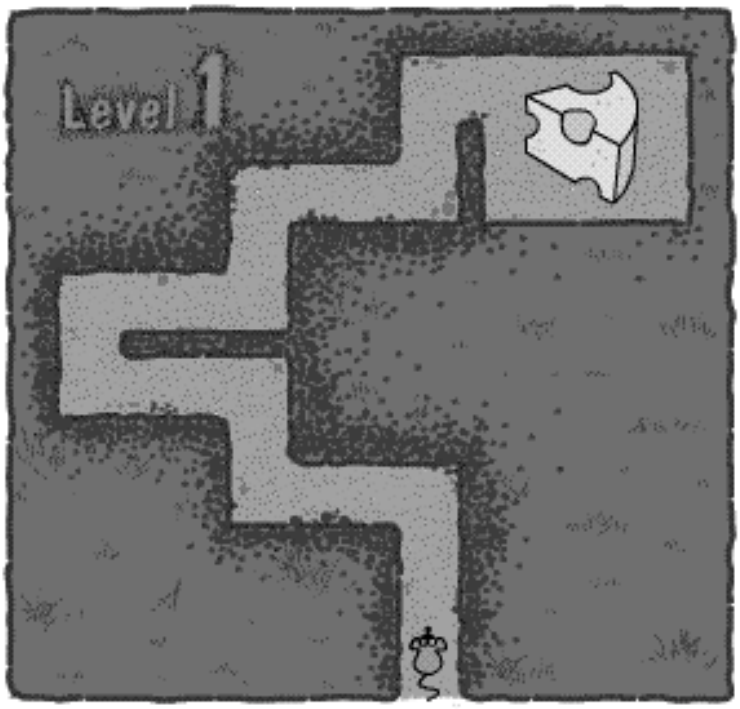
The Guide to

Programming for Computer Scientists (CS118)



Term 1, 2020 - 2021

James Archbold

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, algorithms and artificial intelligence.

1.2 Course structure

1.2.1 Lectures and seminars

The course consists of a number of video lectures, which will cover the topics of both imperative and object oriented programming. These will be released in weekly blocks, and it is important that you stay up-to date. Even if you are already a competent programmer, these videos will guide you on the things you will need to know for the exam and the coursework. The corresponding slides and lecture examples will also be posted each week, allowing you to follow along and make notes. In addition, there will be one live session each week hosted on Microsoft Teams. These sessions will serve as supplementary sessions and will include additional code examples and a Q& A session, where

the questions submitted to the CS118 anonymous form will be answered.

The first live synchronous session will take place on Tuesday of week 1, at 9am. You can find the form to submit questions at: https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs118/questions

You are also required to go to one seminar per week[1]. You will be able to see your seminar group on tabula; you should check tabula to find out when your first seminar is, and the website will list of the details of your seminar tutors. **The seminars are scheduled to begin in week 1.**

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. Throughout the term there will be four optional problem sheets released for you to complete. The seminars are a good opportunity to get assistance on any aspect of these problem sheets or the coursework.

The seminars are **not** optional. Each of you will be monitored to make sure that you are completing the coursework and attending seminars. If you feel unsafe about coming to a lab in person, please either email your tutor, myself or submit a question on the anonymous question form to be answered at the live session each week. Your overall mark for this module will be based on your Exam and your two coursework assignments.

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 complexity.

There are two pieces of coursework. The first coursework must be complete by **Wednesday 4th November (Week 5)** ; the second must be complete by **Thursday 10th December (Week 10)** . In addition to the code, you will be expected to include a preamble that will explain and justify your design decisions. It is not enough to simply produce code, you must be able to explain and present it in a way that can be understood without your presence. Each exercise will discuss what should be included in the preambles for each file.

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.

[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, which session you will attend will be announced in due course.

A record of your work will be stored by the University of Warwick Tabula online submission system. This software allows us to record your progress, justify the mark which you were awarded and detect plagiarism.

More information regarding Tabula can be found at

`http://tabula.warwick.ac.uk/`

1.2.3 Adding it up

The coursework and seminars 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 1 of term 3.

1.3 Computing support

1.3.1 User codes

Before you start programming, 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.

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[2]. 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 when you come to your first seminar**. 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.

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 tutors. 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:

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

`http://go.warwick.ac.uk/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. A download can be found at:

`http://www.oracle.com/technetwork/java/javase/downloads/`

There are Microsoft, Linux and Mac OSX versions of the software at this site and installing the software should be relatively straight forward and should not take you more than about ten minutes. Make sure you download the *Java Development Kit (JDK).*

To run Java on a Windows laptop, open a *Command prompt* window that you should be able to find on most versions of 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\jdk1.12.X\bin`.

The Java package for the robot-maze software which you will use during your coursework can be found on the course web page. Instructions for building your code with the robot-maze software will be set out in due course.

1.4 Books

Programming can be complicated business, but the internet should have plenty of information to ease the learning process. However, some of you may prefer to learn from a introductory text book. 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.

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 0-13-222158-6 (£48.88 on Amazon.co.uk or second-hand for £23)

- *Beginning Java Programming – The Object-Oriented Approach* by Bart Baesens, Aimée Backiel and Seppe vanden Broucke, published by Wrox, ISBN 978-1-118-73949-5 (£19.51 on Amazon.co.uk)

- *Java Programming* by Joyce Farrell, published by Cengage Learning, ISBN 978-1-285-85691-9 (£80.99 on Amazon.co.uk)

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

The other two text books are both recently updated introductory texts and as such should contain the most up-to-date information.

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.

To be honest most of the information that you are going to need can be found on the Web. The only difficulty with this is that it is hard to see the wood for the trees. So a book on programming might be the easiest way to avoid all the noise that you get on the Web.

1.5 Practicalities

If you have any problems regarding CS118 matters you can come and find me in my office (MB3.21) in the Computer Science department.

My email address is `James.Archbold@warwick.ac.uk`.

You will probably find that many of the issues which concern you are also experienced by others on the course. Try talking to your friends and seminar tutors, it is likely they will either have similar problems or will have solutions to these issues.

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 or get depressed. There are a number of procedures in place to catch you before you fall, but you have to be pro-active in finding help.

The seminar tutors are usually pretty good at answering questions. You can email them or 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.

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 Prof. Stephen Jarvis 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

This chapter is intended for use during your first seminar session. Seminars will take place in the programming labs. It is important that you know where you should be each week. At the correct time, find a seat in the lab where your seminar is due to take place. If there are not enough spaces, do not worry, the seminar tutors should make sure that people leave so that only those who need to be in the room are there. In the first seminar, you can 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.

In subsequent labs you are welcome to enter the lab, log in and get started.

2.1 UNIX

All the computers in the Computer Science building run the Linux operating system. We use the CentOS 'flavour' of Linux, though others are available for your laptop (e.g. Ubuntu). Linux is a UNIX-like operating system commonly used in computing industries, especially on server systems where stability and security are key.

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.

♢ I would suggest that you have a look at this book and study the chapter titled 'Getting Started', though I would personally ignore the direction to the text editor known as *vi* and also the section on electronic mail using *mailx*.

2.1.1 Login

When you log in to the computers you will enter a windows-like 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.

2.1.2 Mail

♢ You do not have a mailbox on the CS system. All email will be sent to your University mail account, which you can access from the web using the University's live mail service:

```
http://go.warwick.ac.uk/mymail
```

2.1.3 The Internet

♢ You can invoke a web browser (the default is Google Chrome) by navigating through the menu in the bottom left-hand corner.

