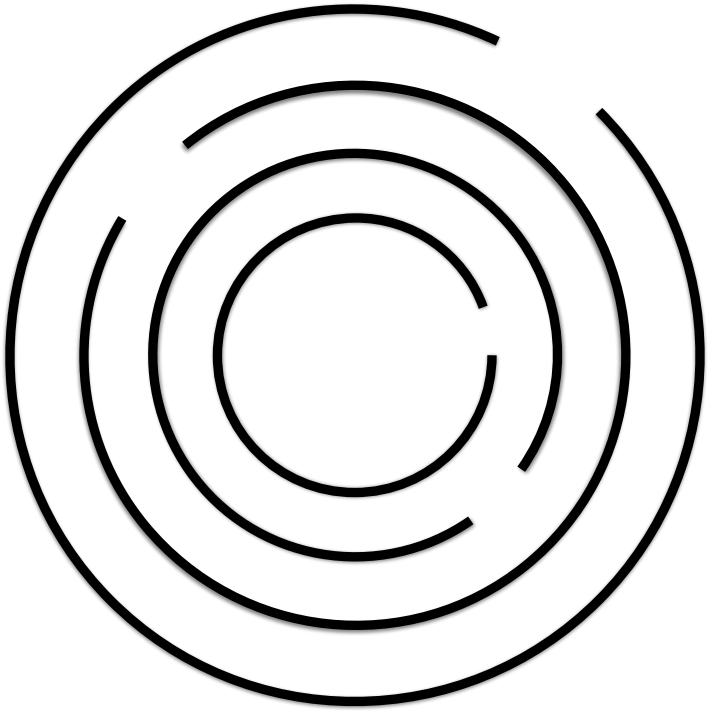
The guide to

Programming for Computer Scientists



Sara Kalvala
sara.kalvala@warwick.ac.uk

Michael B. Gale
m.gale@warwick.ac.uk

2018/19

CONTENTS

Chapter 1

INTRODUCTION

The Programming for Computer Scientists module is designed to give you knowledge and confidence in using a computer as a scientific tool. Whether you already have experience programming or have never programmed before, this module will provide new challenges and leave you with strong, foundational programming skills. In the process, you will gain an equal understanding of fundamental aspects of programming: specification, implementation, and verification. You will get to spend a good deal of time solving software problems, implementing your solutions to these problems on a computer, and running them in order for you to check the resulting behaviour.

The exercises in this module 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 coursework for this module is set in the context of simulating a robot travelling through a virtual maze. This is what you will program; you will be able to see your progress as the robot will either reach the target of the maze or not.

By the end of this module you will have learnt many of the features and techniques needed for computer programming. The first half of the module is structured so that the necessary components of procedural programming are introduced. The concepts covered are applicable to a whole host of different programming languages. Emphasis is placed on writing correct, efficient and maintainable programs. The second half of the module develops on the earlier techniques but with particular application to object oriented programming. The construction of well-designed interfaces, and program encapsulation and abstraction are discussed. The module is based on a number of example programs and emphasis is placed on coursework with the aim of ensuring that the theory covered in lectures is reinforced by practical programming exercises.

1.1 Activities

As part of this module, you will be attending lectures in which you are introduced to programming. This will be done at a relatively quick pace and you will be expected to deepen your theoretical understanding through independent study. Meanwhile, weekly lab sessions will allow you to practice these skills and solidify your understanding with the support of the module organisers and teaching assistants.

1.1.1 Lectures

There are two lectures per week: Mondays at 11am-12pm in L3 and Fridays at 9-10am in R0.21. Lecture slides will be posted to the module website for you to review at any point. You may find that these do not make complete sense outside the lectures since they are only illustrating what we say in the lectures—so don't miss the lectures!

The first lecture will take place on Monday of Week 1 at 11am in L3.

Note that there will be no CS118 lecture on the Friday in Week 1 as you will have an introductory session with our Director of Undergraduate Studies instead at that time.

1.1.2 Labs

You are also required to attend one lab session per week[1]. The lab groups are organised through Tabula and you should check your allocation there to find out when your first lab is.

The labs serve two purposes. Firstly they allow you to interact with the module organisers and the teaching assistants, who are also experts in the subject area, in a direct way. We will be able to guide you through any difficulties you encounter. Make good use of available staff as they are there to assist you and answer any questions which you may have concerning this module. Secondly the labs are designed to allow you to test yourself, to make sure that you are keeping up with the module material. The labs are also good opportunities to get assistance on any aspect of the coursework.

Lab attendance is *not* optional, but the problem sheets for each lab are. Your overall mark for this module will be based on the coursework and the exam.

[1] Don't be confused by the fact that your timetable might have more than one CS118 lab slot marked on it. All this means is that they have printed all the lab times. Which session you will attend will be announced in due course.

1.2 Assessment

Assessment for this module is broken down into two parts. The coursework will account for 40% of you final mark and the remaining 60% of the marks are taken from your two-hour exam which usually takes place in Week 1 of Term 3.

1.2.1 Coursework

There are two pieces of coursework. The first coursework must be submitted via Tabula by 9am on Wednesday 31st October (Week 5) and the second must be submitted via Tabula by 9am on Friday 7th December (Week 10). Your work will be assessed as follows:

Firstly, your programs will be tested automatically to ensure that they work as intended and they will also be checked for plagiarism after you have submitted them. You will then be required to come in to the department where you will have to explain your code to one of the teaching assistants who will also question you about the code.

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

A record of your work will be stored on Tabula, which allows us to record your progress and give you feedback.

1.3 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 an introductory text book. Each year we try to select the most appropriate Java books for this course. In making the selection we 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. We have created a reading list of all the books we would recommend at the moment at:

`https://warwick.rl.talis.com/modules/cs118.html`

This reading list is integrated with the library website, so you can immediately see whether copies of the relevant books are available and where to find them.

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

You may already have a Java book, but you may wish to check that it is up-to-date since programming languages evolve over time. A Java book from ten years ago may not be representative of modern Java. There are enough differences to warrant splashing out and buying a new book if you like having a book. Otherwise, a quick search online can answer most questions.

1.4 Web-based course material

All the module material is available on the web at:

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

We suggest you add this address to your bookmarks. Note that you can also replace `cs118` with the code of any other module to get to the respective module website.

It is strongly recommended that you spend several hours at a terminal during the first few weeks of term getting familiar with Linux, the tools, 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 solving problems, implementing solutions, and verifying their correctness.

1.5 How do I get help?

You will probably find that many of the issues which concern you are also experienced by others on the module. Try talking to your fellow students, teaching assistants, or the module organisers either in person, through Slack, or via email. It is likely that your peers will either have similar problems or will have solutions to these issues. The teaching assistants are usually pretty good at answering questions.

When getting help for the coursework, be mindful not to ask for someone else to do your work for you as this could constitute plagiarism. The teaching assistants and module organisers will be able to guide you in the right direction without directly giving you the answers. You are always welcome to come and find the module organisers in their offices or get in touch via email:

Name	Email
Sara Kalvala	sara.kalvala@warwick.ac.uk
Michael B. Gale	m.gale@warwick.ac.uk

Although we are happy for you to just pop-in whenever we have time, you can find our office hours on the department's website at

`https://warwick.ac.uk/fac/sci/dcs/teaching/officehours/`

These office hours are times that we have specifically set aside in order to meet our students without the need for an appointment.

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! There are plenty of people around who are here to help you. Your personal tutor is usually a great point of contact if you feel that things are getting out of hand.

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 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.

Similarly, this module guide has evolved over the years from Michael Luck's version using Pascal and has benefited from the contributions of many others since then, including Stephen Jarvis, Abhir Bhalerao, and Steven Wright.

Chapter 2

GETTING STARTED

This chapter is intended to be used as part of the first lab session to get you started with some of the tools you will be using throughout this module and your time at Warwick. It is important that you know where you should be each week. For your first lab you should meet in the Atrium of the Computer Science department building. You lab tutor or one of the module organisers will meet you and take you into one of the labs where you will spend the hour working on some tasks. 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.

There are a number of programs we use as part of this module. Some are optional and simply make your life easier, while others are essential. You will need to familiarise yourself with them in order to complete the labs and coursework successfully. This chapter contains an overview of all programs we use with short summaries of what each program does. There are also instructions on how to get started either using the machines in the department or your own. All the machines in the department's labs have everything installed you need to get going.

2.1 Linux

All the computers in the Computer Science building run the Red Hat "flavour" (distribution) of Linux. Other distributions are available for your laptop (e.g. Ubuntu). You will certainly need to pick up some Linux skills while you are at Warwick. One way to learn more is by attending the CS133 Professional Skills lectures.

2.2 Text editors and Integrated Development Environments (IDEs)

Source code written in almost all programming languages is actually just plain text which is formatted according to the *syntax* of a particular programming language. A *compiler* is a program which takes source code and turns it into programs. We will see examples of this shortly, but it means that you can use any text editor you want to write and edit code. Some examples of text editors you may already be familiar with include Nano, Vim, Emcas, Notepad, Notepad++, etc. However, note that programs such as Microsoft Word, Pages, etc. are *not* text editors (they are word processors) and are not suitable for editing source code.

Many text editors have plug-ins which help you write code for a particular language. A very common concept here is *syntax highlighting*, which causes text editors to highlight different parts of your code in different colours to make it easier to read. Since Java is a very commonly used programming language, many text editors already have built-in syntax highlighting for it. However, there are many other features and tools that can assist you, other than just syntax highlighting.

