

Lancaster University Department of Physics
PHYS281 Programming and Modelling Project

Iain Bertram, Jaroslaw Nowak

Michaelmas 2016

Contents

A	Introduction	2
B	Reading	7
C	Programming Basics	8
D	Variables — declaring and assigning values	17
E	Input, mathematical functions and conditional branching	25
F	Iteration	41
G	Multiple methods	52
H	Arrays	68
I	Numerical integration	83
I.1	Left-Endpoint Rectangle Rule	83
I.2	Mid-Point Rectangle Rule	84
I.3	Trapezium Rule	84
I.4	Accuracy of Numerical Integration	86
I.5	An Exercise in Numerical Integration	89
J	Multiple classes	93
J.1	Static classes	93
J.2	Non-static classes and objects	95
J.3	What is an Object?	95
J.4	Creating an object	96
J.5	Static & Non-static methods	99
J.6	Constructor methods	101
J.7	Using private member variables	102
K	Designing and Implementing Classes	109

K.1	Program & Class Design	109
K.2	Class Diagrams	112
K.3	Vectors	115
L	Gravity simulation	119
L.1	A ‘Particle’ in The Earth’s Gravitational Field	119
L.2	Particle traversing a tunnel through the Earth’s core	123
L.3	Orbiting Massive Particles (E.g Planets)	124
L.4	Different methods of simulating kinematics	124
M	Physical Simulation Project	127
	Bibliography	II-i
	Appendices	II-i
Appendix 1	Dealing with errors	II-ii
Appendix 2	More advanced input and output	II-iv

A Introduction

This ten-week module is intended as an introduction to computer programming for physicists. Being able to program a computer is an important skill in scientific work. Scientists encounter new situations all the time, and we need to either combine existing applications into new programs or write new applications from scratch to manipulate and analyse data. There are many different programming languages, and each of them has strengths and weaknesses. The programming language we will use for this module is called Java. This language, developed by Sun Microsystems, is one of the most popular computer programming languages in the world [1]. It is best known for its use in association with the World Wide Web and the Internet, but over the next few weeks you will see that it can also be applied effectively to scientific problems. For the purposes of this module, some advantages of Java are:

- It is portable — Java programs run on most computer systems in use today;
- It is freely available;
- It is relatively simple;
- It is widely used;
- It is more integrated with graphics than some other languages

Java's portability means that you can use your programs on other types of computer easily, and the program should behave in the same way regardless of the computer you are using — this is known as *platform independence*. It is for this reason Java has become very popular on the Internet.

Java has a syntax that is very similar to many modern programming languages (C, C++, perl, Python etc). This means that by learning this language, the others will be easy to pick-up. (For a long time, FORTRAN was the most popular programming language in physics, and it still has attractive features. However, it is not widely used outside of physics. Modern versions of FORTRAN use similar structures to those in Java and the other languages we've mentioned.) Java is also inherently 'Object Oriented' and we will talk a lot about this in the second half of the module. This style of programming is extremely widespread nowadays so it makes sense to learn a language that was designed for it. Finally, Java skills are very much sought after in the modern job market, and so will be a useful addition to your CV.

Commercial analysis packages (e.g. Matlab, Mathematica, Maple, LabView) are also useful and powerful, but they will often not do exactly what you want, and in each case they have a facility for you to program within them. In general, once you have the ability to program, you will be able to make computers do what you want them to do, not just what software companies think you should do with them!

These notes, example files, tips and up to date information about this course can be found at <https://modules.lancaster.ac.uk/course/view.php?id=16164>

Objectives

This course gives you the opportunity to develop the skills to:

1. understand the basic concepts involved in writing computer programs;
2. learn at least one way to program on a PC;

3. understand the main features of the Java language;
4. design and write simple programs in Java;
5. devise and use test procedures for your programs;
6. learn how to 'debug' your computer programs;
7. design and write computer programs to solve numerical problems;
8. design and write computer programs to model a physical system.

Assessment

Assessment is based on the answers you give to the exercises included in the course. We require that all your work be submitted electronically to Moodle. You should always save copies of your work on a cloud drive (e.g. Dropbox) or a memory stick and in your home area on the central server (your H: drive). The computers in the university labs often have their disks re-initialised, so do not rely on your work remaining on them.

The exercises will be numbered w.n, where w is the week number and n is the exercise number for any given week.

The short weekly exercises must be submitted electronically by Friday at 14:00 in weeks 1 to 3 and Thursdays in weeks 4 to 10. Late work will be accepted until the following Monday at 14:00. All late work will have the standard penalty applied (see the [Undergraduate Courses Handbook](#)).

The date and time has been chosen to ensure that your work can be marked and returned before you submit the next exercise the following week.

... HAND IN TIME: Friday 14:00 (weeks 1 to 3...)

... HAND IN TIME: Thursday 14:00 (weeks 7 to 10...)

The larger pieces of work (such as reports) will have separate deadlines. Check the hand in dates carefully and make sure all work is handed in on time. If you need an extension please contact one of the academics in charge of the course and request an extension BEFORE the hand in date.

All summative work must be completed to pass this course. If you leave it until the final week to start work on the final project which forms a major part of the summative assessment that you are very likely to fail this module!

During the lab sessions you will find it helpful to take notes in a physical or electronic logbook. Write down your notes and answers in your logbook as you work. Do not wait until the end to write everything up as you are likely to run out of time if you do this.