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

```
http://go.warwick.ac.uk/cs118/
```

I would suggest that you *Bookmark This Page*.

Any important announcements will be made in lectures and posted to the course webpage.

2.1.4 Editing files

CentOS contains a number of text editing tools. One of these, called *kwrite*, is available from the desktop and will be our editor of choice. Other editing tools can be found from the window manager, or run from a terminal window. When working remotely in a terminal environment, you may like to try out *nano*.

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!");
    }
}
```

Make sure you copy the program exactly (the capitalisation is important). Save the program as `Hello.java` (again, capitalisation is important!) in a folder of your choice (Note: you will need to navigate to this folder in the terminal).

♢ 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.

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.

If at any point you get stuck then ask for some help.

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

```
import java.util.Scanner;
public class Age {
  public static void main(String[] args) {
     Scanner sinput = new Scanner(System.in);
     System.out.println("Enter your age:");
     int age = sinput.nextInt();
     int doubleAge = age * 2;
     System.out.println("How old? That's half way to " + doubleAge + "!");
  }
}
```

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 (including matching capitalisation!).

This program uses the `nextInt` method of the `java.util.Scanner` class. This class makes getting input from the terminal much simpler. It first creates a `Scanner` object, using the Systems input stream (`System.in`), then uses a special method to read an integer into a variable.

♢ The marks for this course might finally be weighted as follows: Coursework 40% (20% for each part); 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[1]. You should not have to create more `Scanner` objects as you can use the same one multiple times (with successive calls to `nextInt`).

2.3 Some UNIX commands

In case you have not had time to learn some UNIX skills yet, here are some useful commands that you might like to type into the terminal window. Write next to each what you think the command does:

```
ls
mkdir test
cd test
cd ..
rmdir test
ls -al
man ls
```

[1]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.

```
passwd
```

Use this last command VERY carefully. You should certainly aim to set your password to something memorable in this session – if you are not yet ready to do this, then type Ctrl-C into the terminal window, and this will kill the command.

2.4 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:

```
http://go.warwick.ac.uk/cs118/
```

If you've missed the first 2 instructions to add this page to your bookmarks, do it now!

It is strongly recommended that you all spend several hours at a terminal during the first two weeks of term getting familiar with Linux, the text editor 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.**

2.5 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.

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 5, and also the first part of Chapter 6. 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).

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 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.

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

- *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 – $((A \wedge B) \vee C)$

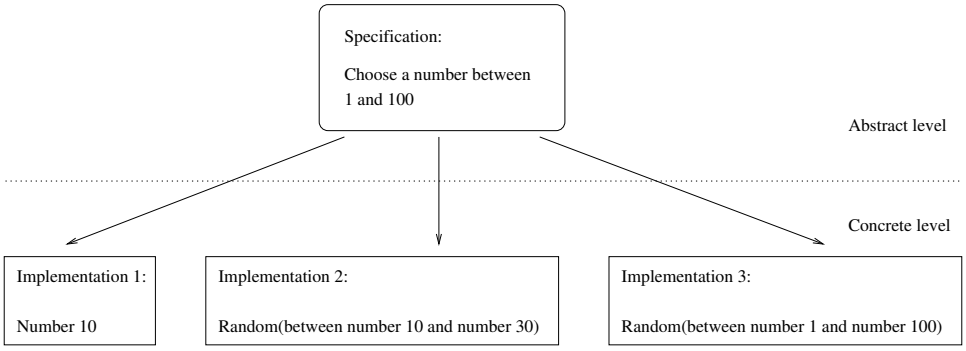


Figure 3.3: Example of abstract- and concrete-level design

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 red bull 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*.

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 a suitable 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 way to test a program is to check all the logical paths. Consider this 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 (which is a special character that cannot be printed to screen).

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 application

A *test application* is a piece of software which will run alongside the program to be tested. The test application may generate test data, supply tests, and record and calibrate the results as the testing takes place. Test applications 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 application 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 Java

This chapter should serve as a Java reference throughout this course (and perhaps the CS126 course later this academic year). In this chapter the basic building blocks of Java programs will be outlined, with some short examples interspersed. If you have a question related to your coursework or exercise sheets, you should consult this chapter before asking for help from your seminar tutor.

Before we start discussing the ins and outs of Java, it's good to have a basic template in which to write all your code. Everything in Java is contained within a "class" (something that will be discussed fully later) and when a program is executed, as you will have done in Chapter 2, the Java environment looks for a particular method for an entry point to your code. For this quick introduction, you should create a file with the .java extension, and you should create a class in this file with a `main` method. The *main* method you write should have a specific function signature, where it should be declared `public`, `static` and `void`, and it should have a single parameter that is an array of String objects. Specifically, your initial class declaration should look something like this,

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

After writing some logic in your main function, you can compile and execute the program as before.

4.1 Variables

In Java, we can use variables to store data, such that it can be reused or changed. Variable must be declared prior to their use and are given a symbolic name, and a *type*. In this course you will use three forms of variables: *primitive* variables, *object* variables and *arrays*.

Primitives

Java contains 8 primitive types that are the building blocks of all other types in Java. There are four *integer* types, two real types, a character type and a boolean type. Specifically the primitive Java types are:

`byte` 8-bit integer type that can represent values between -2^7 and $2^7 - 1$.

`short` 16-bit integer type that can represent values between -2^{15} and $2^{15} - 1$.

`int` 32-bit integer type that can represent values between -2^{31} and $2^{31} - 1$.

`long` 64-bit integer type that can represent values between -2^{63} and $2^{63} - 1$.

`float` 32-bit floating point type that can store approximately 6 digits of precision.

`double` 64-bit floating point type that can store approximately 15 digits of precision.

`char` 16-bit type that can represent any character in the UTF-16 character set.

`boolean` a type that can represent either `true` or `false`.

After deciding on an appropriate type, a variable needs a name, and possibly an initial value. The variable name can be any string of characters and number but cannot start with a number. Furthermore, variables can contain the underscore character (_) and a dollar sign ($), though its generally advised to not use dollar, and to not start a variable name with an underscore. Some names are also reserved; for example, a variable called `int` would be incredibly confusing to the Java compiler, as well as a programmer.

Finally, a variable must be given a value before being used in subsequent statements. Variables are assigned using the *assignment operator*, `=`. Unlike in the world of mathematics, in most programming languages the value on the right, is assigned to the variable named on the left. For example $(1 + 2) \times 3 = x$ would be written in Java as:

```
int x = (1 + 2) * 3;
```

Exercise

Think about what each type above might be used for, e.g., the worlds population would require a `long`, as it exceeds the range of an `int`.

The radius of the earth is approximately 6,371 km; the population is approximately 7,046,000,000 people. Try writing some Java statements to calculate the population density of the earth.