To get started with programming, we would recommend that you try a few different text editors and see which one you like. Michael recommends Atom and Sara recommends Emacs. Both are installed on the lab machines and both have vast ranges of available plugins to support Java programming.

For more sophisticated projects, you may wish to use an Integrated Development Environment (or IDE for short) instead. These are essentially text editors on steroids. Some features that are typically included as a minimum in addition to basic features such as syntax highlighting are:

- Support for project management, allowing you to see all the source files that belong to the program you are currently working on.
- Integration with version control.
- Debugger integration. Debuggers are tools which allow you to troubleshoot your programs while they are running.
- Refactoring tools which allow you to change large parts of your code automatically and consistently.
- Etc.

Some examples of popular IDEs for Java include Eclipse, IntelliJ IDEA, and NetBeans. Some of these are available on the lab machines as well, but while they are more powerful they also have steeper learning curves.

2.3 Java

Java is an *imperative* and *object-oriented* programming language which is used at many universities to introduce students to programming. It is also used widely in the software industry at many big companies. You can find general information about Java as well as links to a wealth of resources at:

`http://www.oracle.com/technetwork/java/index.html`

As with most other programming languages, Java source files are just plain text files. They are conventionally given the `.java` file extension. In order to compile a Java source file, you need a Java *compiler* as well as a Java *runtime*. On the DCS machines, both are already installed but you may find this information useful regardless to gain a better understanding of what these components are and also how you can acquire them on your own machine.

Java Development Kit (JDK) This is a collection of tools which enable you to *compile* Java code. Since source code is just plain text, you need a *compiler*: a program which turns the code you write into programs that can be run on a computer. As software developers, we require these tools. Oracle's JDK is available from:

`https://www.oracle.com/technetwork/java/javase/downloads/index.html`

We will discuss how to use the tools that are included in the JDK in detail in Section 2.3.1.

Java Runtime Environment (JRE) Java source code is compiled to Java *bytecode* by the Java compiler. This bytecode then needs to be run on a Java Virtual Machine (JVM). The Java Runtime Environment contains a JVM. As with the JDK, a JRE is already installed on the lab machines, but if you need to install one on our own machine you can find downloads at:

`http://www.oracle.com/technetwork/java/javase/downloads/`
`jre8-downloads-2133155.html`

2.3.1 Editing, compiling, and running Java code

Ex1 In a text editor of your choice, create a new file which contains the following text:

```java
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.

Ex2 Now compile the program. To do this you need to open a terminal window, navigate to the folder where you saved `Hello.java` and then run the Java compiler that is part of the JDK (we use $ throughout this guide to denote a terminal prompt):

```
$ javac Hello.java
```

If there is no output to the terminal, then the program has been compiled successfully and a file named `Hello.class` has been generated in the current directory. If the Java compiler reports any errors at this point, however, then you may have typed the program incorrectly. Correct any errors in the program by switching back to the text editor, making corrections, and then running the command above again until there is no output and `Hello.class` is generated.

Ex3 You can now run the program by typing the following which invokes the Java Virtual Machine:

```
$ java Hello
```

The program should display `Hello World!` in the terminal. Congratulations! You have just learnt how to edit, compile, and run a Java program.

Ex4 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.

Ex5 Create a new file called `Age.java`. Type the following program exactly as it is written:

```java
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 the `.java` file extension and the filename must match the name which usually follows the `public class` part in the contents of the file (including matching capitalisation!).

You may also notice that the formatting of the lines in the example does not really matter and statements can be spread out over several lines. Java uses semicolons to understand where statements end and curly braces to denote where a block of code starts and ends.

Ex6 The meaning of the individual components of the program above will become clearer as you learn more about programming in Java. However, try to experiment with the different parts of the program to see if you can work out what each of them does.

2.4 Version Control with Git

Version control systems help you keep track of changes you make to your code. This allows you to revert your code back to an earlier version if you make a serious mistake or don't like the changes you have made. As you will find out in the second year of your course, it is also very useful if you are working with other people. There many different version control systems, but the one we would recommend is called Git.

The skeleton code for all practicals and for all coursework is available on GitHub at `https://github.com/dcs-cs118/` and you are encouraged to use version control to obtain and maintain the code. If you have not had much exposure to version control using Git before, *Pro Git* by Scott Chacon and Ben Straub is a very good reference book

which is available for free at `https://git-scm.com/book/en/v2`. Below is a very quick reference of some of the most important commands you will use.

To obtain the code for *e.g.* the first practical which is located in the `lab1` repository, you will want to run the following command on your machine once you have installed `git`:

```
$ git clone https://github.com/dcs-cs118/lab1
```

This will create a local copy of the `lab1` repository on your machine that you can modify. If you are planning to work on your practical or coursework from multiple locations (e.g. a machine in the lab and your personal machine), you may find it beneficial to create a GitHub account and *fork* the relevant repositories to your account instead. This creates a copy of them on your GitHub account which can then be read from and written to from anywhere. You can fork repositories on the GitHub website by visiting e.g. `https://github.com/dcs-cs118/lab1` once logged in and clicking "fork". If you take this approach, you will still need to obtain a local copy of the repository on all machines you plan to work on by running the command shown above, but replacing `dcs-cs118` with your GitHub username.

WARNING: do not fork the coursework repository to your account as they will end up being public and you do not want everyone to be able to see your solutions! Use the links provided on the module website instead which will allow you to create private copies of the repositories.

Once you have made some changes to the skeleton code, you will want to *commit* your changes. This will tell `git` to remember that version in case you ever wish to go back to it. You can commit your changes by running:

```
$ git commit -m 'Some message to describe the changes' -a
```

If you forked the repository to your own GitHub account, you may now wish to update that repository with your local changes by running:

```
$ git push
```

This will update the repository on GitHub with all changes you have made. You may then wish to run the following command if you continue working on another machine, which will update the local repository there with changes from GitHub:

```
$ git pull
```

None of this is necessary if you have cloned the code from `https://github.com/dcs-cs118/` directly, but then you also cannot work on the code from multiple machines, unless you store the local repository on e.g. Dropbox or a similar service.

2.5 Gradle

Java projects can quickly grow very large and may consist of hundreds of files, dependencies to code that other people have written, and many other components. It would be tedious if we had to write `javac File1.java File2.java File3.java ...` all the time to compile our program.

Gradle is a *build* tool (one of many) which simplifies this process. We will use it for the coursework and in later labs. Instead of invoking the Java compiler directly in a terminal, we can create a file named `build.gradle` which stores a configuration for how your program should be compiled. This configuration will be given to you so you just need to run one of the following commands then (*e.g.* in the `lab1` directory that was obtained by following the instructions for `git clone` in Section 2.4):

```
$ gradle build
```

This command will compile your program based on the configuration in `build.gradle`. If you want to compile and run your program, you can use the following command:

```
$ gradle run
```

Finally, some of the skeleton code we give you may include *test suites*. We talk more about testing in Chapter 5, but you can run these test suites by executing the following command:

```
$ gradle test
```

This will run all tests and generate a report of which tests succeeded and which failed. Do not be afraid of compiler errors or failing tests! Both are there to help you and nobody writes flawless code at the first attempt.

2.6 What next?

If you have gone through the exercises above and have made sure to sign in with one of the teaching assistants 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 lab is to look at the next chapters of this guide. If you decide not to do 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 Specification

Designing a computer program can be a complicated business. It often 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.

In a professional setting, 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 the creating of a rota of workers, for example, and that they need a program to allocate workers and email them with their schedule. 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

> 'Allocate time slots to workers and email them their schedules'

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 allocates all hours to one worker, for example; this after all meets the specification, though probably does not work very well for the customer.

Alternatively, a specification which states

> 'Line 10: Assign to the worker with fewest hours the next time slot in the rota, unless it is an early morning slot in which case find a worker who has a car…'

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. Developing an appropriate program specification is an important skill for good programming.

3.1 Writing your own specifications

Most specifications are written either in *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 closer we move towards maths (or logic) the less chance there is of introducing any ambiguities. You may learn more about using what are known as *formal methods* later in your degree course. For now, we will rely on attempting to write specifications as clearly and as easy to read as possible. Programs and their specifications need to be understandable by other people.

3.2 Building computer programs

Building a computer program is the task traditionally described as *programming*. Despite many misconceptions, programming is rarely about sitting at a desk littered with cans of Red Bull and bashing out some obscure lines of instructions which reflect the programmer's thoughts and insights on a particular problem. Programming is an exact and detailed science which involves methodically 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*?

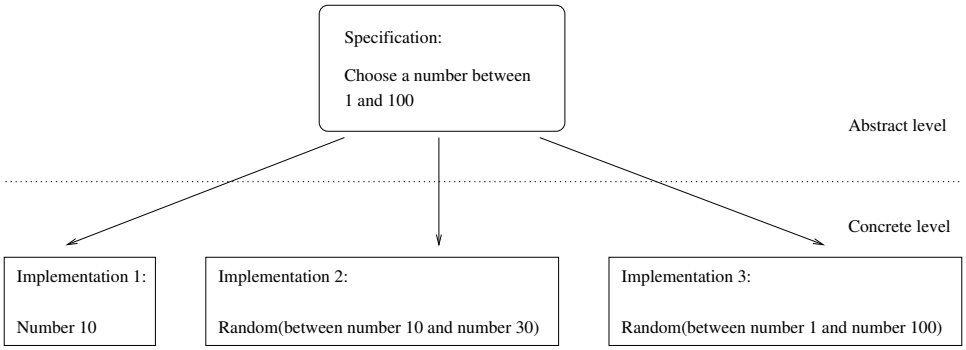


Figure 3.1: Example of abstract- and concrete-level design

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 3.1 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 requesting a software system to write a collection of unambiguous requirements. The customers 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.

