managed to complete some or all of the additional exercises at the end of Part 1 then you should also submit one or more of `Ex22.java`, `Ex24.java` and `Ex25.java`.

In order to make the marking of your work easier, you should make sure that the class names correspond to the file names - that is, file `Ex21.java` contains the definition

```
public class Ex21
```

and similarly for the other files.

To submit the work you should follow the instructions documented in chapter 7. The marking of this coursework will take place on **Friday 2 December (Week 10)**. Be there or be square[8].

9.5 Epilogue

This coursework has touched on a number of different areas of Computer Science. You have learnt something about *specifications* and *refinement*, you have learnt something about *programming* and *software testing*. You will have also touched on *data structures* and *algorithms*, and also *AI*.

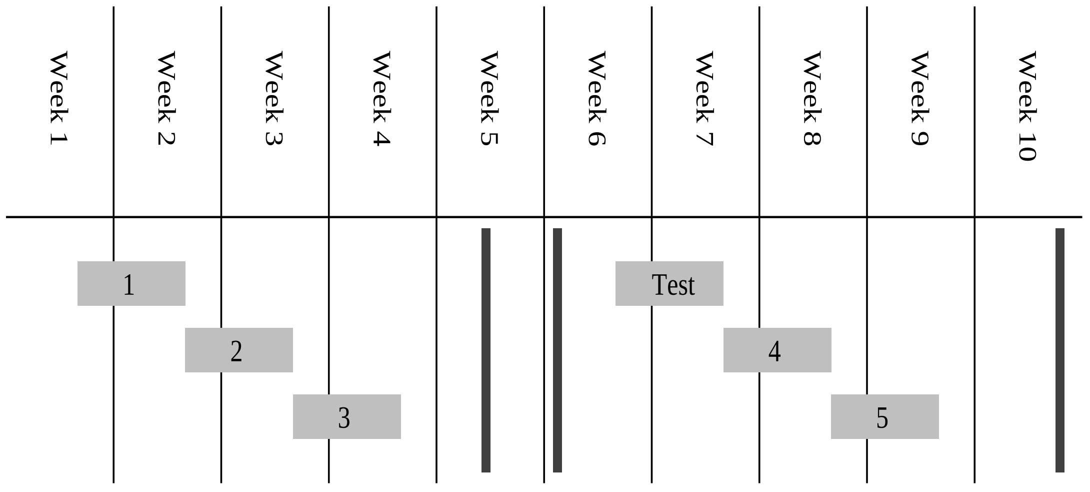
There is much to learn in the remainder of your degree course but this should give you an idea as to how all these areas fit together and how important each is in its own right.

Programming is a complicated business and you are not going to have mastered it in ten weeks. It is a bit like learning to drive - you have probably crashed a few times already, or at least come off the road - and you will get better the more you do. By the summer many of you will be taking up well paid summer jobs fixing peoples' Java code. This may sound hard to believe right now, but each year it is the same; the only thing that changes is that the wages go up!

[8]In fact, be there or get no marks.

Appendix A

Problem Sheets



Note:

Deadline for coursework 1 − Wednesday week 5 (27 October)

Deadline for coursework 2 − Friday week 10 (3 December)

Class test − Monday week 6 − (1 November)

▨ Seminar sheet

Figure A.1: A term planner for CS118, including the deadlines for the two pieces of coursework and also the date of the class test.

The first problem sheet is found in Chapter 2. The seminars for the CS118 course begin on Thursday week 1 (see Computer Science notice boards for details). Because of the way the seminars are timetabled, a *seminar-week* begins on a Thursday at 5pm and ends on a Thursday at 2pm. You can see how the seminar-weeks relate to the problem sheets in the term planner found in figure A.1.

The term planner shows that you should all have finished the first problem sheet by the end of Thursday week 2 at 2pm. The second seminar-week begins on Thursday of week 2 at 5pm and so on. The first problem sheet is to do in the seminar itself; the remaining sheets should be completed before the seminar and handed in so that the seminar tutors can monitor your progress.