```
long population = 7046000000;
int earthRadius = 6371000;
double pi = Math.PI;
double surfaceArea = ...
```

Objects

There will be a more in-depth discussion of objects later in this guide, but one object variable you will come across is the String type. Object variable types usually start with a capital letter and are built from primitive variables. For instance, the String type stores all its data in arrays of `char` types. Interaction with object types is usually done using the methods contained within the class. For example,

```
String s = "THIS IS A STRING";
System.out.println(s.toLowerCase());
System.out.println("The string is " + s.length() + " characters long");
```

The Java documentation is a great source to find what operations are valid on all of Java's in-built object types. Later in the course you will be designing and using your own object types.

Exercise

Find the Javadoc documentation for the String class using google and find out how to find which character is at position n.

Arrays

Finally, Java contains a special type for storing collections of similar objects. Arrays are declared in the same way as previous variable types but square brackets are used to indicate that there will be a collection of these objects. Further, arrays need to be given a length and from this point accessing individual items requires an index value. For example,

```
int myArray[] = new int[3];
myArray[0] = 3;
myArray[1] = 2;
myArray[2] = 3;
```

The `new` keyword is used to initialise both object types and array types, and with arrays, requires a predefined size. Arrays in nearly all programming language begin at 0 and therefore the last item can be found at the location $n-1$, where n is the length. In Java, the length of an array can be determined by checking the `length` field of the array, e.g., `myArray.length`.

Variable scope

The final thing to say about variables right now is that they are very temporal creatures. They exist for a very limited amount of time, determined by their *scope*. Specifically, from the point at which a variable is declared, it only exists within its enclosing braces. Variables declared outside of methods are called "instance variables" or "class variables". The difference between these two things will be discussed in lectures, but these variables maintain their values between different functions. Consider the following,

```
public class MyClass {
    int instanceVar = 3;

    public static void main(String[] args) {
        int functionVar = 4;
    }

    public int myMethod() {
        int functionVar = 5;
    }
}
```

The variable `instanceVar` exists throughout the whole class with the same value everywhere; a change to its value in `myMethod`, would be reflected within all other methods. The two instances of `functionVar` on the other hand only have scope from the point at which they are declared until the closing brace. Their values are independent and changing the value in `myMethod` will not affect the other variable declared in the `main` method. Variable scope will be covered more thoroughly in lectures; but the take away message is that variables have limited scope and cannot be accessed everywhere. Additionally, two variables with the same name may be allowed to coexist, providing their scopes do not overlap.

4.2 Conditional statements

Java programs are executed in-order, that is to say that when a Java application is executed, each line of code is executed in turn starting with the first statement in the `main` method. There are often occasions where we do not want each line to be executed, for instance if we ask the user for two numbers and want to perform a division with them, if one of them is zero it is likely we do not want to perform a division by zero. To control the flow of Java applications, we use conditional statements.

If-statements

An if-statement is the simplest and most common conditional statement in Java and simply uses a condition that evaluates to true or false and executes code accordingly. To make an application execute one statement or another, an else statement can be attached to the end of the if-statement. Furthermore, you can join an if and an else statement to encode three possible choices. The basic format of an if-statement is:

```
if (condition) {
    ...
} else {
    ...
}
```

For example,

```
if (a < 0) {
    System.out.println("a is less than zero!");
} else if (a == 0) {
    System.out.println("a is equal to zero!");
} else {
    System.out.println("a is more than zero!");
}
```

Conditions can also be chained using boolean algebra, where `&&` indicates a logical-AND, and `||` indicates a logical-OR. Boolean conditions can be "lazy" or "strict", where a single `&` or `|` character indicates strict evaluation and two characters indicate lazy evaluation. The distinction between these will be covered in lectures. When using complicated conditions in if-statements, be sure to bracket carefully to ensure clarity. For example,

```
if ((a == 0) && (b == 0)) {
    System.out.println("Both a and b are 0!");
}
```

Switch-statement

The `switch` statement is similar to an if statement evaluating the value of a specific variable to a specific value. However, the switch statement can behave in a more complicated way by allowing code to 'drop through' many conditions. The basic format of a switch statement is:

```
switch (variable) {
    case 1: ...
        break;
    case 2: ...
        ...
    default: ...
}
```

With a statement of this kind, Java will look at the value of the variable, check if there is a specific case statement for its value, and begin executing instructions from this point onwards. If Java encounters a `break` statement, it will skip to the end of the switch statement and continue with the application. Because a break statement is required to stop code from executing, if there is no break between two cases, Java may execute the code for both cases even through only the first matched the variable. If no cases are matched, the `default` case will be executed, unless it is absent.

Exercise

Try converting the following if statement into a switch statement, using the fact that a switch statement will drop through many conditions when there is no break statement:

```
if ((a == 1) || (a == 2) {
    System.out.println("a is one or two");
} else if (a == 3) {
    System.out.println("a is three");
} else {
    System.out.println("a is neither one, two or three");
}
```

Now try converting the following switch statement into an if statement:

```
switch (a) {
    case 5:
    case 7:
    case 8: System.out.println("a is five, seven or eight");
        break;
    case 1: System.out.println("a is one");
    case 2: System.out.println("a is two");
        break;
    default: System.out.println("a is something else");
}
```

4.3 Iteration statements

While coding Java applications you may find that you end up repeating yourself. Repeated code can be encapsulated into an iteration statement to make Java execute the same code multiple times. Consider,

```
int a = 1;
System.out.println("a is: " + a);
a = a + 1;
System.out.println("a is: " + a);
a = a + 1;
System.out.println("a is: " + a);
a = a + 1;
```

Using an iteration statement this code can be condensed to many fewer lines, and the repeated action can be increased without having to add additional lines of code. As with almost all coding, this can be achieved in multiple ways. In this course we will deal with while loops and for loops, as well as some variants on these.

While-loops

A while loop can be thought of as a repeating if statement, where the body of the if will be continually executed until the condition is no longer met. Specifically they are coded like so:

```
while (condition) {
    ...
}
```

To encode the previous code into a while loop we may do something like the following,

```
int a = 1;
while (a < 4) {
    System.out.println("a is: " + a);
    a = a + 1;
}
```

When using while loops, you should be careful to ensure that the condition changes, otherwise an infinite loop may exist in which the program will continually execute the middle block without ever finishing. Consider what might happen if you removed the statement `a = a + 1` in the example above. The code would print `a is:  1` until the end of time, and this was probably not what was required.

There are some cases where you would like the condition to be evaluated *after* the body has been executed, ensuring that the body will be executed at least once. For this we can use a do-while loop like so:

```
do {
    ...
} while (condition);
```

A good reason to use a statement like this would be if you wanted to assign a value to a variable and change it only under certain conditions. You may find this form of loop beneficial for the robot-maze coursework. Consider the following example,

```
double a;
do {
    a = Math.random();
} while (a < 0.5);
```

This would pick a random number for `a` and then continually pick new a one until `a` is greater than or equal to 0.5.

For-loops

Sometimes you may want a loop to iterate a variable over a very specific range of numbers, or for a specific number of times. While this is possible with a while-loop, for these situations we often use a for loop. A for loop is written using three components: an initial condition, an terminating condition and an iterative operation. For example,

```
for (initial condition; termination condition; iteration operation) {
    ...
}
```

Rewriting the while loop at the beginning of this section as a for loop would make for even more concise code, as the variable declaration and incrementation operation are included as part of the loop.

```
for (int a = 1; a < 4; a = a + 1) {
    System.out.println("a is: " + a);
}
```

Exercises

Rewrite the following for-loop as a while-loop:

```
for (int a = 10; i > 0; i = i-2) {
    System.out.println("i is: " + i);
}
```

Given the following code, copy the characters out of the String `s`, in to the `char` array `c`. Use the Javadoc for String to find out how to retrieve individual letters from `s`. Try using both a while and a for-loop.