Read each question carefully and make sure you answer all of the questions and include all the requested information. This is very important as we will be using automated marking for many parts of the weekly exercises. These exercises will be denoted by the message:

This exercise will be marked by the robot.

in these notes. For these exercises it is especially important that you provide the **exact** output requested in the question. The 'robot' is not clever (it is just another computer program) and its automatic marking

system cannot interpret many variations on the answer. If the solution does not match the criteria then you could get no marks. If in doubt then ask one of the lecturers or lab demonstrators.

Other exercises are marked by real humans, so in all cases make sure that your computer programs are written in a clear way so that a reader can understand what you have done, otherwise you cannot be credited with full marks. This is especially important when it comes to putting explanatory comments into your programs. It is vital that code is well-documented, both for your own benefit if you return to the code in the future, and for the benefit of anyone trying to understand your work later. For the same reason, you should choose self-explanatory ‘Class’ and ‘variable names’ (see later).

Your answers will be assessed on the following criteria:

- quality and correctness of the answer — applies to both written answers and program listings and output;
- code presentation — your programs should be clearly laid out with appropriate use of variable/method names, comments and indentation;
- programming style — solutions should be simple, efficient and effective.
- documentation and design — where required you should provide the appropriate documentation for all of your work and any design notes written in creating your program.
- testing — when appropriate, you need to demonstrate that your program works by testing its key features and comparing the resulting output with what you expected, e.g. a result calculated by hand. Make sure you write down the details and results of tests at the time you carry out the test.

Types of Assessment

There will be two types of assessment in this module, **Formative** and **Summative**.

The main goal of the **formative** exercises is for you to learn programming by completing the exercises and to allow us to give you some guidance on your performance. For these exercises most feedback will be given to the entire class rather than individually. At the end of the course all of the **formative** assessment will be combined into a single mark making up 20 % of the total grade for the module.

The **summative** assessment will make up the remaining 80 % of the modules mark and is intended to measure your overall understanding of the course and your ability to apply computing to physics problems. These exercises usually require the submission of a report as well as computer code. Please see the departmental report guidelines [2] for a clear description of what is required in any report. Each **summative** exercise will have a mark breakdown included in the exercise. You will get written feedback on all **summative** assessment. The contribution of each **summative** exercise is proportional to the number of marks allocated to the exercise. This means that the exercises from week 6 onwards make up the bulk of your final grade. Note that in order to get good marks in the course you must leave yourself enough time to write up your reports.

The **summative** assessment will be marked on the University scale:

Broad Descriptor	approx %	Grade	Agg. Score	Class
Excellent	100	A+	24	First
	80	A	21	
	70	A-	18	
Good	67	B+	17	Upper Second
	63	B	16	
	60	B-	15	
Satisfactory	57	C+	14	Lower Second
	53	C	13	
	50	C-	12	
Weak	47	D+	11	Third
	43	D	10	
	40	D-	9	
Marginal fail	31	F1	7	Fail
Fail	18	F2	4	
Poor fail	9	F3	2	
Very poor fail	<9	F4	0	

Submission Format: You will be submitting your work electronically. Unfortunately we cannot accept all possible document types. For the actual code you are required to submit the .java files (not the .class files!), while for your reports you can submit pdf or plain text files. Any work submitted in other formats (e.d. .doc, .docx, etc) will not be marked unless you have made special arrangements with the module convener.

....Only .java, text and pdf files will be accepted....

You should complete each exercise in a separate computer directory (folder). If there are multiple files you may be required to submit the entire directory in a zip file. The submission on link on the Moodle site will specify if this is the case. If you need help with this see a demonstrator during one of the 281 labs.

It is strongly recommended that you use L^AT_EX to produce any reports (see <https://modules.lancaster.ac.uk/course/view.php?id=16164>).

If you need to make data plots we recommend you use the QtiPlot package:
<http://www.lancs.ac.uk/iss/software/qtiplot/>.

Note that the version of QTIPlot available through the university is more recent and reliable than the free version downloadable from the www.

Plagiarism: You have previously been made aware of the University rules on plagiarism, and have agreed to abide by them. In general, a good programmer will reuse code developed by him/herself and by others in order to build new applications. However, in this course you are meant to develop code yourself as part of the learning experience. Automated code-checking tools may be used to look for copying (both from other students in the class and from elsewhere). We appreciate that programming can be confusing, frustrating and intimidating if you have not done it before, but do not be tempted to copy blindly without understanding what you are doing. We of course do hope and expect that you will be supportive of each other in the lab and help each other, but it is important that ultimately the work you submit is your own.

Lectures and Labs

In the weekly lectures we will cover some of the main concepts needed to solve the weeks exercises and discuss some of the common pitfalls and problems encountered when programming. Like any language (human or computer) the key to understanding it is to use it in a practical situation and experiment with it. Hence the emphasis of the course is on the laboratory sessions which are intended to operate as workshops where you can get assistance with your programming and can ask questions of the teaching staff. We cannot of course individually fix every problem which you will encounter as you attempt the exercises, but we hope that you will rapidly develop the skills you need to do this yourselves. The more preparation you do outside of class the better you will perform in this course.

Using the Book

The recommended text for this course is *Introduction to Java Programming* by Y. Daniel Liang [3]. The book is intended to provide an alternative source of information on Java and programming in general to the lectures. At the beginning of each section of these notes there will be a set of suggested reading. To gain full benefit from the course you should read this material at your earliest opportunity.