You will see from the term planner that there is some respite in the timetable. This will allow us to schedule trouble-shooting sessions for the coursework. If you think that you would like to talk about the coursework (or the marking, or whatever...) before the deadline then you should make sure you talk to your seminar tutor. If there is no interest then the seminar tutors may will not hold classes in seminar-weeks 4, 5 and 9.

Seminar week 6 is dedicated to the class test which you will have done at the beginning of that week. In this seminar you will run through the solutions to these exercises and also receive your mark.

A.1 Problem Sheet 2: Simple statements

A.1.1 Identifiers

Which of the following are not valid identifiers?

1. `hello`
2. `hello2`
3. `hello_3`
4. `4hello`
5. `_hello`
6. `hllo`
7. `$100`

For discussion: The textbook which you are using will probably have a set style which is used for the definition of identifiers. Can you work out what that style is? Is there a minimum or maximum length prescribed or do you think that some other rule is used for deciding what a good identifier should look like?

A.1.2 Data types

1. What are the eight primitive data types in Java?
2. What does primitive mean in this context?
3. For each primitive datatype suggest something from the real world that it would be good at representing.

A.1.3 Arithmetic expressions

Write the following as expressions in Java.

1. An expression for the volume of a rectangular room in terms of its length, width and height;

2. The age of a cat in cat-years in terms of its actual age. (A cat is said to age 15 cat-years in its first year of life and 4 cat-years for each subsequent year. You may assume the cat is at least 1 year old.)

3. $q = -\frac{1}{2}(b + \sqrt{b^2 - 4ac})$ (Hint: `Math.sqrt(x)` calculates the square root of `x`.)

Example: a good answer to 1 is: $volume = length * width * height$

For discussion: The `Math` reference in the question above is to the Java math library. One of the advantages of a language such as Java is the existence of an extensive set of library files (in Java called the API). See if you can find the `Math` library in the Java API.

A.1.4 Boolean expressions

If

```
boolean finished=false;
boolean negative=true;
boolean error=false;
```

state whether each of the expressions below is valid, and if so whether it evaluates to true or false.

1. `finished && negative`

2. `error || finished && negative`

3. `!(5<7)&&(6>=5)`

4. `(5=6)||true`

5. `finished && true && (6)`

Example: 1 is valid and would evaluate to: `false`

A.1.5 Precedence of operators

What is the value of `x` after each statement is performed?

1. `x = 5+2*6;`

2. `x = 2*2+3*4-3%3;`

3. `x = 2*(2+3*4)-3%3;`

4. `x = (4-(3*(2-(2+7*(1-4)))));`

Example: the answer to 1 is: 17

A.1.6 Writing strings and numbers

Given that `int x = 2;`, what would be the output produced by the following lines of code:

1. `System.out.println(x);`
2. `System.out.println("x");`
3. `System.out.println(x + 3);`
4. `System.out.println("x" + "3");`
5. `System.out.println("x + 3 = " + x + 3);`

Example: the answer to 1 is: 2

A.1.7 Strictness

In order to answer this question you will have to revise the definitions of *strict* and *non-strict* from the lecture notes (or from your text book if you have not yet sat through lecture 3).

If `int x = 0;`, what is the result of the following statements:

1. `(x != 0) & ((100/x) != 2)`
2. `(x != 0) && ((100/x) != 2)`
3. `(x > 2) & (x++ < 2)`
4. `(x > 2) && (x++ < 2)`
5. `(x > 2) | (x++ < 2)`
6. `(x > 2) || (x++ < 2)`
7. `(x == 0) || (x++ < 2)`

Example: the answer to 1 is: The program blows up!

For discussion: What do you think are the major benefits of including lazy operators into a programming language? Do you think that it is a sensible thing to do, or can you think of any reasons as to why is might not be such a good idea?

A.1.8 Representable values

Without using you lecture notes, work out what the range of representable values are for the following data types:

1. `byte`
2. `int`
3. `long`
4. `short`

Example: the answer to 1 is: -128 to +127

For discussion: Why do you think Java has so many primitive numeric types? Why not just have an `int` and `float` and have done with it?