```
String s = "This is a string";
char c[] = new char[s.length()];
...
```

4.4 Input and output

Computer systems normally have three "streams" that can perform input and output (I/O) to and from the users console. Two of these streams are for writing data out, and they are the output stream and the error stream, and the final stream is the input stream. In Java these can be found under the `System` class as `System.out`, `System.err` and `System.in`. We've already been writing out to the *output* stream using the `println()` method; you can write out on the error stream in the same way, but instead of `System.out`, we use `System.err.println("...");`. To find more things you can do with `System.out` and `System.err`, you can look up the `PrintStream` Java documentation. You should find a method called `print()` that works like `println()`, but doesn't put a line-break at the end; this can be used to build strings on the output, instead of concatenating things beforehand. For example:

```
String s = "This"
s = s + " is a string";
System.out.println(s);

System.out.print("This");
System.out.print(" is a string");
System.out.println();
```

Both bits of code will produce the same output, but the second can be done over the course of an exercise without subsequent additions being placed on new lines.

For input streams, data has to be read either byte-by-byte or by using another class to do this for us. Luckily, Java provides the `Scanner` class exactly for this purpose. First, a Scanner object is created and assigned to a variable, and then its methods can be called to retrieve user input. For example:

```
Scanner sc = new Scanner(System.in);
long aLong = sc.nextLong();
double aDouble = sc.nextDouble();
```

This will first read in a value as a long and then read a value as a double. Explore the `Scanner` documentation to find more interesting uses of Scanner before proceeding to the following exercise.

Exercise

Using your knowledge of loops and the documentation for Scanner, use the `nextDouble()` and `hasNextDouble()` to read in numbers until something that is not a number is encountered and then output the total of the doubles encountered. Your output should look something like this:

```
Enter double: 4.5
Enter double: 5.2
Enter double: blah
The total of the doubles entered was: 9.7
```

4.5 Methods

You've already been using methods throughout this chapter. You differentiate methods from variables by the use of the following brackets, `()`. Breaking down problems into subproblems, you may find that some operations are repeated; extracting these behaviours into methods allows sensible code reuse, decreasing the amount of coding required and facilitating future extensions. In Java, the terms method, function and subroutine are used interchangeably, though outside of the Java-world, their definitions vary subtly. All methods in Java have a name, a list of parameters and a return type. There is a special type that can be used when no return value is required, we call this type `void`. You've already seen a `void` function in Java, as the `main` method does not return any value to the computer. Further to this, methods also require an *access modifier* (`public`, `private` or `protected`), which we will cover in the next section. The basic format of all function declarations follows the following form:

```
[access] [type] methodName([type] varName, [type] varName, ...) {
    ...
}
```

Where `access` is either `public`, `private`, or `protected`, and `type` is a variable type. Thus, a basic add function that takes two inputs and returns the sum of the two may look something like:

```
public int add(int a, int b) {
    return a + b;
}
```

Here the method is made public and will return an integer value. There are two inputs to the method, called `a` and `b`, and they are also integer values. Java requires that methods which have a non-void return type, explicitly return a value using the `return` keyword. You would call this method and provide two values (or variables) to the method with a function call like so:

```
int c = add(3, 4);
```

As state previously, some methods may not need to return a value and can be declared `void`. For example, if a method writes a String to the screen, it may not need to return a value to the program:

```
public void printError(String errorstring, int errornumber) {
    System.err.println("Error number " + errornumber +
        ": " + errorstring);
}
```

Exercise

Given the formula:

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

Write a function to calculate x. You may want to look up the Math java documentation to find methods for square root and powers. You may also require two functions, one to return the result with a positive determinant, one for the negative case. Unless you're feeling very brave, ignore the cases where there are complex roots.

4.6 Object-oriented programming

In Java, whether you've realised or not, (almost) everything is an object. You've been writing classes, which you can think of as a blueprint for an object. Specifically, an object is an instance of a class and is initialised with the `new` keyword. Consider the Scanner class:

```
Scanner sc = new Scanner(System.in);
```

This is creating an object variable called `sc`, and it is initialising it to be a new instantiation of the Scanner class. Furthermore, it is calling a special method that exists in the Scanner class called a *constructor*. Constructor methods have the same name as the containing class, and have no return type. These methods are used to initialise instance variables (i.e. variables with scope across an entire instance of a class) and potentially call some set up functions. From this point on the object can be used much like the Scanner class is used, calling functions by using the instance variable name, a stop and the name of the function ending with bracketed parameters. Consider the following example:

```
public class MyClass1 {
    public static void main(String[] args) {
        MyClass2 mc = new MyClass2(4);
        System.out.println(mc.getA());
    }
}

class MyClass2 {
    private int a;
    public MyClass2(int a) {
        this.a = a;
    }

    public int getA() {
        return a;
    }
}
```

This example contains two classes, `MyClass1` and `MyClass2`. The first class has a main method and is therefore where execution begins. It creates an object called `mc`, and initialises it using the `MyClass2` constructor method to set the instance variable `a`. As `a` is declared twice in the class, the `this` keyword is used to differentiate between the instance variable `a` and the method variable `a`; `this.a` refers to the instance variable, whereas `a` refers to the parameter passed to the constructor method.

After creating the object, the method `getA()` is used to get the value of `a` and return it such that it can be printed with the `println()` function.

Access modifiers

If you've been paying careful attention so far, you'll have noticed that words such as `private` and `public` have been dotted around before variable and method declarations. These are fundamental features of all object-oriented programming languages and allow us to make use of *encapsulation*. When writing object oriented code, we generally follow the

rule of **private variables, public methods**. What this means is that our class variables must be private and can therefore only be changed by method in the class. Imagine a ATM system that is programmed like this:

```
public class BankAccount {
    String name;
    long accountNumber;
    double balance;
    ...
    public double withdrawFunds(double amount) {
        if (balance > amount) {
            balance = balance - amount;
            return amount;
        } else {
            System.err.println("You do not have enough money!");
            return 0;
        }
    }
}
```

Money is withdrawn from the machine using the `withdrawFunds()` method, which can check the available balance beforehand and warn the user if a transaction is not allowed. In this example the instance variables are not given an access modifier and are therefore implicitly *public* and can be changed by another class like so:

```
BankAccount b = new BankAccount("Bob", 1235323, 1000.0);
b.balance = b.balance - 1100.0;
```

Here, the balance variable has been manipulated directly and so no check has been made to see if the action is valid. If the variables are made *private*, then we can only manipulate them using predefined methods that can perform additional operations. Therefore, it would be better to write this class like so:

```
public class BankAccount {
    private String name;
    private long accountNumber;
    private double balance;
    ...
}
```

This is not to say that all your variables should be private and all your methods should be public. Infact, the safest way to program is to make all things private unless external access is required (i.e. you have another class that needs to perform a particular action).

Inheritance

The final thing to say on object oriented programming in this quick-stop guide is on the subject of *inheritance*. One of the most powerful facets of Java and other object oriented programming languages is that you can create classes that inherit properties and methods from other classes, allowing you to take advantage of existing code and allowing you to build an object hierarchy. This may seem strange but consider the furniture you have at home. Many of the items share some properties and functions but also require some different properties. Let's create a class for furniture.

```
public class Furniture {
    private String material;
    private double weight;
    ...
}
```

Now think about your dining room table and chairs. They're both forms of furniture, but they have different properties, thus we can `extend` the Furniture class and inherit the methods and properties that all items of furniture share, and add our new functions and properties.