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

The translation between an abstract-level specification and a concrete-level design is technically called a *refinement*. Each of the programs in Figure 3.1 is a valid refinement of 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*.

Chapter 4

Implementing Programs in Java

This chapter is a short introduction to Java that you can use to get started with some of the basic principles of the language. We outline the basic building blocks of Java programs and intersperse them with some short exercises. If you have a question related to Java, you could quickly consult this chapter before asking for help from elsewhere (Section 1.5).

Before we cover 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). 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 method named `main`. Note that file and the class must have the same name (*e.g.* `MyClass` in the example below). The `main` method you write should have a specific 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) {
        // your code here
    }
}
```

After writing some statements in your `main` method, 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. Variables must be declared prior to their use and are given a name, and a *type*. In this module, we cover three categories of types: *primitive* types, *objects*, and *arrays*.

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

Type	Description
`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`.

To declare a variable, we need to choose an appropriate type, a name, and optionally an initial value. The variable name can be any string of alphanumeric characters, but cannot start with a numeral character. Furthermore, variables can contain the underscore character (_) and a dollar sign ($).

```
// a variable named x with no intial value
int age;

// a variable named y with an initial value of 42
int answer = 42;
```

```
// a variable that stores the character 'a'
char c = 'a';

// a variable that stores a floating point number
float myExamAverage = 89.5;

// a variable that stores a boolean value
bool result = false;
```

Keywords, such as `int`, `void`, etc. cannot be used as names for variables since it would be incredibly confusing to the Java compiler, as well as a programmer.

Before a variable's value can be accessed, it must be given one. The value of a variable can be assigned or changed with the *assignment operator*, `=`. Unlike in the world of mathematics, in imperative programming languages the value on the right is assigned to the variable named on the left. For example assigning $(1 + 2) \times 3$ to `x` would be written in Java as:

```
// initialise a variable with the result of (1+2)*3
int x = (1 + 2) * 3;

// change the value of x to the result of 1+2*3
x = 1 + 2 * 3;
```

Note that when assigning a new value to an existing variable, we do not have to specify its type since the Java compiler already knows what the type of `x` is. If we did specify the type again, the Java compiler would treat it as an attempt to declare a new variable with the same name as an existing one and we would receive an error.

Ex7 Think about what each type above might be used for. What type of variable would you use to store the radius of the earth, which is approximately 6,371 km.

Ex8 What type of variable would you use to store the number of people in the world? There are approximately 7,046,000,000 people.

Ex9 Try writing and running a Java program to calculate the population density of the earth. The answer should be roughly 13.82 people per km^2.

4.1.1 Objects

There is an in-depth discussion of what *objects* are later in this guide (Section 4.6), but for now you can just think of objects as variables which we can interact with by calling *methods* on them. Variables of type `String` are objects you will come across early on. As convention, the names of types which represent objects usually start with an upper-case character while the names of primitive types start with a lower-case character. Consider the following example:

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

Here, `s` is a variable whose type is `String`. Therefore, `s` is an object and we can call methods such as `toLowerCase` and `length` on it. The methods that can be called on a particular type of object vary from type to type. The Java documentation is a great source to find what methods can be called on all of Java's in-built object types. Later in the course you will be creating your own types of objects by defining *classes*.

Ex10 Find the documentation for the `String` type using your favourite search engine and find out how to find which character is at position n.

4.1.2 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:

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

In this example, we create an array of three `int` values named `myArray`. The `new` keyword is used to initialise arrays and objects. When initialising an array, we must

specify the number of elements the array will store. Array indices in nearly all programming languages begin at 0 and therefore the last item can be found at the location $n-1$, where n is the size of the array. In Java, the length of an array can be determined by reading the `length` *field* of the array:

```
System.out.println(
        "The array has " +
        myArray.length +
        " elements");
```

4.1.3 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 class variables maintain their values between different methods:

```
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 only if that condition evaluates to `true`. To make an application execute a different block of code if the condition is `false`, an `else` statement can be attached to the end of the `if`-statement as shown below:

```
if (condition) {
   // this gets executed if condition is true
} else {
   // this gets executed if condition is false
}
```

Furthermore, you can combine the `if` and `else` statements to check an additional condition. Although one is shown below, you can have as many `else if` statements as you like:

```
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 be any boolean expression that you like. Therefore, we can use *e.g.* boolean operators such as `&&` for logical AND and `||` for logical OR. When using complicated conditions in if-statements, be sure to bracket carefully to ensure clarity:

```
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 but is used to check equality of one value to many others. The basic format of a `switch` statement is:

```
switch (expression) {
case 1:
    // some code here for when expression is 1
    break;
case 2:
    // some code here for when expression is 2
default:
    // some code here for when expression is neither
}
```

With a statement of this kind, Java will look at the value of the `expression`, check if there is a specific case statement for its value from top to bottom, and begin executing instructions from this point onwards. If no cases are matched, the `default` case will be executed, unless it is absent. If Java encounters a `break` statement, it will skip to the end of the switch statement and continue with the program from there. 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. In the example above, if `expression` evaluates to 2, the code for the case for 2 and for the `default` case will be executed.

Ex11 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");
}
```

Ex12 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 Loops

While coding Java applications you may find that you end up repeating yourself. Repeated code can be encapsulated into loops to make Java execute the same code multiple times. Consider the following code which contains a lot of repetition:

```
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 a loop 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 module we will deal with `while` loops and `for` loops, as well as some variants on these.

4.3.1 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) {
    // body of the loop that will be executed as long as
    // condition is true
}
```

To encode the previous code into a `while` loop we could write 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 be created in which the program will continually execute the middle block without ever terminating. 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 {
  // body of the loop that will get executed
  // until condition is false
} 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 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. Note that this isn't an efficient way to accomplish that.

4.3.2 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, we often use a `for` loop for this instead since it is more convenient. A `for` loop is written using three components: an initial condition, an terminating condition and an iterative operation. For example,

```
for (initial; condition; iteration) {
   // body of the for loop
}
```

Rewriting the `while` loop from 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 i = 1; a < 4; a = a + 1) {
    System.out.println("a is: " + a);
}
```

Ex13 Rewrite the following `for` loop as a `while` loop:

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

Ex14 Given the following code, copy the characters out of the `String` named `s` and into the `char` array named `c`. Use the Java documentation for the `String` type to find out how to retrieve individual characters from `s`. Try using both a `while` and a `for` loop.

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

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 are static fields of the `System` class accessible 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 documentation for the `PrintStream` type. You should find a method called `print` that works like `println`, but doesn't put a line break at the end. 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.

Ex15 Using your knowledge of loops and the documentation for `Scanner`, use `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. When accessing members of objects, you differentiate methods from fields by the use of the following brackets,

`()`, which surround any arguments that a method is called with. When breaking down problems into sub-problems, 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 greatly. 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 which is called `void`. You've already seen a `void` method in Java, as the `main` method does not return any value to the operating system. Further to this, methods may also be given 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:

```
/*modifier*/ /*type*/ methodName(/*type*/ paramName, ...) {
   // body of the method
}
```

Where `/*modifier*/` is either `public`, `private`, `protected`, or nothing. If no access modifier is specified, Java defaults to `private`. `/*type*/` is a type and is required. Thus, a basic addition method 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`, which must also be of type `int`. 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 parameters to the method with a method call like so (assuming that `add` is called from within the same class in which it is defined):

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

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

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

Ex16 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 documentation for the `Math` class 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:

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

This is creating an object named `sc`, and it is initialising it to be a new instance of the `Scanner` class. Furthermore, it is calling a special method that exists in every class called a *constructor*. Constructors 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 other methods to perform set-up actions. 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 the program's main method and is therefore where execution of the program begins. It creates an object named `mc` and initialises it using `MyClass2`'s constructor, which sets the instance variable `a` to an integer value that is given to `MyClass2`'s constructor as argument. As there are two variables named `a` (one is a class variable and the other is a parameter of the constructor), the `this` keyword is used to disambiguate between the class variable `a` and the method variable `a`; `this.a` refers to the class variable, whereas just `a` refers to the parameter passed to the constructor method. The general rule is that parameters and local variables "shadow" class variables.

After creating the `mc` object, the method `getA` is called on it to get the value of `a` and return it so that it can be printed with the `println` method.

4.6.1 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 a feature of Java called *access modifiers* 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 class variables should be private and can therefore only be changed by method in the class. The reasoning for this is simple: if only methods of a class can modify the values of fields, then it is much easier to understand the program and identify potential sources of errors. Consider the following example of a lift:

```
public class Lift {
    private int floor = 0;
    private int maxFloor;

    public Lift(int maxFloor) {
        this.maxFloor = maxFloor;
    }

    public void moveUp() {
       this.floor = Math.min(this.floor + 1, this.maxFloor);
    }

    public void moveDown() {
       this.floor = Math.max(this.floor - 1, 0);
    }
```

```
    public int getCurrentFloor() {
       return this.floor;
    }
}
```

As we can see, the fields of the `Lift` class are `private` and all the methods, including the constructor, are `public`. This means that we have easy-to-follow behaviour because, once a `Lift` object has been initialised with a value indicating how many floors there are, it can only be moved up or down one floor at a time within the range of floors that exists:

```
Lift l = new Lift(4);
l.moveUp();
l.moveUp();
l.moveDown();
System.out.println("Lift is on floor: " + l.getCurrentFloor());
```

Let us now make the fields `public`:

```
public class Lift {
    public int floor = 0;
    public int maxFloor;

    // same code as before for the methods
}
```

We can now do anything we want with `Lift` objects, regardless of whether it makes sense or not:

```
// initialise a lift for a building with 4 floors
Lift l = new Lift(4);
l.floor = 108;
System.out.println("Lift is on floor: " + l.floor);
```

Even though we initialised the lift to have only four floors, we are able to set the floor the lift is on to 108 and have thus violated the logic we were trying to implement. This demonstrates that by using access modifiers, we can stop ourselves from making stupid mistakes.