A.2 Problem Sheet 3: Control structures

A.2.1 `if` statements

Given that `int x=5` and boolean `okay=false` what will be the output of the following fragments of code:

```
a)
if (x>5) System.out.println("yes");
  else System.out.println("no");
```

```
b)
if (x<=4)
{
  if (x>=6)
  { System.out.println("one");
  }
  else if (x==5) System.out.println("two");
  else System.out.println("three");
}
  else System.out.println("four");
```

```
c)
if (x>7)
if (x==9)
System.out.println("if");
else
System.out.println("dangling else");
System.out.println("hello");
```

```
d)
if (x<7)
{ if (x==9)
  System.out.println("if");
}
else
System.out.println("dangling else");
System.out.println("hello");
```

```
e)
if (okay||(!(!(true)))) if (okay||(true&&!(okay))
&&(false||(true&&false&&true)))System.out.println("one");
else if (okay||(true&&!(okay))&&(false||(true&&
!false&&true))){}else System.out.println("one");
```

Example: the answer to `a)` is: `no`

For discussion: The layout of the control statements in this question varies. Does it make a difference how you go about presenting your code or do you think that it doesn't really matter? Would the introduction of some extra brackets (i.e. { and }) have helped you understand how the code in any of these questions worked?

A.2.2 `switch` statements

The following is an example of a simple user interface:

```
Type 1 to Delete
     2 to Send
     3 to Save
     4 to Quit
```

Given that the user's option has already been read into an `int` called `option`, write a switch statement that prints out a message asking if they are certain. For example if `option` equals 2 then the switch statement should print: Are you sure you want to send? If `option` is not in the range 1-4 then it should write the message: Please type 1, 2, 3 or 4.

For discussion: The use of the `break` statement is sometimes confusing. Are you completely sure that you know how this statement works?

A.2.3 `while` and `do ... while` loops

1. What is the difference between a `while` and a `do ... while` loop? Which have you decided to use in your answer to chapter 5 exercise 4?
2. Write some code that repeatedly reads in an `int option` and only stops if the value 4 (Quit) is entered. Interaction with the code should look like this:

```
Enter an option: 3
Enter an option: 7
Enter an option: 4
Goodbye!
```

3. A student needs to keep track of his finances. Write some code that will allow him to enter his current balance and then repeatedly subtract amounts as he spends money. If at any time his balance is less than his overdraft limit (of 400 pounds) then program must issue him with a warning and then stop. The following is an example interaction with the code:

```
Enter opening balance: 134.34
Enter amount spent: 115.99
Enter amount spent: 87.40
Enter amount spent: 350.00
Warning: your balance is now -419.05. Put some money in the bank!
```

One way to be sure that the code you have written for this exercises is correct is to test it! It is very easy writing code on paper - but very difficult to convince people that it is right if you have not bothered to type it in and run it. If you are avoiding time in front of the computer then you are doing yourself a disservice. Nag nag...

A.2.4 `for` loops

1. Write a `for` loop that reads in an integer from the user and then writes out that number of *s. For example:

```
Enter the number of stars you want: 5
*
*
*
*
*
```

If the user enters zero or less then it should write no *s.

2. Write a `for` loop that calculates the factorial of a number. The factorial of an integer n is the number $1*2*...*n-1*n$, so the factorial of 5 would be $1*2*3*4*5 = 120$.

A.2.5 Hard to spot mistakes

What is wrong with the following bits of code?

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

b) switch(choice)
   case 1:
     System.out.println("choice = 1");
   case 2:
     System.out.println("choice = 2");
   default:
     System.out.println("neither");

c) if (a=10)
   { a = 10+ +b; }
```

```
d) int x=0;
   do
   { System.out.println("x=");
     System.out.println(x);
   }
   while (x<10);

e) for (int x=0; x<10; x++)
   { System.out.println("outer loop");
     for (int x=0; x<10; x++)
       System.out.println("inner loop");
     System.out.println(x);
   }
```

A.2.6 Correctness, preconditions and postconditions

One problem with conditionals in any language is knowing when each part of the code is executed. In complex programs this can be difficult and one way to ensure that your programs are correct is to use *pre-* and *post-conditions.*

Explain the terms *precondition* and *postcondition* using the following code to illustrate your answer. What can you say about the conditions under which p1, p2, p3 and p4 are executed and what can you say about their effect on the state of the program? In this exercise you can ignore any overflow errors.