```
public class Chair extends Furniture {
    private short numberOfLegs;
    ...
}

public class Table extends Furniture {
    private short numberOfLegs;
    private int length;
    private int width;
    ...
}
```

We can even extend the extended classes, to create a specialisation for an office chair, or a stool, or an exam desk with an integrated chair!

Exercise

Starting with a class for animals, try writing a series of classes for cats, dogs, spiders and flies. It may be beneficial to have intermediate classes such as a Mammals class. Don't worry too much about the implementation of the functions, just think about how properties are shared by all of the subclasses.

4.7 Debugging Java

The final section of this whistle-stop tour of Java is perhaps the most important of all. When you encounter problems with the code you have written, before approaching a seminar tutor, your friends or me, you should attempt to fix the problem yourself. Luckily, Java provides perhaps the best, most informative error messages of all compilers/programming languages. Consider the following code (with added line numbers!):

```
1: public class Test {
2:    public static void main(String[] args) {
3:        System.out.println(a);
4:    }
5: }
```

If we try to compile this code, we get an error message like so:

```
$ javac Test.java
Test.java:3: error: cannot find symbol
 System.out.println(a);
                    ^
  symbol:   variable a
  location: class Test
1 error
```

Here we see that the Java compiler has told us that the error is in Test.java and is on line 3. The error "cannot find symbol" means that a particular variable or method (which Java calls symbols) cannot be found. Furthermore, the compiler tells us which symbol it cannot find (variable `a`)!

Exercise

Fix this error!

The error you just encountered is called a *syntactic* error, and occurs at compile time. There is an error in the syntax of the code and this cannot be understood by the Java compiler. Throughout this course you may also encounter *semantic* errors. These are errors where the meaning of the code has been understood by the compiler, but the logic is incorrect. Consider the following:

```
1: public class Test {
2:    public static void main(String[] args) {
3:        int a[] = new int[2];
4:        a[2] = 3;
5:    }
6: }
```

The code above compiles without error, but when running the program we see that there is an *exception* in the code. Again, Java does a good job telling us about the error:

```
$ javac Test.java
$ java Test
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
 at Test.main(Test.java:4)
```

The error is in the `main` function in Test.java and is on line 4. The exception that was generated was an ArrayIndexOutOfBoundsException (which you can look up in the Javadoc) and has occurred because we are trying to store a number in a location in the array that is out of bounds. If you remember, when we create an array of a particular size, the last index is $n - 1$. Here we're trying to write something into array index 2 (which the error tells us), but the array only contains space for 2 integers, at index 0 and 1.

Exercise

Try writing example applications to perform very simple tasks. While doing this, you will undoubtably encounter errors and you should now be able to identify exactly where and what the errors are. Try to learn about as many error types as possible and how to fix them, this will benefit you greatly in the future.

Chapter 5

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 algorithms. 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.

5.1 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 5.1. 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 control procedure – a method called `controlRobot` – which is run once before the robot makes a move.
- A robot run *usually* 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

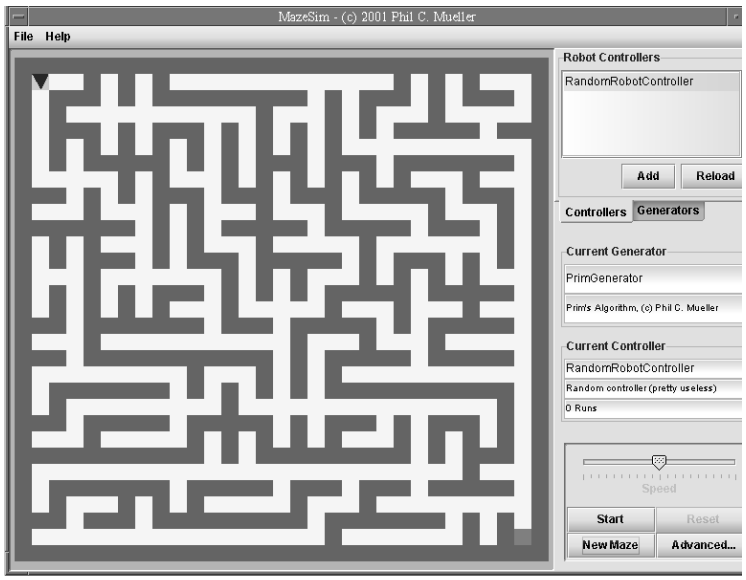


Figure 5.1: The Robot-maze environment

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.

5.2 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.

5.2.1 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`.

5.2.2 Specifying directions relative to the robot heading

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

```
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* (i.e. `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.

5.2.3 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 5.2: Example of sensing robot surroundings

Figure 5.2 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`.

5.2.4 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 5.2 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`).

5.2.5 Sensing the location of the robot and target

The method `robot.getLocation()` returns a `Point` object with two public variable members. You can get the x and y co-ordinates of the robot in the maze from the `Point` class by accessing the `x` and `y` members of the instance using `robot.getLocation().x` and `robot.getLocation().y`. The top left square in the maze is square (1,1).

Similarly, the method `robot.getTargetLocation()` can be used to return the x and y co-ordinates of the robot's target.

5.2.6 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()`.

5.2.7 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 point the robot in a suitable direction. After the completion of your polled controller, the robot will automatically advance in the direction the robot is facing[1].

5.2.8 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 (almost!) 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`.

5.2.9 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.

[1]This is a change from previous years, where your controller would advance the robot manually. If your controller points the robot at a wall, when the environment advances, the robot will cause a collision with the wall.

Chapter 6

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 4th November (Week 5)** . You will be expected to justify key decisions and explain the process of your code in a short preamble at the top of each file.

Your work will of course be marked for functionality, that is ensuring 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. It should also go without saying that you are prohibited from uploading your solutions to the internet.

6.1 Getting started

To begin you need to copy the robot-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 Chapter 2).

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 -l` in your terminal window (you may need to navigate to the correct folder using some of the commands you encountered earlier).

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:

```
-rw-------   1 saw   dcsstaff     697 Aug 11 10:32 DumboController.java
-rw-------   1 saw   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 robot-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
```

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 `&` symbol runs the java command in the background, such that it 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 change the maze, 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. Occaisionally 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.

6.2 Loading controllers

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.

6.3 Analysing code

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 5 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?

6.4 Exercise 1

♢ 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 previously 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 5.2.3 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, this guide 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. If you've carefully planned how your new robot should work, it should only require about two lines of code to be changed or added, 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.

6.4.1 Performance monitoring

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). An alternative approach is to add a logging mechanism, so that

Figure 6.1: Coverage through an example maze.

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 6.1).

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.

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 the previous task) 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 (again, refer back to Section 5.2.3 of *The Guide*!).

For the robot controller to print a log of this direction chosen, you need to include an instruction to output text. You may 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. We could use the `System.out.print()` to build our output string as we go, or we could store `"forward"`, `"backwards"`, etc. in a string and combine all variables later. We could even query the direction variable to find which direction we have chosen to face and use a conditional statement (hint: such as a `switch`) to write out the correct part of the sentence. The main point here, is that there are many ways to achieve the same goal, try a few and pick your favourite.

♢ Design a program which prints the direction the robot is going in.

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 1: 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 2: Is there a corresponding method in the maze-environment package which allows this detection to take place? Hint: see Section 5.2.3.

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:

Design question 3: Where in the control program do you want to declare this variable? Think carefully about the required **scope** of the variable.

Design question 4: Where in the control program do you initialise this variable and to what value is it initially set?

Design question 5: What criteria must be fulfilled if values are to be assigned to this variable?

Design question 6: 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[1], copy your solution to `Ex1.java` (Make sure you change the class name and check your code compiles **after** you've done this).

Exercise 1 Preamble: At the top of your `Ex1.java` file you should include a comment that briefly discuss the problem of avoiding collisions. How do you ensure the robot does not hit a wall? Why did you design your code the way you did? This should not be very long - no more than a couple of sentences.

6.4.2 Assessing performance

♢ 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. Perhaps even a blank maze?

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.

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

You should download this analysis software from the course web page. Here you will find a link to the file called `count.pl`. 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.