However, this is not to say that all your variables should always be private and all your methods should always be public. In fact, the sanest 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).

4.6.2 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 build an object hierarchy. This may seem strange but consider *classifying* the animals on campus. Many of them share properties and abilities but some also require different properties. Let's create a class for animals:

```
public class Animal {
    private int cuteness;

    public Animal(int cuteness) {
        this.cuteness = cuteness;
    }

    public void die() {
        this.cuteness = -this.cuteness;
    }
}
```

That is animals are born cute and are less cute when they die. Now think about your squirrels and ducks. They're both adorable forms of animals, but they have different properties, thus we can *extend* the `Animal` class and inherit the methods and properties that all animals share, and add our new methods and properties:

```
public class Squirrel extends Animal {
    private int nutsStoredInCheeks;

    public Squirrel() {
        super(100);
        this.nutsStoredInCheeks = 2;
    }

    public void climbTree() {
       // ...
    }
}
```

```
public class Duck extends Animal {
    private int timesQuackedToday = 0;

    public Duck() {
        super(9001);
    }

    public void quack() {
        this.timesQuackedToday++;
    }
}
```

Here we have defined two classes, `Squirrel` and `Duck`, which extend `Animal`. By doing so, they *inherit* all of `Animal`'s methods and fields. For example, we could write the following program:

```
Duck vader = new Duck();
vader.quack();
vader.die();
```

Since the `die` method is inherited from the `Animal` class, we can call it on objects of type `Duck`. In the code for the constructor of the `Duck` class, we use the `super` keyword to call the parent's constructor: setting the cuteness of a duck to 9001. Therefore, after calling `die` on `vader`, its cuteness will be -9001.

We can even extend the extended classes. For example, we could define classes which extend `Duck` for different species of ducks. Indeed, Java has an extensive range of object-oriented features and we cannot address all of them in this short guide. The lectures cover more topics related to object-oriented programming in Java. If you want to read more instead, you should consult one of the recommended textbooks.

Ex17 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.

Chapter 5

An Introduction to Testing

Testing a computer program is an extremely important business. There are many well-known examples where software has failed due to inadequate testing. One such example is the case of the Ariane 5 launch vehicle. 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! It can be argued that more extensive and methodical testing could have avoided this accident.

Of course not every example of software failure will end in a disaster quite as catastrophic as this. However, the consequences of code failure 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.

Failure is defined as 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 are reduced, the more you will feel that the program exhibits the correct behaviour.

5.1 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 module.

Test of logical paths of program One useful way to test a program is to examine all the logical paths. Consider this example:

```
while (x < 10) {
  if (even(x)) {
    System.out.println("The number is even");
  } else {
    System.out.println("The number is odd");
    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 would 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 destroys life on earth. 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 be 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, in the case of an airplane 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 to make sure that one specific ordering 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'.

To generate pseudo-random numbers in Java, we can use the static `random` method of the `Math` class. This method returns a pseudo-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 the static `round` method of the `Math` class), and then cast the result to an `int` value:

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

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

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

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 that stores and deletes a collection of names will often fail 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, as 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 This consists of software that runs alongside the program to be tested. The test application, also known as a test rig, 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 overnight, 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... There are software tools that allow test rigs to be generated semi-automatically.

You will be encouraged to try some of these test methods during your software development assignments, and to explain how you have tested your software, 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.

5.2 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`)!

Ex18 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:

```java
public class Test {
  public static void main(String[] args) {
    int a[] = new int[2];
    a[2] = 3;
  }
}
```

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` method in `Test.java` and is on line 4. The exception that was generated was an `ArrayIndexOutOfBoundsException` (which you can look up in the Java documentation) 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.

Ex19 Try writing example applications to perform very simple tasks. While doing this, you will undoubtedly 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 6

Introduction to the Maze

The coursework exercises for the CS118 module are 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) or, more generally, the *agent* is to find the given target as rapidly and efficiently as possible, learn 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.

6.1 The Robot Maze environment

To get hold of the maze environment, you can clone the code from GitHub by running the following command in a terminal. A new folder named `cswk` will be created in your current working directory, so you may want to navigate to *e.g.* your home directory (with `cd ~`) or a folder you created for CS118 first:

```
$ git clone https://github.com/dcs-cs118/cswk
```

Alternatively, you can also download the skeleton code from the module website. Once you have obtained the skeleton code through either means, you may wish to verify that the code compiles by entering the `cswk` directory and running `gradle run`:

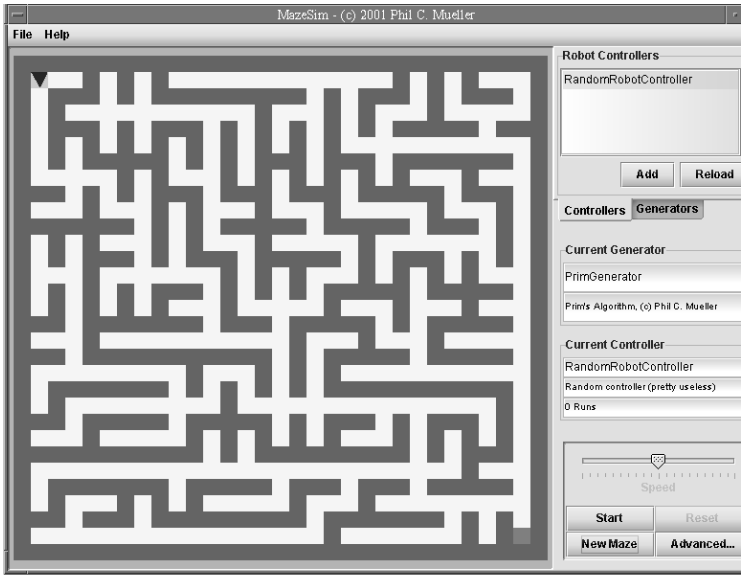


Figure 6.1: The Robot-maze environment

```
$ cd cswk
$ gradle run
```

This should open the maze environment with a randomly generated maze as shown in Figure 6.1. The simulated Robot Maze environment has the following characteristics:

- The robot moves through a simple square-block maze of the type illustrated in Figure 6.1. 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 direction 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 `start` – which is run at the start of a simulation.
- A simulation *usually* starts with the robot at the top left-hand square of the maze and ends when either the robot reaches the target or the user loses patience and stops the robot (by pressing **Reset** in the user interface). 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 squares the robot has been at before (which are otherwise empty squares). 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 traversing the given maze.

6.2 Programming robot control programs

The simulated robot maze environment is written in Java. The programs which you are required to write for this module 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. This interface is documented in the following sections. However, because you are only required to write control software known as *controllers* for the maze environment and while every part of the maze software is accessible for you to use, the implementation of the maze environment itself is hidden from you. This is an example of abstraction. Everything you need to know to use the maze environment is documented in this guide.

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 with your peers first and then ask for clarification from the teaching team. Understanding *program interfaces* like this is very important in software development, particularly if you are to use it to write your own program code.

6.2.1 Specifying headings in the maze

Four pre-defined constants are defined in the code for the robot maze program to specify directions in the maze. These are

`IRobot.NORTH`, `IRobot.EAST`, `IRobot.SOUTH`, `IRobot.WEST`

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

As the values are defined as part of a Java interface named `IRobot`, the constants are prefixed with the name of the interface as shown above when they are used in the actual program code that you write. This might seem a bit quirky but you will soon get used to it.

These elements of the interface are concretely represented as Java values of type `int`. The values represented by these directional constants are chosen so that the following equalities hold:

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

Ex20 Note that `IRobot.WEST+1 != IRobot.NORTH`. Could you find a way to represent the directions in Java so that this would be the case? 除以4

6.2.2 Specifying directions relative to the robot heading

Four pre-defined constants are defined in the code for the robot maze program to specify directions relative to the robot's current heading. These are:

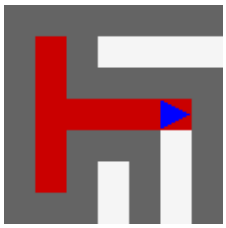
`IRobot.LEFT`, `IRobot.RIGHT`, `IRobot.AHEAD`, `IRobot.BEHIND`

As with headings these are also of the type `int`. The values represented by these directional constants are chosen so that the following equalities hold:

```
 IRobot.AHEAD+1  ==  IRobot.RIGHT
 IRobot.RIGHT+1  ==  IRobot.BEHIND
IRobot.BEHIND+1  ==  IRobot.LEFT
```

A fifth constant `IRobot.CENTRE` is also defined, which can be useful as a "do nothing" 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. `IRobot.RIGHT`, `IRobot.NORTH`, etc.) and not the constant *value* used to represent it. This is our first encounter with *program abstraction*.



Method 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.PASSAGE`

Figure 6.2: Example of sensing robot surroundings

6.2.3 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`	An empty square that has not yet been visited on the current traversal of the maze.
`IRobot.WALL`	A wall or the edge of the maze.
`IRobot.BEENBEFORE`	An empty square that has already been visited during the current traversal of the maze.

Figure 6.2 shows a typical situation that might arise during a traversal of the maze. The robot is located in the square with an arrow, 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 results shown in the table.

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`.

6.2.4 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 6.2 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`).

6.2.5 Sensing the location of the robot and target

The method `robot.getLocation()` returns an object of type `Point` which has two public variable members. You can get the x and y coordinates of the robot in the maze

from the `Point` class by accessing the `x` and `y` members of the object:

```
Point p = robot.getLocation();
System.out.println("The robot is at " + p.x + " and " + p.y);
```

Did you know that you could also write the following instead:

```
System.out.println("The robot is at " +
    robot.getLocation().x + " and " +
    robot.getLocation().y);
```

The top left square in the maze is represented by the coordinates $(1, 1)$. Similarly, the method `robot.getTargetLocation()` can be used to return the x and y coordinates of the robot's target.

6.2.6 Turning the robot

The method `robot.face(direction)` makes the robot turn in the specified `direction` (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()`.

6.2.7 Moving the robot

Your code should point the robot in a suitable direction. Afterwards, you should instruct the robot to advance into the specified direction. The `robot.advance()` method will cause the robot to move by one square into the direction that it is facing. For example, the following code would cause the robot to turn right and advance into the resulting direction by one square:

```
robot.face(IRobot.RIGHT);
robot.advance();
```

6.2.8 Detecting the start of a traversal and a change of maze

The method `robot.getRuns()` returns a value of type `int` which corresponds to the the number of previous traversals which the robot has made of a given maze. After you have guided a robot through a maze, you will notice that the user interface of the robot maze environment displays `1 Run` on the right. This is the result of the `robot.getRuns()` method. You will find that this method is useful in the second coursework.

Chapter 7

THE MAZE I

This chapter contains the specification for the first coursework, which is about preventing robots from running into walls in the maze and leading them towards the target in a more efficient manner.

Your solutions are due at *9am on Wednesday 31st October (Week 5)*. On the same day, you will be asked to come in to the labs where you will demonstrate that your code works and that you have understood the material for each of the exercises. If we are not able to mark your work on that day, for whatever reason, then you will receive no marks[1].

7.1 Getting started

By this point you should have read Chapter 6 and familiarised yourself with the Robot Maze environment. Your starting point for the coursework is the code that you have obtained from either GitHub or the module website in Section 6.1.

7.2 Task 1.1

An order is received from an existing customer of the Robot Maze software for a modified robot controller:

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

[1] If you have serious mitigating circumstances that are outside of your control, such as severe illness or a family emergency, please speak to your personal tutor about this as early as possible.

This description can be thought of as the *requirements specification* 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 *Specification*, *Implementation*, and *Evaluation* which were described in the previous chapters in this guide.

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 chooses 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 `start` method in the `RandomController.java` file can be modified to achieve this behaviour. 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 modifying the code found in `RandomController.java`. The only difference between the existing controller and the new one should be that the modified controller will prevent the robot from crashing into walls.

Design Question: How do you think you can detect if the robot is about to crash into a wall? *Hint*: the answer is in Chapter 6 of this 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: How do you plan to do this? *Hint*: look at the lecture slides, this guide, and any Java books you have to find out more about loops.

Task 1.1.1 Once you have designed the program you are ready to *implement* your changes to the robot controller. 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 to feel daunted by the programming task ahead!

After saving the updated `RandomController.java` file, you should compile the code using Gradle. Remember that you need to run this command in a Terminal and that you need to have navigated to your project directory:

```
$ gradle build
```

If you receive no errors from running this command, then the new program has been compiled successfully. If you do receive errors, you should fix them before continuing.

For this first exercise, we have also provided a *unit test* for you which can automatically test whether your updated controller does not run into any walls. The code for this test can be found in `src/test/RandomControllerTest.java`. You may wish to inspect the code at this point. You can run the unit test by executing the following command in your terminal window:

```
$ gradle test
```

This will run the unit test, generate a random maze, and then use your controller to find the target in it. Finally, the test will check that the robot has not run into any walls during the simulation. Once the test has finished running, you can find a report in the `build/reports/tests/test` folder relative to the project's root directory. There will be a file named `index.html` which you can open in a web browser of your choice.

It is recommended that, after completing every task, you make a new commit to your Git repository by running *e.g.*:

```
$ git commit -m 'Stops the robot from running into walls' -a
```

This will tell `git` to record the changes you have made since the last commit and will later allow you to revert to this snapshot of your code in case something goes horribly wrong. The `-m 'Stops the robot from running into the walls'` gives the commit a description that will help you understand what it was for. The `-a` flag tells `git` to include all updated files in the commit.

7.3 Task 1.2

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 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.

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 `RandomController.java`.

7.3.1 Keeping track of moves

The Robot Maze software provides a way of logging the directions in which the robot moves. For example, you could write the following to log a move forward:

```
robot.getLogger().log(IRobot.AHEAD);
```

Calling this method would record a single move forward. It will also result in some output to the terminal:

```
Summary of moves: Forward=1 Left=0 Right=0 Backwards=0
```

Task 1.2.1 You should now modify `RandomController.java` so that the direction of every move is logged. Once you have implemented this, you may wish to run a simulation to see if it works. For every move the robot makes, one of the four counters should go up by one. 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 you are logging the directions incorrectly.

7.3.2 Addressing the customer's complaint

Using the logging from Section 7.3.1, 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 and how often it goes right, compared to how often the robot chooses to go forward and how often it goes backwards? You may find it helpful to run the controller on different mazes. Perhaps even a blank maze?

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 `RandomController` are indeed random, but that the directions are not chosen with the same probability.

Investigating the bias in choice of directions would have been difficult by hand. However, with the additional logging code we can analyse the frequency by which the robot moves in each direction.

Investigate the definitions of the `Math.random` and `Math.round` methods in the Java documentation. Using this information note the probability of the robot choosing the directions *left*, *right*, *ahead* and *behind* in a comment in `RandomController.java`.

The Java documentation is 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.

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."

Task 1.2.2 You should base your solution to this exercise on the previous exercise. So make sure that you continue working on the `RandomController.java` file. You should also remember to keep regular backups at this point. *Hint*: there are many ways to accomplish this task; think carefully about the use of the `Math.random` and `Math.round` methods.

7.4 Task 1.3

Use the same method of *specification*, *implementation*, and *verification* in order to further modify `RandomController` 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 `RandomController` 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."

Task 1.3.1 Your solution should be building upon the work you have done previously, so again modify the code in the file `RandomController.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?

7.4.1 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 `RandomController.java`. This time, as well as displaying all the previous characteristics, the robot controller should also choose a new direction randomly, on *average* every 1 in 8 moves, irrespective of whether there is a wall ahead or not.

There are two design questions which need to be considered:

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

Task 1.3.2 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 `RandomController` be expected to reach the target if given enough time? Test your solution on some suitable examples.

7.5 Task 1.4

The main aim of the remaining exercises 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.

If you have not done so yet, you should commit your changes with *e.g.*:

```
$ git commit -m 'Finished part 1 of the first coursework!' -a
```

There is some new skeleton code for the following exercises. In order to "make it appear", run the following command (if you have downloaded the `.zip` file from the module website):

```
$ git merge cswk1part2
```

If you have cloned the code from *e.g.* GitHub, use the following command instead:

```
$ git merge origin/cswk1part2
```

A new file named `HomingController.java` will appear in the `src/main/` subfolder. You will be working with this file instead of `RandomController.java` from now on.

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 `HomingController.java`. This robot controller has two programming errors

that the compiler cannot detect. As before, you can compile and run the code using `gradle run` in your terminal.

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 following line:

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

If you study the details of the `robot.look` method in Section 6.2.3 of this guide, then you will find that it returns a result which is `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, we know that the method `robot.look` is expecting something like `IRobot.AHEAD`, `IRobot.LEFT`, etc. and is instead passed something like `IRobot.NORTH`, `IRobot.SOUTH`, etc. But the Java compiler is oblivious to the problem. As far as the compiler is concerned, all of these things look the same. Both are represented in the program as values of type `int`, and so any type-checking that the compiler performs will just ensure that the value assigned to the variable `direction` and passed to the `robot.look` method is of type `int` – which it is. Unfortunately, this means that the Java compiler cannot detect such mistakes for us. 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). We will later learn how we can address such problems somewhat by constructing our programs in just the right ways[2].

Task 1.4.1 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 in Section 7.2.

7.6 Task 1.5

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 head 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.

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[3]. 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.

[2] If you choose to take CS141 Functional Programming in Term 2, we will see a more expressive type system which allows us to represent semantic information in types to catch many different sorts of errors.

[3] The coordinates begin with (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).

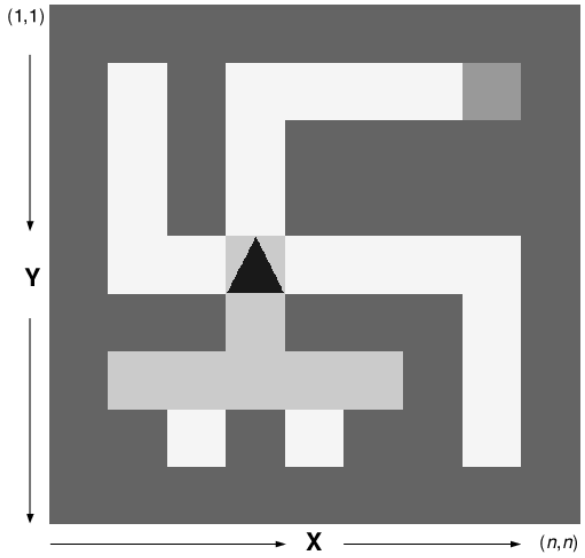


Figure 7.1: The robot homing towards a target that is north-east would choose to go ahead (north) or right (east) as opposed to behind (south) or left (west). Note the relationship between the x- and y-coordinates of the robot and the target.

7.6.1 Is the target north?

Task 1.5.1 Implement the `isTargetNorth` method in the `HomingController.java` file. A "stub" of this method is already given to you which always returns 0. However, the method should actually return 1 if the target is north of the robot, -1 if the target is south of the robot, and 0 otherwise. Implement this now. The method should not be longer than about five lines long.

You may wish to first sketch your solution on paper, answering the following design questions as you go along:

Design Question 1: How can you determine the relative positions of the robot and the target?

Design Question 2: 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 Chapter 5 of this guide to see what testing technique would be 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 during the marking day.

7.6.2 Is the target east?

Task 1.5.2 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.

7.6.3 Converting absolute headings to relative ones

Currently the robot can sense its environment using the `look` method. The method takes *relative* directions in order to operate correctly. The methods 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.

Task 1.5.3 Complete the definition of the `lookHeading` method that takes an *absolute* direction and returns whether there is a wall, a passage, or a square the robot has been in before, much like the current `look` method.

7.6.4 Putting it all together

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 robot's surroundings. 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.

Task 1.5.4 Complete the `determineHeading` method so that it returns the heading into which the robot should move next and which exhibits the following behaviour:

1. If it can select a heading that will move the robot closer to the target then it should do so
2. It should not lead the robot into a wall
3. If the robot has the choice of more than one route, then it should randomly choose between them
4. 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.

7.6.5 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. Once again, we have included some unit tests to help you with this. The tests are contained in the `HomingControllerTest.java` file in the `src/test/` folder. You can run them with the same command in the terminal as before:

```
$ gradle test
```

This will run the tests for both `RandomController`, `HomingController`, and any you may have added while working on this coursework. The provided tests for your new `HomingController` test the `isTargetNorth`, `isTargetEast`, and `lookHeading` methods. They do not test the `determineHeading` method or the general behaviour of your `HomingController`. You may wish to add additional tests for those methods at this point to ensure that everything works as specified.

7.6.6 Final remarks

Ask yourself the following: can the homing robot always be expected to find the target? Carefully justify your answer.

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

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

An answer to this sort of problem is to build a *prototype*. Software developers will often produce some small cheap code to model a potential solution to a coding problem. The code does not need to be shown to the customer, as in itself it is not important. What is important is the input and output behaviour that the code exhibits. A customer 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, languages such as Python or JavaScript are favoured for the speed at which software can be developed and the size of the resulting code. They do not produce particularly fast or maintainable code, but then again at the prototyping stage this probably does not matter.

7.7 Marking & submission

This coursework is worth 15% of the overall module mark. It will be marked out of 100% as follows:

- 30% for *correctness*. You gain full marks here if all parts of the coursework have been attempted and are correct according to the specification.
- 30% for *understanding*. You should document your code with comments which explain the code sufficiently well so that someone who is unfamiliar with your code can understand it. You will also be asked questions about your code during the marking day.
- 20% for *good coding practice*. Code should be concise as well as readable, new methods should be introduced where needed, existing library functions used when applicable, etc. You may wish to consult a style guide for Java such as: `https://google.github.io/styleguide/javaguide.html`

- 20% for *testing*. You will do well here if you add comprehensive unit tests to your program. These should ideally make use of JUnit and integrate with the existing `gradle test` mechanism.

Submit a `.zip` or `.tar.gz` archive of the whole, completed project through Tabula by *9am on Wednesday 31st October (Week 5)*:

`https://tabula.warwick.ac.uk/coursework/submission/1f82736b-8797-469c-afbe-fbed98647a4e`

Be aware that, before the deadline, you can update your submission as many times as you wish, so it may be worth submitting a working submission early as insurance even if you plan to work on it further. However, after the deadline, you cannot make any more updates and you can only submit a late coursework if you haven't submitted anything yet.

7.7.1 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.

Chapter 8

THE MAZE II

In this second coursework you are required to develop more sophisticated robot controllers which adopt a more systematic approach to exploring a maze and that learn:

1. You will build a controller that systematically searches a dense maze (one that does not contain loops). In such mazes the empty squares form a *tree* of corridors one square wide. There are additional exercises at the end where we extend this controller so that it is able to deal with mazes containing loops[1].

2. You will then 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.

Your solutions are due at *9am on Friday 7th December (Week 10)*. On the same day, you will be asked to come in to the labs and demonstrate that your code works and that you have understood the material for each of the exercises. If we are not able to mark your work on that day, for whatever reason, then you will receive no marks[2].

8.1 Getting started

This coursework is builds on the first coursework and you should continue with the code that you wrote for that. There are some new bits of skeleton code which you can

[1] Mathematically speaking this means extending our robot from one that explores *trees* to one that explores *graphs*. You will explore these concepts more in other modules.

[2] If you have serious mitigating circumstances that are outside of your control, such as severe illness or a family emergency, please speak to your personal tutor about this as early as possible.

obtain by opening a terminal window, navigating to the root folder of the coursework project, and running the following command (if you have downloaded the `.zip` file from the module website):

```
$ git merge cswk2part1
```

If you have cloned the code from *e.g.* GitHub, use the following command instead:

```
$ git merge origin/cswk2part1
```

This will create a new file named `Explorer.java` in the `src/main` folder.

8.2 Task 2.1

An entirely new `Explorer` robot controller is going to be built; this should be done in the file named `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 to avoid collisions with walls.
- At junctions it should, if possible, turn so as to move into a square that it has not previously explored, choosing randomly if there is more than one. If this is not possible it should randomly choose a direction that doesn't cause a collision.
- Similar behaviour as with junctions should be exhibited at crossroads: the robot should select an unexplored path if possible, selecting randomly between the paths if more than one is possible. If there are no unexplored paths, 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 by the robot if it 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. A method stub for this method is already present in `Explorer.java`.

Task 2.1.1 Implement the `nonwallExits` method in your `Explorer.java` file so that it 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 long.

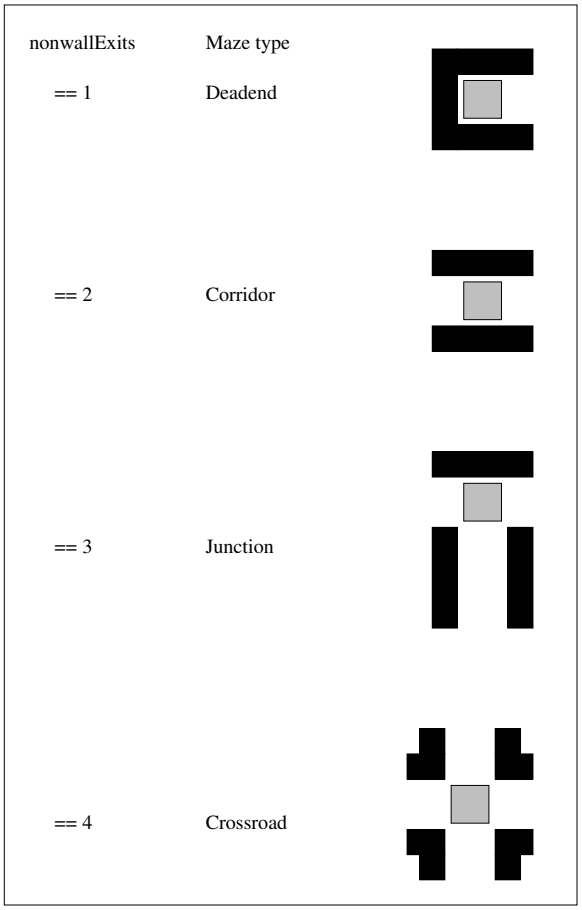


Figure 8.1: There is a clear relationship between the number of non-wall exits and the situation which the `Explorer` robot finds itself in which is demonstrated above.

Compile your code and verify that it works as designed. You may find it helpful to write some unit tests for this.

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. 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 `run` method so that it detects which of these four cases it is dealing with. If the robot is travelling along a corridor, then the 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.

Task 2.1.2 Modify the `run` method so that it records the result of calling the `nonwallExits` method. Store the result in a variable called `exits`.

Task 2.1.3 Now extend the `run` 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. Therefore, your controller will need to introduce a variable named `direction` and assign the result of calling the subsidiary method to that you have *e.g.* the following:

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

The `run` method will need to execute a call to `robot.face(direction)` for the direction to take effect.

You can compile your changes to check for any syntax errors. You will need to include empty definitions (stubs) 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.

Task 2.1.4 *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 of the maze. 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.

Task 2.1.5 *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 should 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. This should again not be longer than about ten lines of code.

Task 2.1.6 *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.

Task 2.1.7 *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 may find it useful to define an additional `passageExits` method which is similar to your `nonwallExits` method returns the number of passage exits in relation to the robot position instead.

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 any errors you encounter. When you have finished this you should be able to test your new `Explorer` controller in the Robot Maze environment. Test your robot controller carefully to ensure that it meets the specification.

8.2.1 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 yet unexplored exits.

This is the behaviour of the robot controller which will be built in this and the following section. In order to do this, the controller in `Explorer.java` will need to 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* algorithm.

Task 2.1.8 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 named `beenbeforeExits` that is similar to the `passageExits` method which you defined previously. This method should return the number of `BEENBEFORE` squares adjacent to the robot.

Task 2.1.9 The recording of junction and crossroad information[3] 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 en-

[3] From here on in the text we will refer to junctions/crossroads simply as junctions. Most of you will have realised that there is in fact no difference in the treatment of the two.

countered this junction. This information can, for example, be stored in three arrays as follows:

```
// Maximum number of junctions likely to occur in a given maze
private static int maxJunctions = 10000;
// Number of junctions stored so far
private static int junctionCounter;
// X-coordinates of the junctions
private int[] juncX;
// Y-coordinates of the junctions
private int[] juncY;
// Headings the robot first arrived from
private int[] arrived;
```

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 be).

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

On the first pass of a new run `junctionCounter` should be set to 0. This can be done by observing the `robot.getRuns` method which allows the control program to detect when it is computing the first run through a maze. This value alone is not enough as it will remain 0 throughout the robot's first run through the maze.

The code for this might look something like:

```
public class Explorer {
  // Data store for junctions
  private RobotData robotData;
  // ...

  public void run() {
    // ...
    // On the first move of the first run of a new maze
    if (this.robot.getRuns() == 0) {
      // initialise the data store
      this.robotData = new RobotData();
    }
    // ...
```

```
    }
    // ...
```

We assume that the constructor code for the `RobotData` class does something sensible, such as setting its `junctionCounter` to zero, for example.

Task 2.1.10 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 something like the following to the controller's `reset` method:

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

This requires you to add a `resetJunctionCounter` method to the `RobotData` class:

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

What this does is ensure that when you press the **Reset** button in the Robot Maze environment, your `junctionCounter` variable will be reset[4].

Task 2.1.11 Now modify the `run` 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 each array at the index indicated by the `junctionCounter` variable. Remember to increase `junctionCounter` by 1 after a junction has been recorded.

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 with `robot.getLocation().x`), the y-coordinate of the junction (which you can access with `robot.getLocation().y`), and the robot heading (which you can access with `robot.getHeading()`).

Task 2.1.12 When you have completed the above modifications, test that the information recorded is correct by printing it out on the screen (*e.g.* by adding 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 your `printJunction` method) the following information (your formatting may vary):

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

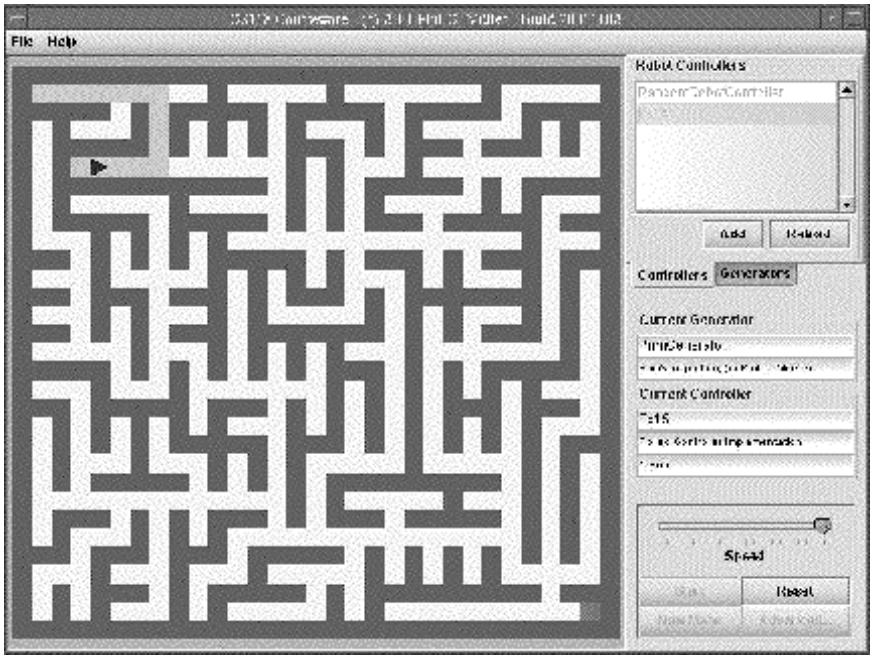


Figure 8.2: The robot has passed through three junctions. The information which is stored in your new `RobotData` class includes the x- and y-coordinates of these junctions as well as the heading of the robot as it arrived at these junctions.

```
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.

8.2.2 Using stored data

Modify the `Explorer` controller 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* the maze.
- 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, for example, be implemented as a class variable of the `Explorer` class:

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

Task 2.1.13 Add the `explorerMode` member 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[5].

We now wish to implement two new methods, `exploreControl` and `backtrackControl`. The robot will be able to switch between these states depending on the situation.

Task 2.1.14 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 the `run` method and paste it into the `exploreControl` method. You will find that the `run` method then contains just the basic control code 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` field to zero when you reach a dead end.

Task 2.1.15 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. What will this method return if it is called when the robot is at a junction which it has not previously encountered?

[5]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 modified in the previous exercise.

Task 2.1.16 Your backtracking control method can now be written. Start by introducing a new method named `backtrackControl` 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* method.

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. 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.

Task 2.1.17 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 one of the teaching assistants or module organisers. Time will be set aside in the labs to discuss this set of problems.

8.2.3 Worst-case analysis

Task 2.1.18 Will the `Explorer` controller 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? Explain your answers.

8.3 Task 2.2

In a "real life" situation it may be highly desirable to minimise the amount of data storage required by the control program. Your objective for this task is to re-implement your `Explorer` controller so that the robot does not require the location of *every* junction to be recorded. Begin by creating a duplicate of `Explorer.java` by executing, for example, the following command in a terminal:

```
$ cp src/main/Explorer.java src/main/Explorer2.java
```

You will also need to change the name of the `Explorer` class to `Explorer2` and register it with the Maze Environment. You can do this by adding a line for `Explorer2` to the `main` method in `src/main/Program.java`:

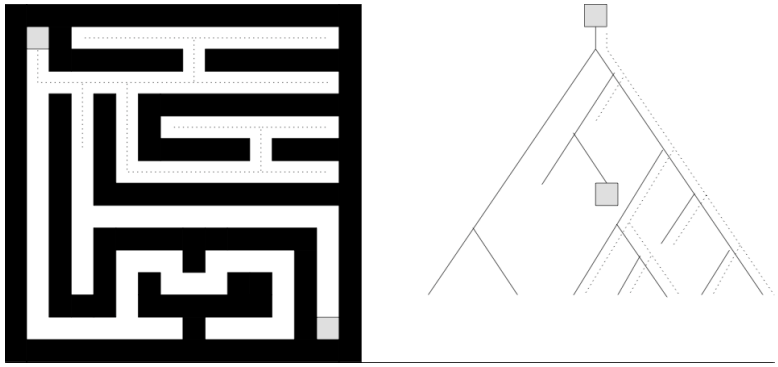


Figure 8.3: Representing the maze as a search tree

```
public static void main(String args[]) {
  // initialise the maze configuration and add an instance of
  // the random robot controller
  MazeLogic logic = new MazeLogic();
  logic.getControllerPool().addController(new RandomController());
  logic.getControllerPool().addController(new HomingController());
  logic.getControllerPool().addController(new Explorer());
  logic.getControllerPool().addController(new Explorer2());

  // run the maze
  new MazeApp(logic);
}
```

8.3.1 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. If the target is not found, the robot backtracks to the junction at which the choice to explore the path was made. This procedure is analogous to searching one part of the 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 through the tree have been 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.

Task 2.2.1 Modify `Explorer2` so that it uses depth-first search.

8.4 Task 2.3

Currently our `Explorer2` robot uses a depth-first search. This is all very well as long as our maze does not contain loops. We say that a maze containing loops is *loopy*. As soon as a loop is introduced to the maze, the current exploring algorithm breaks[6].

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.

8.4.1 Single loops

Task 2.3.1 Create a copy of either `Explorer.java` or `Explorer2.java` named `Explorer3.java` and create a controller named `Explorer3` which is able to navigate mazes with single loops. Some of you may find it easier to work from your `Explorer` answer, others may find it easier to extend from `Explorer2`.

[6] You can test this out for yourself by using the "LoopyGenerator" from the "Generators" tab in the Robot Maze software.

[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.

8.4.2 Multiple loops

Task 2.3.2 Extend your `Explorer3` controller so that your robot can navigate mazes with multiple loops.

8.5 Task 2.4: The Grand Finale

The exercises in this section are designed to test the very best of you. Although it should be said that it is possible to get an acceptable mark in the practical component of the CS118 module without having done the exercises in this section. 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 final 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 we expect to see lots of exciting and innovative solutions. You can find the marking criteria in Section 8.7.

The programs which you will write for this section are probably the most interesting in this module. 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.

There will also be a prize for the person who develops the best solution. You could win this by going beyond the specification, or perhaps you could produce the shortest solution to the grand finale[8]. Previous winners have also turned the maze environment into a game, and created interesting graphical displays to show the tree representation of the maze as it was being explored.