```
if (a>=0)
        { a=a+1; }      \\ p1
else if ((a<=0)&(b>0))
        { a=a-1; }      \\ p2
else if ((a<0)&(b>0))
        { b=b+1; }      \\ p3
else if ((b>a)&((b+1)==a))
        { b=b-1; }      \\ p4
```

A.2.7 Seminars

This is the end of the third problem sheet and you are now not scheduled to have a seminar until seminar-week 6. You should have mixed feelings about this; great no seminars for a couple of weeks, but how am I going to know whether what I am doing in the coursework is correct?

If you would like a seminar next week (which would probably involve discussing the first coursework) then you must let your seminar tutor know. You might also like to think about the topics which you would like to discuss and let him/her know beforehand.

Make sure you agree at the end of this seminar whether you will be having a seminar next week and if so where it will be held (you might like to meet in the lab for example).

A.3 Problem Sheet 4: Methods, arguments, and scope

A.3.1 Variable scope

What, if anything, is wrong with the following code? If the code does not work, suggest how to fix it. If the program executes, what is it's output?

a)
```
   public static void main(String[] args)
   {
     int n = IO.readint("Enter size: ");
     for (int n = 0; n < 10; n++)
     {
       System.out.println(n);
     }
   }
```

b)
```
   public static void main(String[] args)
   {
     int n = IO.readint("Enter size: ");
     for (int i = 0; i < n; i++)
     {
       System.out.println(n);
     }
     System.out.println(i);
   }
```

c)
```
public class Test2
{

   public static void MyMethod()
   {
     System.out.println(n);
     n = n + 1;
   }

   public static void main(String[] args)
   {
     int n = IO.readint("Enter size: ");
     MyMethod();
     System.out.println(n);
   }
}
```

d)
```
public class Test2
{

   public static void MyMethod(int n)
   {
     System.out.println(n);
     n = n + 1;
   }

   public static void main(String[] args)
   {
     int n = IO.readint("Enter size: ");
     MyMethod(n);
     System.out.println(n);
   }
}
```

e)
```
public class Test2
{

   public static void MyMethod(int n)
   {
     n = n + 1;
     System.out.println(n);
   }

   public static void main(String[] args)
   {
     int n = IO.readint("Enter size: ");
     MyMethod(n);
     System.out.println(n);
   }
}
```

f)
```
public class Test2
{
   public static void MyMethod(int n)
   {
     n = n + 1;
     System.out.println(n);
   }

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

```
  {
    {
      int n = IO.readint("Enter size: ");
      System.out.println(n);
    }
    for (int n = 3; n > 0; n--)
    {
      MyMethod(n);
    }
    int n = 0;
    System.out.println(n);
  }
}
```

For discussion: Are you sure that you know what is meant by *scope*? Scoping rules are responsible for lots of compilation errors when programming and therefore it is important that you know what you are doing in this regard.

A.3.2 Arguments and return types

Imagine that you are developing Java methods to solve a number of small programming problems. When you declare a method there are three important considerations: What the method should be called; what it's return value should be; what argument values it should take.

Suggest solutions to each of these criteria for the following coding problems:

1. Code which will perform multiplication;
2. Code which will print your birthday (Eg. 2 7 1981);
3. Code which will perform factorial;
4. Code which will always produce the number 2000;
5. Code which will always print your name;
6. Code to average your assignment marks.

A.3.3 An exercise in using methods

This exercise is on the simulation of hardware circuits. It will involve building methods for logic gates. If you are not familiar with these you should look them up in any reasonable book on logic or computer hardware. Once you have constructed methods for the logic gates, you will construct a larger piece of hardware - a half adder.