If you read the suggested chapters before the class you will have a much better idea of where to find the information required. It is well worth reading through the examples to understand how they work.

Using the Web

For programming in Java the most useful web-pages are probably those at the Oracle java documentation site

(<http://docs.oracle.com/javase/8/>). In particular we recommend the java tutorials at <http://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html>. There are many other java resources online that you can try. If you find one that you like then you can post a link to it on the Moodle forum.

B Reading

Week1: Sections C, D & E of these notes; Liang: Chapters 1 & 2 & 5.10

Week 2: Section E & F of these notes; Liang: Chapters 2, 3, 4

Week 3: Section G of these notes; Liang Chapter 5

Weeks 4: Sections H of these notes; Liang: Chapters 6 & 7

Week 5: Section I of these notes; Liang Chapter 5 & 7

Week 6: Sections J.2 and K of these notes; Liang Chapters 8 to 10

Weeks 7 – 10 : Sections K & L of these notes; Liang Chapters 9 to 10

C Programming Basics

Computer programs and java

A computer is essentially just a machine which stores and manipulates data according to a clear set of rules. A computer program is fundamentally just instructions for a computer to follow. *Running* a program means the computer executes the instructions inside the program.

Programming (also known as coding) in Java involves the programmer writing a set of instructions in the Java language. The file this program is saved in, is known as a *source file* and has a name of the form something.java. The source file must then be *compiled*¹. The compiler is itself a program which reads and tries to make sense of the source file you've written. The compiler translates your program and writes it out into a second file (called something.class). This second file contains the same program instructions but written in a different form, to better suit the computer rather than you the programmer. Finally, whenever the program is *run* the contents of this second file go through an *interpreter* which turns the contents into instructions that the computer's processor² can understand and carry out.

Basic program structure

We will start right away with a look at a Java program. For most of this course all your programs will have the same basic structure, which is reproduced below. This contains all the essential parts of code that must be included in any program.

```
1 public class Simple
2 {
3     public static void main(String [] arguments)
4     {
5         // Some Java program code goes here
6     }
7 }
```

So what does this all mean?

public class Simple

Java programs are made up of blocks of code called *classes*. The first line says that we are going to define a *class* and we have chosen to call it Simple. You could of course give it any other name but the convention is for all class names to start with a capital letter. The class Simple will contain anything written within the outermost set of curly brackets {}. The purpose of the word `public` is something we will come to later; for now you should always use it when you are writing a *class*.

public static void main(String[] arguments)

Classes themselves contain smaller blocks of code. These smaller blocks are called *methods*. This line (3) is the start of a special *method* called *main*. When the computer runs any Java program it starts by looking for the *main method* and carries out whatever instructions are within it. All programs must contain one *main method*.

Each method is just a collection of commands which the computer will carry out.

¹Don't worry if you don't know what this means yet; you will be shown how to compile a program during the first session.

²The processor is the part of the computer which does all the work, e.g. an Intel Pentium processor in a PC.

lines 4-6

Instructions for what the *main method* should do are in the form of some Java code which goes in the main method — i.e. within the corresponding set of curly brackets, `{}`. The `// Some . . .` part on line 5 is called a *comment*, a note to yourself or others reading the file which is ignored by the computer. There are other ways to write comments which will be covered later in these notes.

Also note how within the class all the other lines have been indented by four spaces, and within the method everything is indented³ by a further four spaces. The spaces don't have any effect on the program — they are just used to make the structure of the program clearer to understand at a glance. Indenting your code is important to make it clear, so please follow this convention. You may also use blank lines occasionally for extra clarity.

Editing a program with Jedit

In this section you will learn how to edit a java program file. We will be using the program `jedit`⁴ to create the files. It is a simple text editor designed for computer programming. It highlights any *keywords* which have special meaning in java, and it helps with keeping your code indented properly.

Before creating any java files you should create a folder (directory) structure in your central University file store⁵. It is suggested that you create a directory for the course (e.g. `Phys281`) and subdirectories for each week (e.g. `Week_01`) and further subdirectories for each exercise (see Fig. C.1).

Once you have created a directory for the first exercise in week 1 you can start `jedit`. Follow the directions described in Fig. C.2. Jedit is a simple source code editor that has

- syntax highlighting⁶
- auto indent
- intelligent bracket matching

See the Jedit web site for details <http://www.jedit.org/>.

³Indentation is the standard way of making your code easy to read see section 1.10 of [3]

⁴If you cannot open this on a Mac due to security issues you can run the following command
`xattr -d -r com.apple.quarantine /Applications/jEdit.app`
to create an exception.

⁵Drive H: (see <http://www.lancs.ac.uk/iss/myaccount/pclan/>)

⁶This works once you have saved the file with a `.java` extension.