The `count` program will examine your output from the previous custome request and produce a summary of the moves of the robot. For this reason, your output must be precisely formatted, otherwise the analysis will not work properly. To run the maze environment alongside the `count` program you should type

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

If you are using your own computer for this assignment, with the Windows operating system, you will need to download a different count application, and it is used in a slightly different way. Instruction for this are posted on the course web-page.

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

If you have done everything correctly[2] 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

[2]If, for some reason, this does not work, do two things: 1. check that the file you have saved is called `count.pl` and if it is not do a `mv x count.pl`, where `x` is the name of the file you saved; 2. make sure that it executes correctly, you can ensure this by typing `chmod u+x count.pl` in the command window.

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 are not chosen with the same probability[3].

♢ Investigate the definitions of the `Math.random()` and `Math.round()` methods from the Java API. 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.

6.5 Exercise 2

♢ 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 the previous exercise. So make sure that you continue working on the `DumboController.java` file. You should also remember to keep regular back-ups at this point.

Hint: Review your answer to Exercise 1. There are many ways to accomplish this task; think carefully about the use of the `Math.random()` and `Math.round()` functions.

6.5.1 Improving performance?

♢ Use the same method of *design*, *build* and *test* and 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.'

[3]If you did not spot this previously then this is the difference between the **DumboController** and the **RandomRobotController**.

Your solution should be building upon the work you have done previously, 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`?

6.5.2 Reaching the end

You may notice your robot has some issues after implementing the last algorithm. To correct this issue, it is decided some elements of randomness will be added back into the robot controller.

♢ Again building on your previous work, modify the controller in `DumboController.java`. This time, as well as displaying all the previous characteristics, the robot controller should also have a chance to choose a new direction randomly, irrespective of whether there is a wall ahead or not. This chance should result in the robot selecting a random direction *on average* every 1 in 8 moves. It should be a *chance*, meaning that the robot may choose a random direction 2 moves in a row, or not for 20 moves, but, over a long time, the proportion of random moves should be about 1 in 8.

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 earlier.

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 boolean algebra? 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 final solution to this exercise by typing

```
cp DumboController.java Ex2.java
```

Exercise 2 Preamble: At the top of your `Ex2.java` file you should include a comment that briefly discusses the problems solved during this exercise. In particular, you should explain your method of ensuring 'fair' probabilities and justify how it ensures even probabilities. Include a discussion of the initial issue that had to be solved - what caused the initial uneven probabilities? Also discuss how you decided to incorporate the 1-in-8 chance into your design. Was this a disruptive change? Do you think it could be done in a better way? This preamble will likely be longer than your previous preamble, but should still be no more than two short paragraphs.

6.6 Final words

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?

Important note

Before moving onto Part 2, ensure you have correctly renamed all of your files and classes ready for submission. The marking process for this module is partially automated and any compile errors will result in a loss of marks. Additionally, ensure that you haven't accidentally pasted rubbish into your files by accidentally hitting the middle click button.

Chapter 7

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 the technique which we will follow in this chapter.

7.1 Starting again

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 downloaded from the course web page. This robot controller 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 1.

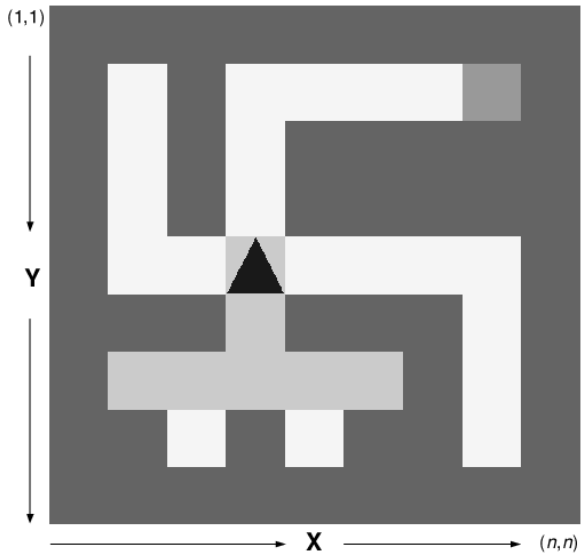


Figure 7.1: The robot homing towards a target that is north-east would choose to go ahead (north) or right (east) as opposed to behind (south) or left (west). Note the relationship between the x- and y-coordinates of the robot and the target.

7.2 Exercise 3

The controller of the homing robot which we are going to build is based on a heading controller.

Our new homing robot will choose a heading based on its current location and the location of the target. For instance, if the robot is heading `NORTH` and the target is to the north of it, as in Figure 7.1, then it makes sense for the robot to select `NORTH` in preference to `SOUTH`. 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 a heading controller 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.

7.2.1 Finding the target

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 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.

[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).

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

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.

♢ 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.

7.2.2 Sensing your environment

Currently the robot can sense its environment using the `look()` function. The function takes *relative* directions in order to operate correctly. The functions you have just written return the target's position in *absolute* directions and your controller will need to give the robot an *absolute* direction, so it makes sense that you sense the environment using *absolute* directions.

♢ Write a method called `lookHeading` that takes an *absolute* direction and returns whether there is a `WALL`, a `PASSAGE` or a `BEENBEFORE` square, much like the current `look()` function. **Hint:** You may need to pass the robot object to the function in addition to a heading.

7.2.3 Building a heading controller

Using the methods that you have developed so far, it is possible to calculate where the target is relative to the current position of the robot and to analyse the robots surround-

ings. As in Figure 7.1, if we detect that the target is to the north-east of the robot, it makes sense to direct it either north or east. That is, as long as there is not a wall in either (or both) of these headings.

♢ Create a `headingController` method that exhibits the following behaviour:

> Given the state of the robot in the maze, the controller should **return** a heading that will head the robot towards the target if at all possible. This means that: (*i*) if it can select a heading 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; (*iv*) if there are no headings that will move the robot towards the target, pick randomly between all available headings.

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 `NORTH` or `EAST` 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 `NORTH` or `EAST` will move the robot towards the target, and there is a wall to the north but not to the east?

Design question 3: What should the robot controller do if travelling `NORTH` or `EAST` will move the robot towards the target, and there is a wall to the east of the robot but not to the north?

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.

7.2.4 Testing your solution

♢ 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 `headingController`.

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 sets its heading to the newly chosen direction and advances, i.e.

```
...
heading = headingController(robot);
ControlTest.test(heading, robot);
robot.setHeading(heading);
...
```

This test-calling code will check each heading 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 class (outside the definition of the `controlRobot` method, yet within the brackets of the whole class).

These modifications will allow you to test the behaviour of your `headingController` method. You should ensure that your robot passes these tests (indicated by a status report of **ok**). 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.
After thoroughly testing your solution, save it as `Ex3.java`.

Exercise 3 Preamble: At the top of your `Ex3.java` file you should include a comment that discusses key elements of this exercise. How did you choose your design for your heading controller? Does it ensure that the robot always moves, when possible, closer to the target? Furthermore, can the homing robot always be expected to find the target? Carefully justify your answer. What improvements would you suggest for the robot? This preamble should be of a similar size as exercise 2.

7.3 Final remarks

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 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.

This is the end of the first coursework. Submit your work through the Tabula submission system, making sure to include the following files:

`Ex1.java, Ex2.java, Ex3.java`

Don't forget that your deadline for submission is **Wednesday 4th November (Week 5)** .

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 code 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 challenge the best students. Even if you do not complete these every exercises, you may find that you pick up marks for partial solutions (or even non-working solutions), so it is worth having a go at these questions if you have time.