Given the following code:

```
public class Test2
{
  public static void PrintTruthTable(byte x, byte y, byte v)
  {
    System.out.println("X = " + x + " Y= " + y + " V = " + v);
  }

   public static void main(String[] args)
   {
      for (byte x=0; x<=1; x++)
        for (byte y=0; y<=1; y++)
        {
          PrintTruthTable(x,y,AND(x,y));
        }
   }
}
```

1. Write the code for the logic gate `AND` so that the program will execute and produce the output:

   ```
   X = 0 Y= 0 V = 0
   X = 0 Y= 1 V = 0
   X = 1 Y= 0 V = 0
   X = 1 Y= 1 V = 1
   ```

2. Add methods for the logic gates `OR`, `Exclusive-OR`, `NAND` and `NOT`.

3. The sum part for a binary half-adder can be constructed using `AND`, `OR` and `NOT` gates using the following formula

   ```
   SUM = (x .  y') + (x' .  y),
   ```

   where `.` refers to an `AND`, `+` to an `OR` and `'` to a `NOT`. Write a new method which produces a `SUM` for a half adder.

4. The carry part for a half adder can be constructed using an `OR` and `NOT` gates using the following formula

   ```
   CARRY = (x' + y')'.
   ```

   Write a method which produces a `CARRY` for a half adder.

5. Modify `PrintTruthTable` so that it is able to print a truth table for a half adder:

```
X = 0 Y= 0 S = 0 C = 0
X = 0 Y= 1 S = 1 C = 0
X = 1 Y= 0 S = 1 C = 0
X = 1 Y= 1 S = 0 C = 1
```

`PrintTruthTable` should lose no functionality. So think carefully about what you need to do to extend the method.

You should test your solutions before arriving at your seminar. You seminar tutors have been instructed to pick on people, you will not be able to blag your way through a solution. An organised student will have planned the answers to these questions, tested them on the computer and then printed them out for the seminar class. Are you one of these students?

A.3.4 OOP

We have now made the paradigm shift from *procedural programming* to *object oriented programming*. If we are looking for a crude way to describe what this means then we might say that we are extending the type system of our programming language so that it now not only includes primitive types (`int` and `char` etc.) but more complex (object) types as well.

Objects are a pretty smart idea as they not only include data (like a primitive datatype) but they also include the operations (methods) which can be applied to that data.

For discussion: Think about this difference between primitive and object types. If you know any other programming languages do you know whether they include these two styles of datatype? Can you see any advantages/disadvantages to having object types as well as primitive types?

A.4 Problem Sheet 5: Arrays and recursion, abstract classes and interfaces

There are a lot of questions on this last seminar sheet and so you might find that you don't have time to do all of them. This does not mean that you should just pick and choose those questions which you like the look of, rather you should do the exercises that correspond to the material which you have covered in the lectures up to this point in time. For example, you might find that you have not yet covered *interfaces* in the lectures yet; if this is the case then do not do the interface-related question.

Any exercises which you do not cover in the seminar should be done in your own time. It is important not to just ignore these exercises as you may find that they come up in the exam.

A.4.1 Arrays

The following code is a simple procedural type program which sets up an array with fifteen integer values, and also provides a search method for finding a single integer within the array.

```
import uk.ac.warwick.dcs.util.io.IO;

public class ArrayExample
{
  public static boolean search(int[] a, int x)
  {
    for (int i = 0; i < a.length; i++)
    {
      if (a[i] == x) return true;
    }
    return false;
  }

  public static void main(String[] args)
  {
    int input = IO.readint("Enter the number you want to find");
    int myArray[] = {1,5,67,4,456,67,23,7,24,7,86,23,67,4,75};

    if (search(myArray,input)) System.out.println("Found it!");
    else System.out.println("Sorry it's not there!");
  }
}
```

1. Is the array `myArray` passed to the search method by value or by reference? What are the differences between these two modes of variable passing?

2. Write another method called `largestValue` which returns the largest value in the

array. Hint: you now only need one parameter value - the array itself; set the return type to `int`; and consider what the range of integer values is in Java.