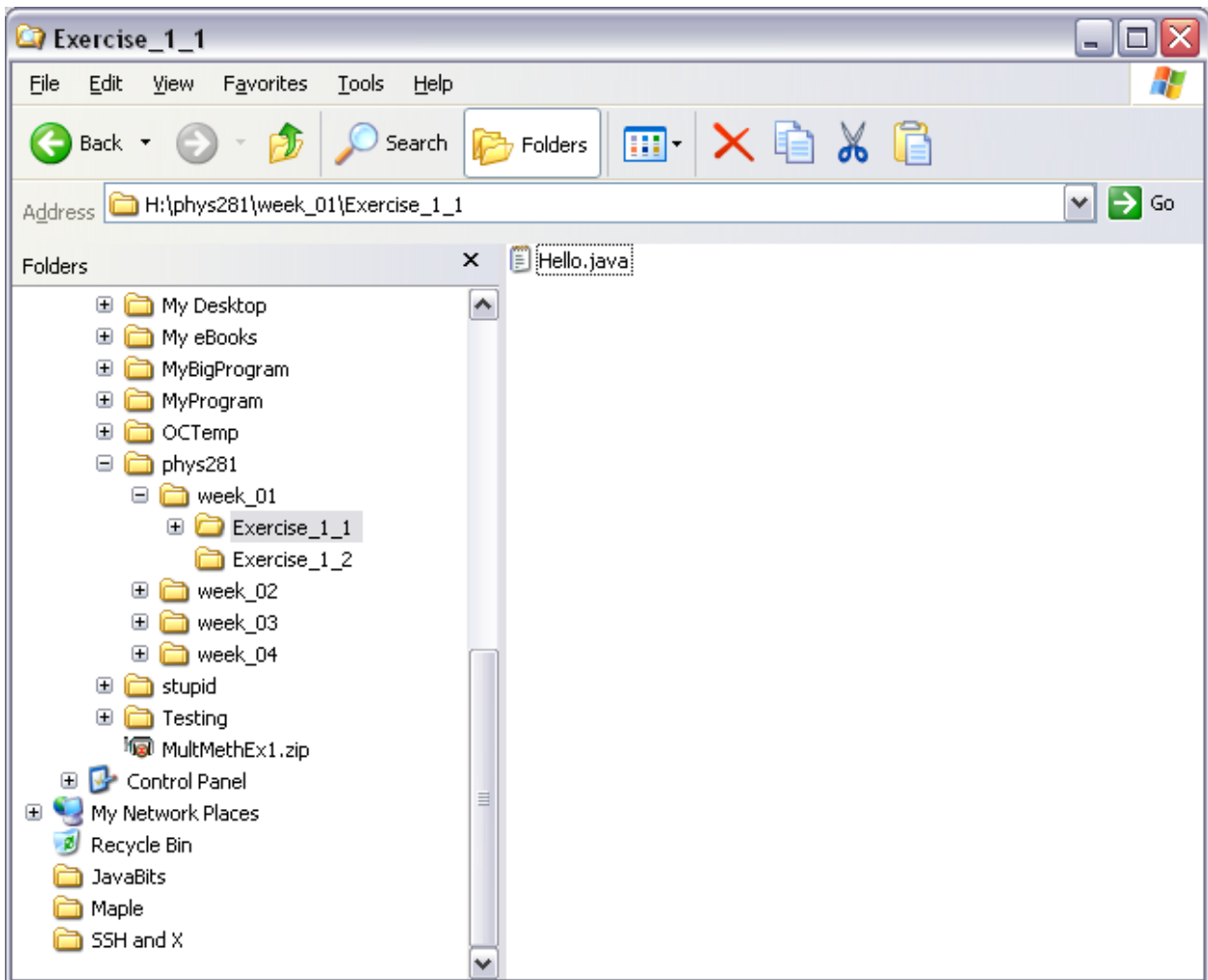


Figure C.1: *Example directory structure in windows.*

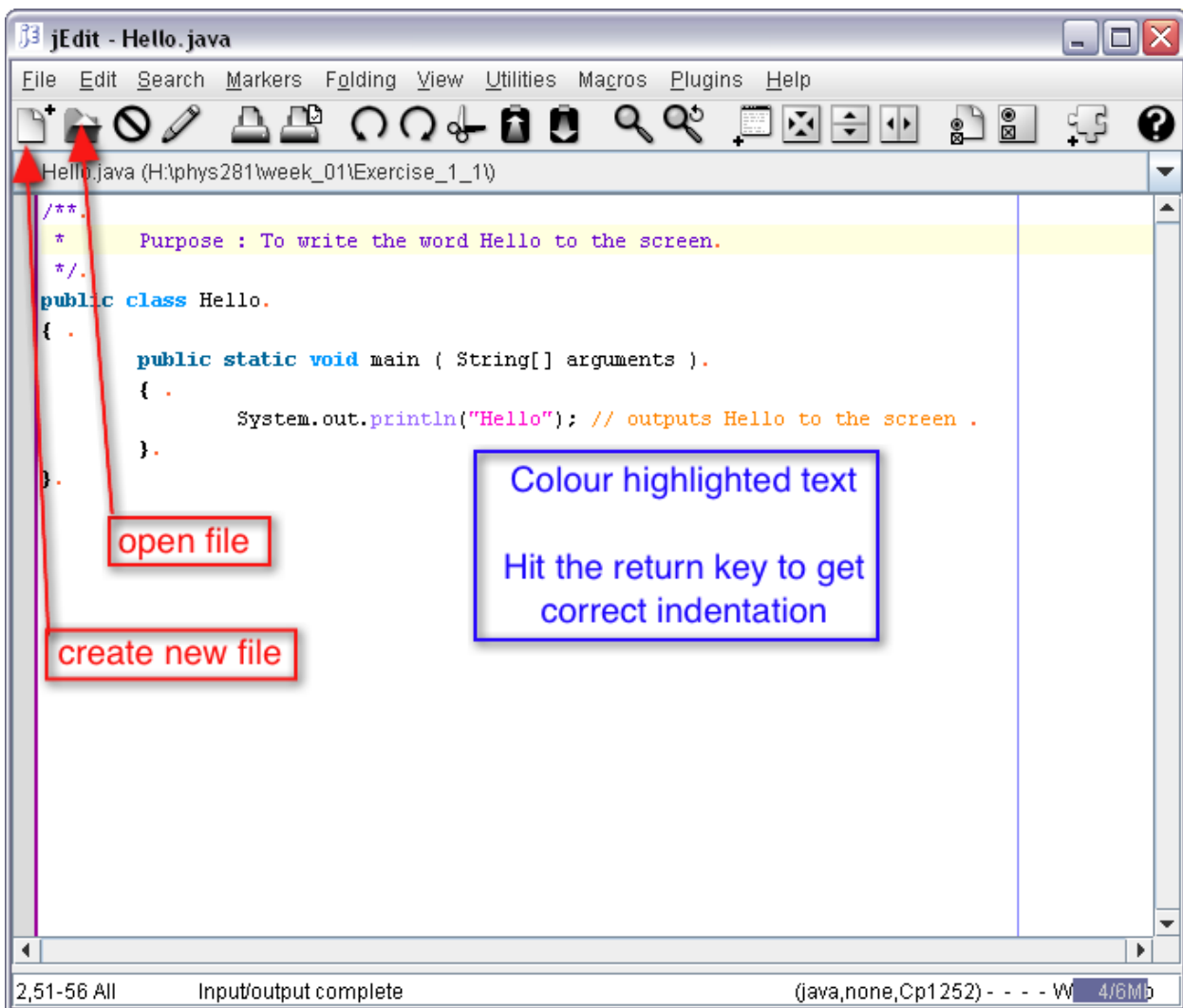


Figure C.2: Example of the jedit window

Running a program on a University PC

In this section you will learn how to run your first Java program.

Here is perhaps the simplest program you can write in Java.

```
/**
 * Purpose: To write the word 'Hello' to the screen.
 */

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

The program simply writes 'Hello' to your screen. You will see that it has the same basic structure as was shown before. The semi-colon ; is used in Java to indicate the end of a line of code, or one instruction. Don't worry about understanding the `System.out.println`⁷ just yet, the program is just for use in the exercise that follows. The purpose of this exercise is to show you how to run Java programs on the computers you are using.

You need to know how to compile and run your program. If you are writing a program, you must save it with a file name which is **exactly** the same as the class name (including the same upper and lower case letters) with the file extension 'java', e.g. `Hello.java`.

If you are using a PC, then you can work with Java by typing commands into a MSDOS window also called a terminal or command window. To open a MSDOS window, go to the Windows Start menu → Programs → MS-DOS prompt, or go to Start → Run and type "command" or "cmd". On Windows 8 or 10 then you can just type "cmd" in the search box.

To change to the folder where you saved your Java file from the text editor, for example `H:\Phys281`, type:

```
H:
cd H:\Phys281
```

Some other useful MSDOS commands and tips are given later in this section.

In order to run a java program, you must first compile it. To compile a java class defined in the file 'ClassName.java', you would simply type:

```
javac ClassName.java
```

To run a compiled class, type either ⁸:

```
java ClassName
or
java -cp folderName ClassName
```

⁷This command tells the computer to print the text contained in the brackets to the command line (i.e. your MS-DOS windows or Unix shell.)

⁸The -cp stands for 'classpath' and tells java where to find your compiled code.