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

All your solutions must be complete by **Thursday 10th December (Week 10)** . You should remind yourself of the rules for coursework by taking a quick look at the first page of Chapter 6.

8.1 Exercise 1

♢ 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 been 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 command `robot.face(direction)` 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.

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.

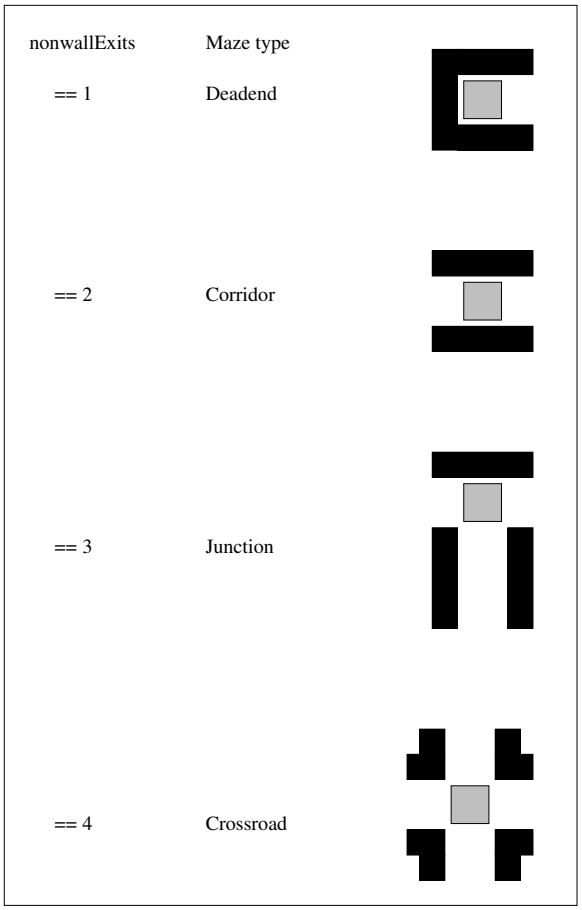


Figure 8.1: There is a clear relationship between the number of non-wall exits and the situation which the `Explorer` robot finds itself in; this is demonstrated in the above figure.

Design step 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.

8.1.1 Storing data

♢ 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 parts 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 heading 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 previously. 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 *absolute* direction which the robot arrived from when it first encountered this junction. This information can be stored in three arrays, i.e.,

```
private static int maxJunctions = 10000; // Max number likely to occur
private static int junctionCounter; // No. of junctions stored
private int[] juncX;            // X-coordinates of the junctions
private int[] juncY;            // Y-coordinates of the junctions
private int[] arrived;          // Heading the robot first arrived from
```

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.