3. Further modify the code so that it has the method `smallestValue` which finds the smallest value in the array.

4. Finally modify the program so that it finds the largest interval between two elements in the array. So if the array is [7,1,9,2], the largest interval

   $$max(abs(7-1), abs(7-9), abs(7-2), abs(1,9), abs(1,2), abs(9,2))$$

   is 8. Write a program which produces this result by some efficient means (abs gives the absolute difference; this is written `Math.abs(x)` in Java).

A.4.2 Recursion revisited

We can implement the array searching method recursively. If you think about it all we are doing when we search an array is check the first element, stop if we have found what we are looking, and carry on searching a slightly smaller array if the element is not yet found.

```
public static boolean searchRecurse(int[] a, int x)
{
  if (a[0] == x) return true;
  else
  {
    a = chop(a);
    if (a.length == 0) return false;
    else return searchRecurse(a,x);
  }
}

private static int[] chop (int[] sourceArray)
{
  int targetArray[] = new int[sourceArray.length-1];
  System.arraycopy(sourceArray, 1, targetArray, 0, sourceArray.length-1);
  return targetArray;
}
```

The method called `chop` simply takes an array and chops the first element off the front. So `chop([4,6,78,43])` returns the array [6,78,43]. The method `search` has now been rewritten so that it works recursively. First it checks the first element in the array. If this is the element which is being searched for then it returns `true`. If it is not, then the array is chopped (thus losing the first element) and the search continues on the slightly smaller array. Of course, if the array has been chopped down to length zero, then there are no more elements to search and the method returns `false`.

1. Using the same idea as `searchRecurse` rewrite your `smallestValue` method to `smallestValueRecurse`. There are (at least) two ways to do this:

   You might like to add an extra parameter variable to your new method which represents the smallest value which you have seen so far. This accumulating parameter will contain the smallest value at the end of the recursion.

   Alternatively you might like to write a destructive method which compares each element in the array with the last (`a[a.length-1]`) value, swapping the values if the one at the end is larger. This will ensure that at the end of the recursion, the smallest value is in the last element in the array.

2. Write a similar recursive method for finding the largest value in the array.

3. Now use these two methods to re-implement the interval program above. If you have implemented destructive methods in parts 1 and 2 what extra does your program need to do?

A.4.3 Classes, abstract classes and interfaces

Describe the differences between the following:

1. A Class, as seen throughout the CS118 course;

2. An Abstract Class;

3. An Interface.

Give an example where each might be used. You should illustrate your examples with some outline code.

A.4.4 Programming with abstract classes

Recall the Java code for the abstract class Player

```
abstract class Player
{
  private char piece;

  Player(char c)
  { piece = c; }

  public char playingWith()
  { return piece; }

  public abstract int chooseRow(Board b);

  public abstract int chooseColumn(Board b);

  ...
}
```

Write a definition for an abstract class which satisfies the header below

```
abstract class Exercise2Player extends Player
```

Is it possible to extend this class hierarchy further with a definition which satisfies the header

```
abstract class YetAnotherPlayer extends Exercise2Player
```

A.4.5 Inheritance

What does it mean when we say that Class B inherits method M from Class A?

What does method overriding mean? Illustrate your answer with some example code.

Multiple inheritance is something which Java does not allow. What is multiple inheritance and why might the designers of Java have decided that it was not such a good idea?

A.4.6 Interfaces

Design an interface which meets the following specification.

> *Communication between computers is often described as taking place over a Channel. Channels can either have high-priority (signified by the value 1) or low-priority (signified by the value 0). As well as a priority value, a channel also has a number of operations: send - which takes a message (String) and returns a value which denotes whether the send was successful or not; receive - which takes no arguments and returns a message; setPriority - which will set the internal priority of the system to the argument value supplied (a 1 or a 0); viewPriority - which takes no arguments and returns the internal priority value.*

A.4.7 Seminars

This is the end of the seminar sheets. If you would like a seminar next week, to cover material from the second coursework for example, then make sure you let your seminar tutor know. You might decide to meet in the lab and ask him/her to look through your code and comment on your solutions.