If you replace `folderName` with `.'` in this last command it is understood by the computer as meaning 'the current folder' and then both commands behave identically.

MSDOS tips

Directories

Directories, also known as folders, are organised in a tree-like structure. To move around directories:

```
cd .. (change directory) moves to the 'parent' directory;  
dir lists the files and sub-folders in the current directory;  
cd stuff moves into the directory called "stuff", if it exists.
```

Screen size

You can change the font size of the MSDOS window and the number of lines it displays. Click on properties on the MSDOS toolbar, then once you have changed the settings, exit the MSDOS window and start it again for them to take effect. NB The font size must be small enough to accommodate the number of lines you wish to display on the screen.

You can switch between full screen mode and back by pressing `<Alt>+<Return>`.

Editing on the command-line

You can press F3 to copy the previous command.

Once you have typed in part of the directory name you can use the "tab" key to automatically complete the name.

For richer command-line editing features, run the `doskey` program (i.e. type `doskey`). Then you can use left and right arrows, insert mode to edit a line and the up/down arrows to choose from the previous lines you entered.

Cut and paste

To copy text from the MSDOS window, click on the top-left icon of the window to get a menu →Edit →Mark. Now use the mouse to mark out an area of the text in the window to copy, then press Return to copy it.

To copy the whole MSDOS window as a picture, press `<Alt>+<Print Screen>`. You can paste this into other applications. This works for any window, not just MSDOS. To store the image for later use you can use the "Paint" program that comes with Windows.

Exercise 1.1 _____ (Formative: 4 Marks)

This exercise will be marked by the robot.

To ensure that you can use your computer to write, compile and run Java program type in the following example, save it, compile it and run it.

Submit your java program (.java file) to Moodle which has the URL

<https://modules.lancaster.ac.uk/course/view.php?id=16164>

Do not change the file name from 'Hello.java' before uploading it as this will stop it compiling!