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

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:

```
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].

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

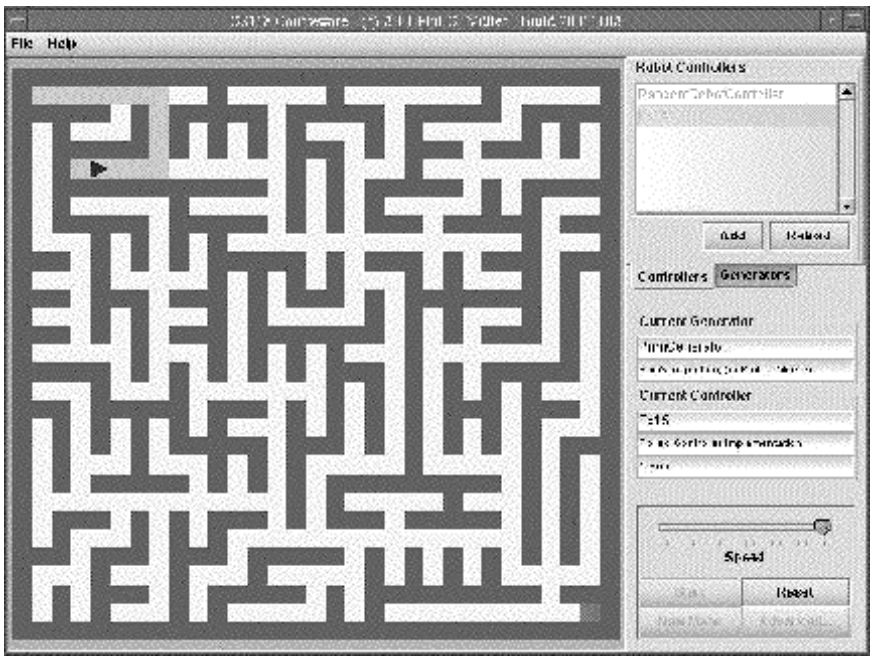


Figure 8.2: The robot has passed through three junctions. The information which is stored in your new `RobotData` class includes the x- and y-coordinates of these junctions as well as the heading of the robot as it arrived at these junctions.

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 `robot.getLocation().x`); the y-coordinate of the junction (which you can access by making a call to `robot.getLocation().y`); 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.

8.1.2 Using stored data

♢ 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` and `backtrackControl`. 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 previously. 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 search the `RobotData` for a junction which has already been encountered. This will return the direction that 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.

[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.

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 or crossroad. 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 `Ex1.java`. No file; no marks.

8.1.3 Worst case analysis

♢ Will the robot **Explorer** always find the target using this strategy? Can you place a limit on the length of time (or maximum number of steps) it will take **Explorer** to find the target?

Exercise 1 Preamble: At the top of your `Ex1.java` you should include a short preamble that justifies your design decisions during this exercise. In particular, discuss your handling of `passageExits`, `nonwallExits` and the four controller methods. How did you ensure your program is efficient? You should discuss your chosen implementation of the RobotData class and how it is utilised by the backtracking. Discuss your reasoning for how you've designed your explorer and backtracker. Is there repeated code anywhere? Could some functionality have been combined in a better way? Finally, you should include your answer to the worst case analysis question above.

[5] See Section 5.2.4.

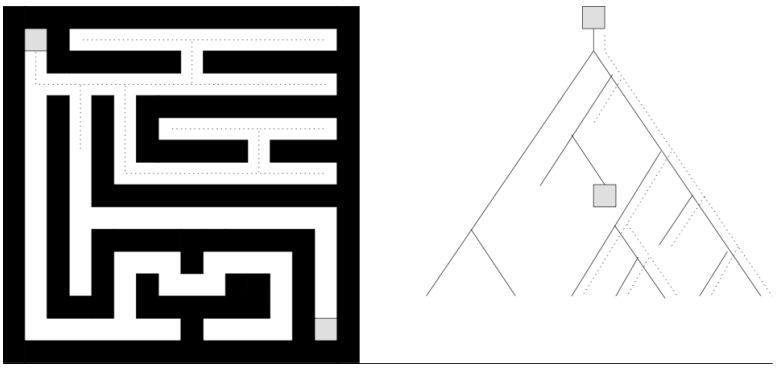


Figure 8.3: Representing the maze as a search tree

8.2 Exercise 2

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 `Ex2.java`.
Exercise 1 Preamble: At the top of your `Ex2.java` you should include a short preamble that simply explains how your implementation saves space.

8.3 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.

8.4 Exercise 3

8.4.1 Single loops

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. Some of you may find it easier to work from your Exercise 1 answer, others may find it easier to extend from Exercise 2.

8.4.2 Multiple loops

♢ Extend your answer so that your robot can navigate mazes with multiple loops.

Save your solution to `Ex3.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) by M. Trémaux.

Exercise 3 Preamble: At the top of your `Ex3.java` you should include a short preamble that justifies your design decisions during this exercise. In particular, discuss you should highlight why your previous robot was incapable of solving loopy mazes, and how your new implementation corrects that fault.

8.5 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

[6] You can test this out for yourself.

[7] From longer ago than you might think – from the ancient Minoans in Crete in about 2000 BC, 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.

ensure that you submit a copy of the files `Ex1.java`, `Ex2.java`, and `Ex3.java`.

Make sure that you change the class name at the top of each file from `Explorer` to `Ex*` such that they compile correctly. Then submit your solutions through Tabula.

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 this chapter. 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).

This year there will also be a prize for the person who develops the best 'robot of the year'. You could win this by going beyond the specification, or doing something unique within the robot maze environment. Previous winners have turned the maze environment into a game, and created interesting graphical displays to show the maze-tree as it was being explored. Last year the winner took webcam input and made it display as a maze!

[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.

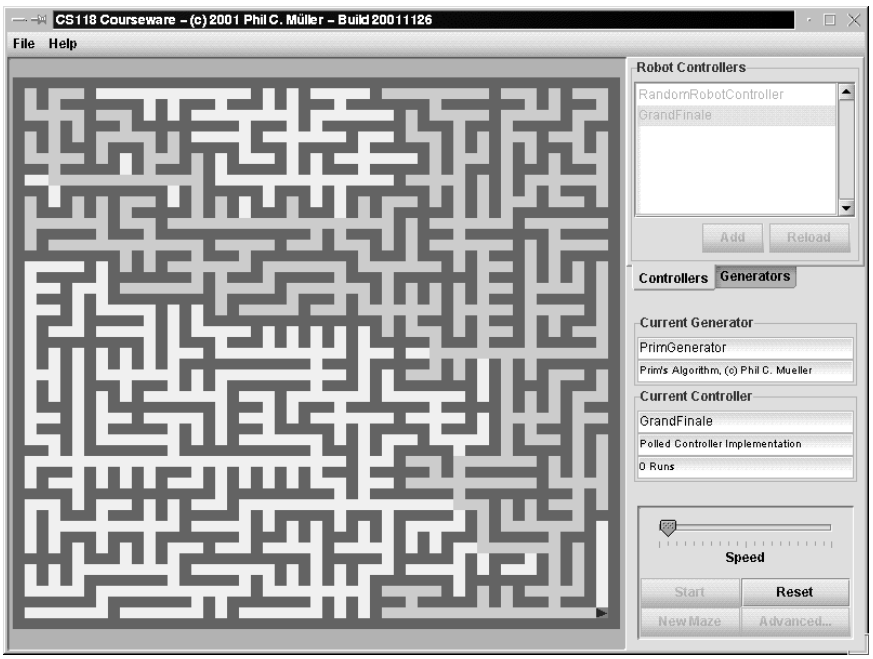


Figure 9.1: A trace of the **GrandFinale** robot the first run through the maze.

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[2]. 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[3]. 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 your previous solution. The robot will search the maze (in its **Explorer**-like way) the *first time* the robot is run through a maze but, the *second time*[4] 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.

[2]The lack of funding has really hit them hard.
[3]Unlike NASA, of course.
[4]or more

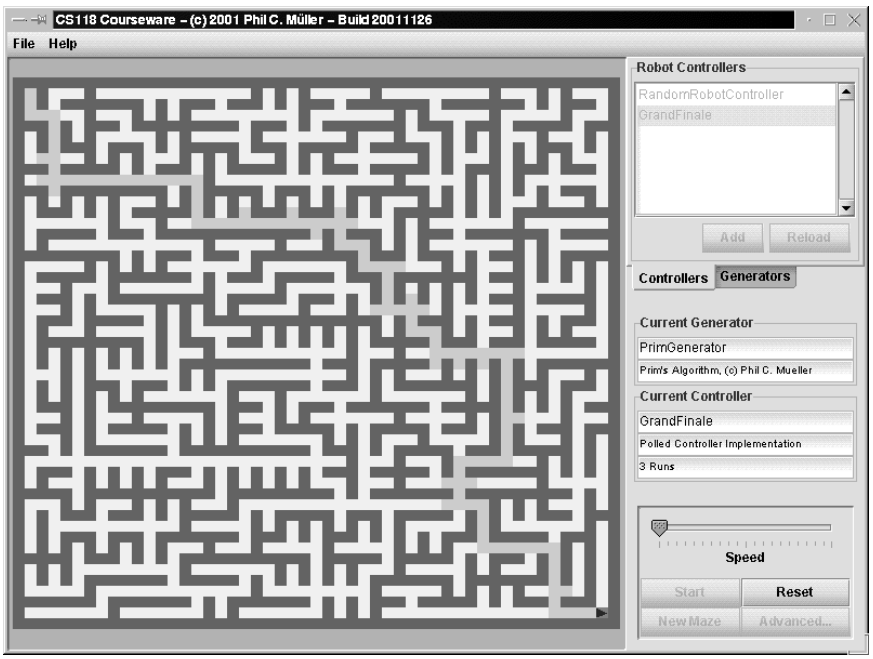


Figure 9.2: A trace of the **GrandFinale** robot the second time it runs through the maze.

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

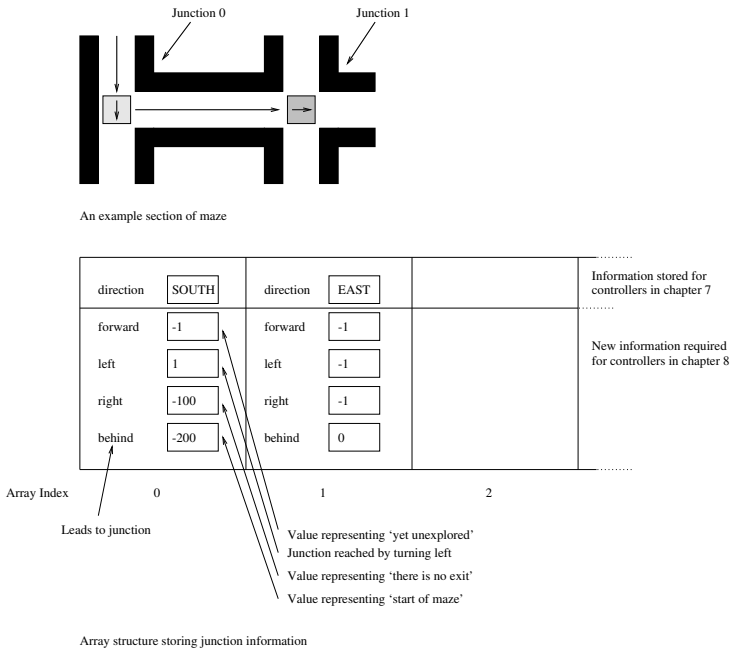


Figure 9.3: Storing more junction information.

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 2 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

Grand Finale Preamble: At the top of your `GrandFinale.java` file you should include a comment that discusses your approach. Did you use one of the suggest approaches? Why? Did you implement your own approach? Why is it better than one of the suggest approaches? Can your Grand Finale solve loopy mazes? Does your robot deal with new mazes? Repeat runs of the same maze? Why or why not? Justify your design decisions.

The files which should be submitted for the second coursework are `Ex1.java`, `Ex2.java`, `Ex3.java` and also `GrandFinale.java`. You may also wish to submit additional files for your 'Robot of the Year' submission.

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 `Ex2.java` contains the definition

```
public class Ex2
```

and similarly for the other files.

Submit your working using Tabula, as before.

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!