8.5.1 The Grand Finale

You receive a call from NASA (as you do). They have seen your robot in action and plan to use some of your code in their next mission[9]. 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[10]. What they would like is a robot that learns.

[8] The current record for shortest solution is 5 lines of code, including all of the Java filler! Though a submission of 30 lines or below would be very impressive for first year students.

[9] Heaven forbid.

[10] Unlike NASA, of course.

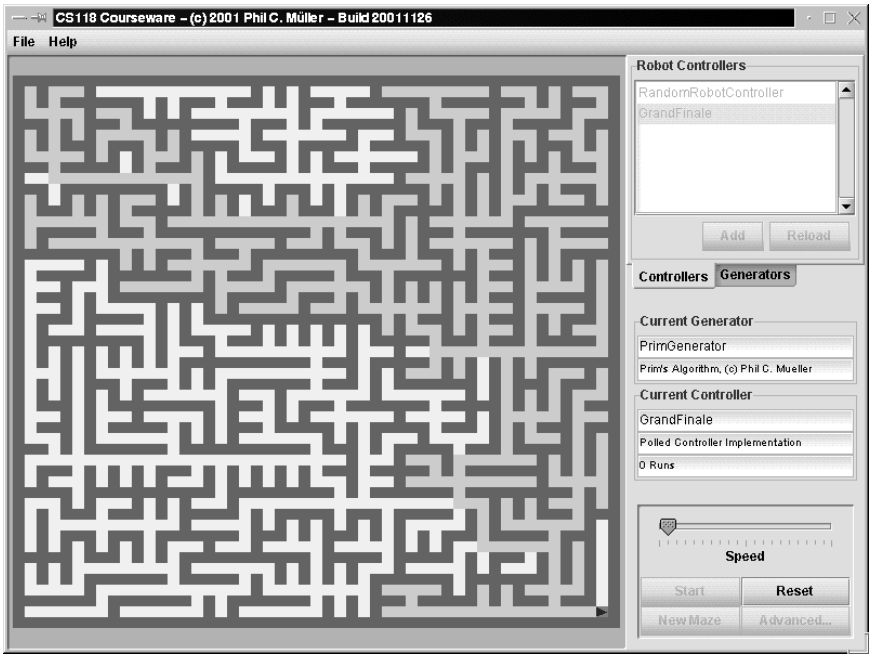


Figure 8.4: A trace of the `GrandFinale` robot the first run through the maze.

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 during subsequent runs.

The plan is to build on your previous solution, `Explorer3`. The robot will search the maze (in its `Explorer3`-like way) the *first time* the robot is run through a maze. However, the *second time*[11] time the robot is run, it will use its virtual map (its memory if you like) from the first run 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. In other words, it will always take a direct path to the target.

An example of this behaviour is demonstrated in Figure 8.4 and Figure 8.5. In Figure 8.4 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. However, in the end, it reaches the target.

When the robot is run again, shown in Figure 8.5, 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 this part of Coursework 2 is to implement this behaviour.

There is more than one way of doing this and in order to provide you with some (modest) assistance we shall describe two approaches which you might like to take. We do

[11] or more

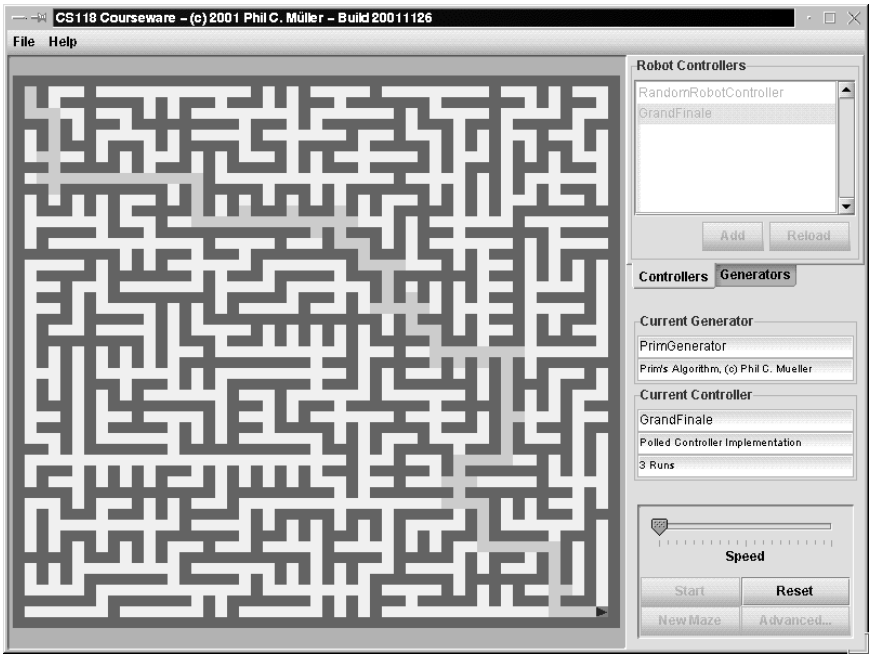


Figure 8.5: A trace of the `GrandFinale` robot the second time it runs through the maze.

not mind which of these you go for, if any.

Regardless of which approach you take, you should create a copy of `Explorer3` named `GrandFinale` (in `GrandFinale.java`) which you make your changes to.

Route A

Task 2.4.1 Modify `GrandFinale` 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 to the standard output 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 8.6 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 Figure 8.6 you 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

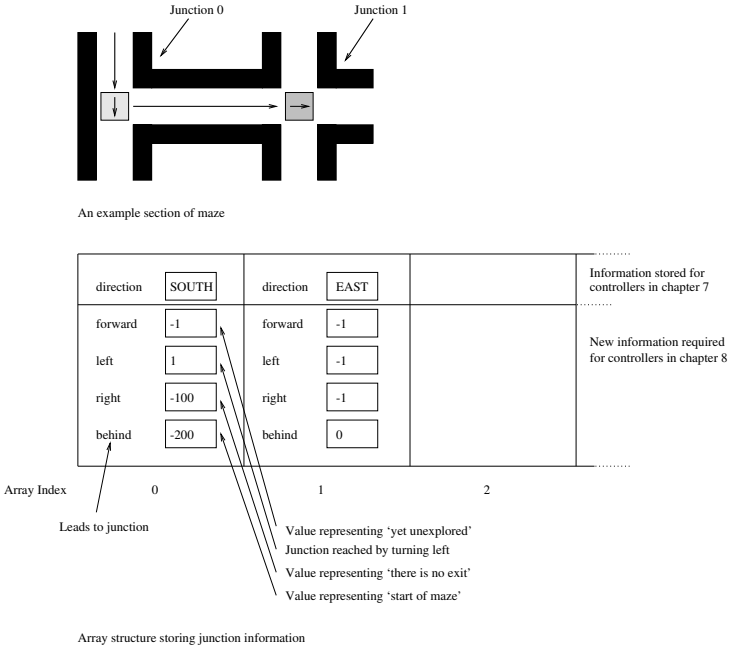


Figure 8.6: Storing more junction information.

explored can be represented as a negative number. Remember that a method exists which tells you whether you are simulating the first run in a maze. Use this to detect whether the maze has changed since the previous run.

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.

Our 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 algorithm. Then try to implement and test your ideas. If inspiration fails to strike even after thinking hard, the teaching team 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 teaching staff – the chances are that you are both right.

Route B

Task 2.4.2 There is a solution to this problem which many perceive as simpler to that presented in Section 8.5.1. This route is based on your answer to Section 8.3 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.

8.6 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 also have touched on *data structures* and *algorithms* and even *AI*.

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

You are not going to have mastered programming in ten weeks, but we hope this has been a fun and enjoyable experience for you. Learning to program involves a lot of trial and error, can be frustrating at times, but is also incredibly rewarding once things work out. Even as experiences programmers, we often write code that fails to work straight away, doesn't make any sense, or is very inefficient. However, the more we do, the better get become. By the summer many of you will be taking up well paid summer jobs fixing other peoples' Java code. This may sound hard to believe right now, but each year it is the same and you will be amazed by how quickly you will gain more and more confidence programming if you keep practising!

8.7 Marking & submission

This coursework is worth 25% of the overall module mark. It will be marked out of 100% as follows:

- 20% for *correctness*. You gain full marks here if all parts of the coursework have been attempted and are correct according to the specification.
- 20% for *understanding*. You should document your code with comments which explain the code sufficiently well so that someone who is unfamiliar with your code can understand it. You will also be asked questions about your code during the marking day.
- 20% for *good coding practice*. Code should be concise as well as readable, new methods should be introduced where needed, existing library functions used when applicable, etc. You may wish to consult a style guide for Java such as: `https://google.github.io/styleguide/javaguide.html`
- 20% for *testing*. You will do well here if you add comprehensive unit tests to your program. These should ideally make use of JUnit and integrate with the existing `gradle test` mechanism.
- 20% for *improvements and extensions*. This is an opportunity for you to demonstrate creativity and advanced understanding. You could achieve this in many different ways, such as adding additional functionality, improved algorithms, etc. The amount of marks awarded will depend on the complexity and creativity of your extension(s) and improvement(s).

Submit a `.zip` or `.tar.gz` archive of the whole, completed project through Tabula by *9am on Friday 7th December (Week 10)*:

`https://tabula.warwick.ac.uk/coursework/submission/d8b779c9-bc72-4b0b-b032-4e242af900c2`

8.7.1 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.