```
/**
 * Purpose: To write the word 'Hello' to the screen.
 */

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

Do not change anything in the file. Type it in exactly as it appears here and then check you can compile and run it. If it does not compile or run then you have probably mistyped it and introduced a mistake (a 'bug') which will need to be fixed. See below for notes on each stage of the process. You will be marked on the **exact** output of the program. Remember that the robot is not clever so don't change 'Hello' to another word or add extra punctuation in the version you submit.

Writing

Type in the program **exactly** as shown in the example above. Take care to copy upper and lower case correctly from the example, as Java is *case sensitive*: the program will not work if you get this wrong. Don't try cutting and pasting from this .pdf document as there is an incompatibility between the way that Windows interprets text characters and the standard used in .pdf documents which is very likely to cause problems⁹.

Saving

The code should always be saved in a file that has the same name as the *public class* and with the extension .java so you should save the above code under the name 'Hello.java'. This is your source file. Note that the filename is case sensitive.

⁹Windows uses a non-standard character encoding called Cp 1252.

File Naming

One of the most common problems in this course is the creation of java files or report names with “bad” file names. Please only use standard characters in your file names and do not include spaces in the file name. To be completely safe only use letters, numbers and dashes (-). Special characters that you must not use are:

\$ % () + , . ! = @ [] ^ ' { } ~ " * / : < > ? \ | ' & ; #

Compilation

In the command window type

```
javac Hello.java
```

The compilation process produces a file called Hello.class, which then is used when you run the program.

If you get error messages when you compile the program, check very carefully what you have typed, then try again or ask a demonstrator. Remember that java is case-sensitive.

Especially when you are creating longer programs you are bound to make mistakes so you will often get errors when you first compile a program. Deal with one error at a time starting from the first that the *compiler* found. Sometimes an error at the beginning of a program can cause many errors later on, and if you correct the first problem and compile again you may find some of the other errors have vanished. The error message will contain the line number and some description of the error which you can use to help find what went wrong. Appendix 1 contains more help on errors, you should refer to this when you come across problems.

Running

Try typing

```
java Hello
```

You should now see the word ‘Hello’ printed on the screen.

If you don’t then you may get a message about an *exception* written to your screen though it is unlikely to happen with a program as small as we are dealing with here. *Exceptions* arise from problems that occur during the running of the program that were not detected by the compiler. You will be warned about the most common exceptions as we go along.

If you have an *exception* now, ask for help. You will learn how to solve these problems yourself as you gain experience.

Comments

Comments in a computer program are notes you make to yourself (or any one else trying to read your program). They will be ignored by the compiler and do not affect the way the program runs. For this reason you must indicate to the compiler where each comment starts and stops. There are three ways to do this depending on the circumstances. The first looks like this:


```
/**
 * Purpose: describe what the program does,
 * add any extra lines needed using an asterisk
 *
 * @author Joe Bloggs
 * @version 1.1
 * 12 September 2014
 */
```

This form should be used at the beginning of all your files and give any necessary information about your code, including its purpose, your name, the date and the exercise number where applicable. If you edit your program you may find it helpful to include a modification history here as well, stating what changes were made and when. This will help you keep track of different versions of your program.

The two lines that begin with “@” are the standard form for java comments and allow you to automatically generate standard web page documentation from your programs. @author is followed by the name of the person who wrote the code (you can have as many authors as required), and @version is followed by the version number of the program (so that you can distinguish between different versions of the same program). We’ll explain how to generate the web pages later.

The other two types of comment are:

```
// comments go here, rest of line is ignored

/* comments here may use
   more than one line
   and finish with */
```

When // appears on a line, everything to the right up to the end of the line is ignored by the compiler. Java code can appear to the left of //. Longer comments may take the second form shown. Anything between /*...*/ is considered to be a comment by the compiler and ignored. Unlike //, the comment can span several lines.

As // comments may be placed within /*...*/, you may try putting the latter around a block of code in your program that you think is causing an error or exception, as long as the block of code only includes // comments¹⁰. Note that two sets of /*...*/ comments can not be nested within each other.

It is important that you add sufficient comments to your programs so that it is clear what they do and how they work. When someone else reads your program they will need more help to understand it than you do. You will see more clearly how comments are put to use as you progress through the course.

Exercise 1.2 _____ (Formative: No Marks)

Practice Exercise: This exercise will not be marked.

Save a copy of the program you wrote in exercise 1.1 with a different name. Modify the code to print the following to separate lines on the screen: your name, favourite colour, favourite real number

¹⁰Appendix 1, *Dealing with Errors*, explains this and other techniques for debugging.

D Variables — declaring and assigning values

Before discussing how input and output works, including `System.out.println(...)` which you used in `Hello.java`, it will be helpful to cover the topic of ‘variables’. You will have come across the idea of variables in mathematics, and the concept of a variable in programming is similar.

The computer needs to store data in its memory which it can then apply operation to. The question is what types of data need to be stored in memory? The data might be something very simply such as a single integer number, or it might be something more complicated like a list of the marks for all the students in this module. For now we are going to assume that the data is in a simple form. Later we will generalise this idea to more complicated data.

A variable is associated with space in the computer’s memory that can hold a value (often a number). Each variable has a name, which you choose, and subsequently use to refer to the space in memory where the value of the variable is stored.

You must start by *declaring* the variable, this gives the variable a name and reserves some space in memory to store whatever value the variable takes. It also tells the compiler what kind of data you intend the variable to represent. This is known as the variable’s *type*. For example a variable which always takes integer values, can be *declared* as *type* `int` in Java. The line of code

```
int i;
```

declares the existence of an integer variable (i.e. a variable of ‘type’ `int`) which we have chosen to call `i`. As an `int`, it can store any integer value between -2147483648 and 2147483647. This is a fundamental difference between an integer in mathematics and an integer on a computer. The fact that numbers represented on a computer (or calculator) have finite size and precision has important consequences for the accuracy of calculations. You’ll see an example in a later exercise where you write a program to solve quadratic equations.

Java provides *types* (called ‘primitive data types’) to represent several kinds of number, e.g. integer and floating point, non-numerical things like text, and other more abstract things. These are listed on page 19. You can give a variable a longer name if you like, and it is usually a good idea to choose a word that explains what the variable is for. The convention is for variables to be named using lower case letters, or if the name consists of more than one word, that a capital be used at the start of each word other than the first. You may also use numbers or an underscore `_` in your variable names, but not at the beginning of the name. Examples of some well-chosen and valid variable names might be `total`, `maxValue`, `answer1`.

Exercise 1.3 _____ (Formative: 4 Marks)

This exercise will be marked by the robot.

Consider the simple program below. As it contains `public class VarTry` you must download it from Moodle and save it to a file called 'VarTry.java'.

Compile and run VarTry in the same way that you did with the Hello program to check it prints out the number nine. Edit the program so that the variable 'i' is given the value 11 instead of 9. Check again that the program now prints out the number 11. Submit the program to the MLE. Read the explanation that follows and try to understand how it works.

```
/**
 * Date:      3 October 2014
 * Exercise:  1.3
 * Version:   2.0
 * Purpose:   To demonstrate the declaration and simple uses of variables.
 *
 * @author Iain Bertram
 * @author Ian Bailey
 */

public class VarTry
{
    public static void main(String[] arguments)
    {
        int i;           // declares integer variable named i
        i = 9;           // gives i the value 9
        System.out.println(i); // print the value of i to the screen
    }
}
```

You can see here how each line of code, apart from comments, must end with a semicolon, `;`, (if you need to write a line of code that is longer than the width of the window you can continue it on the next line by omitting the semicolon until the end of the statement).

The line `i = 9` gives the value 9 to variable `i`. This is known as *assigning* a value to a variable — `i` is *assigned* the value 9. It means that the space in the computer's memory associated with `i` now holds this value. The first time a variable is assigned a value, it is said to be *initialised*. The `=` symbol is known as the *assignment operator*.

It is also possible to declare a variable and assign it a value in the same line, so instead of writing `int i` and then `i = 9` separately, you can write `int i = 9` all in one go. If you have more than one variable of the same type you can also declare them together e.g.

```
int i, j;
```

or

```
int i=1, j, k=2;
```

which can be a useful way of saving space. Where necessary you should add comments explaining the meaning of the variables, both so it is clear to you if you come to look at your program at a later date, and clear to those marking your programs.

A comment like:

```
int i, j;           // i and j are integers
```

is of no use. Anyone who is learning Java will already know what the line 'int i,j' means so you don't need to explain it. However, a comment like

```
int i, j;           // i and j are the number
                    // of neutrons and protons in an atom
```

is very useful as it gives information which cannot be understood by reading the code alone. Of course, in this case it would have been better to give the variables clearer names too.

You will of course want to use many quantities which are not integers, and there are several different variable *types* which cover these possibilities. For real numbers there are two possibilities — `float` and `double`. As `double` uses twice as much memory as `float` to store values, it can represent numbers with more significant figures. When representing real numbers in your program you will probably mostly want to use `double`. All the variable *types* are described below.

`byte` Integer variable allocated only 8 bits¹¹ (i.e 1 byte) of memory. May store values from -128 to 127: -2^7 to $(2^7 - 1)$.

e.g. `byte aByte = 44;`

`short` Short integer, allocated only 16 bits of memory. May take values from -32768 to 32767: -2^{15} to $(2^{15} - 1)$. e.g. `short aShort = -5000;`

`int` Most commonly used form of integer variable, allocated 32 bits. May hold any value in the range -2147483648 to 2147483647: -2^{31} to $(2^{31} - 1)$.

e.g. `int aInt = 3248883;`

`long` Used if working with particularly large integers, allocated 64 bits. Up to nineteen digits and a sign: -2^{63} to $(2^{63} - 1)$

e.g. `long aLong = -4457L;`

Note use of L to designate a number of type long (this must be used).

`float` *Floating point* real number, which must be written in the format 3.45, or 3.0e-5 — that is they must include a decimal place and may include the letter e. What follows e is the power of ten, so the two examples mean 3.45 and 3.0×10^{-5} respectively. A `float` variable is allocated 32 bits and holds a value between $\pm 1.4 \times 10^{-45}$ and $\pm 3.4 \times 10^{38}$, to eight significant figures.

e.g. `float aFloat = 5.6522f;`

Note use of f to designate a float

`double` A 16 significant figure, floating point, real number, allocated 64 bits. Values are written as for `float`, e.g. 5.8e59, and may take value between $\pm 4.9 \times 10^{-324}$ and $\pm 1.8 \times 10^{308}$.

e.g. `double aDouble = 5.65e25;` or `double aDouble = 2.48d;`

Note use of d to designate a double, on many systems a double is the default.

¹¹A *bit* is the smallest unit of computer memory. It may be thought of as either a 1 or a 0. 8 bits make up 1 *byte* of memory.

boolean May take one of only two possible values, *true* and *false*. This is a *logical truth* variable (1 bit).

e.g. `boolean aBoolean = true;`
or `false`

char A single character (this may be an upper or lower case letter, number or other keyboard symbols like `:`, `#` or `!` for example.)

e.g. `char aChar = 'c';`
characters should be surrounded by single quotes.

You can also create a **String** of characters. A `String` is like a word or a line of text and can include spaces, upper and lower case letters, numbers and other keyboard symbols. Strings are created in a similar way to the variables above, and the double quote symbol `"` is used to mark the start and end of the text. You may for example write

```
String name = "Bob1";
```

This creates a `String` called `name` which stores `'Bob1'`.

You should be aware that there are some words which you may not use as names for variables (or names for methods or classes for that matter) as they have a special meaning in java. These are:

`abstract`, `boolean`, `break`, `byte`, `byvalue`, `case`, `cast`, `catch`, `char`, `class`,
`const`, `continue`, `default`, `do`, `double`, `else`, `extends`, `false`, `final`, `finally`,
`float`, `for`, `future`, `generic`, `goto`, `if`, `implements`, `import`, `inner`, `instanceof`,
`int`, `interface`, `long`, `native`, `new`, `null`, `operator`, `outer`, `package`, `private`,
`protected`, `public`, `rest`, `return`, `short`, `static`, `strictfp`, `super`, `switch`,
`synchronized`, `this`, `throw`, `throws`, `transient`, `true`, `try`, `var`, `void`, `volatile`,
`while`, `widefp`.

Simple output

In the Hello program you used the line `System.out.println("Hello")` to write `'Hello'` to the screen. This is the standard way of outputting to the screen and it works for numbers and variables (see previous section) as well. If you want to print several things at the same time you can use the `+` symbol as follows to 'concatenate' the things together.

```
System.out.println("Hello" + 1.234);  
System.out.println("Hello " + "number " + 1.234);  
System.out.println("The value of variable x is: " + x);
```

The first two commands result in the outputs `'Hello1.234'` and `'Hello number 1.234'` respectively. The third command will print to the screen `'The value of variable x is: '` followed by whatever number, character, or string, is stored in the variable `x`.

You can also concatenate separate Strings together using `+`, e.g. `String name = "Bob" + " Smith"`. Or `String id = "abc" + 1.4` creates a string called `id` which stores `'abc1.4'`. Here `1.4` is initially a number but it is automatically converted into a string before being added to `"abc"`. We explore strings a little more in the next section.

Exercise 1.4 _____ (Formative: 4 Marks)

With reference to the section above, decide what type of variable would be most suitable for each of the following and think of a suitable name for each. Write this up in a text file in your working directory giving in each case a short explanation of your choice of variable type. You can write the text file in jedit or another 'plain text' editor such as notepad++.

Having done that, make a copy¹² of the program you submitted in exercise 1.3. Call the copy VarTry2.java, changing the class name to match. Edit VarTry2 to declare, initialise and print variables holding the following data:

- (i) your age;
- (ii) the number of molecules in a mole;
- (iii) your name;
- (iv) the age of the universe in seconds;
- (v) the first letter of your family name

Submit a zip file of your directory containing the program and the file containing your working to Moodle. This is marked by a person, so don't forget to make your code clear, with informative comments.

If you wish to print several things to the screen on the same line without using `+` all the time, you can use `System.out.print()` which works in the same way as the `System.out.println()` you've just been using, but will not move to the next line each time.

Arithmetic operators

Now that you are able to *initialise*¹³ and output data, it's time to do something with it in between. For numbers or numerical variables there are the operators `+`, `-`, `/`, and `*` which add, subtract (or reverse sign), multiply and divide respectively, just as in arithmetic. There is also `%` which gives the remainder when the first number is divided by the second, e.g. the value of `4 % 3` is 1.

There is also the *assignment* operator, `=`, which you used earlier. This does not act in the same way as an equals sign in maths, instead it puts the value which is to the right of the operator into the variable which is to the left. E.g. `x = 1` gives `x` the value 1, `x = y` gives `x` the same value as `y`, `x = 2*y + 1` gives `x` the value of `2y + 1`, and `x = x + 1` adds 1 to the value of `x`.

As in standard arithmetic, expressions within round brackets are evaluated first, then division, multiplication, addition and subtraction operations in that order — so you can add brackets where necessary to influence the order in which the expressions are evaluated. E.g. `2 + 4/2` returns the value 4 whereas `(2 + 4) / 2` returns the value 3. You will also need to use brackets sometimes in situations such as `x * (-y)`. Leaving the brackets out here would not only make the meaning unclear but also cause two operators to be placed next to each other which is not allowed.

¹²It is a good idea to save versions of your programs as they develop so that if the changes you make cause new errors you have a working previous version of your program to refer to. Remember to change the name of the public class so it matches the filename or it won't compile.

¹³Initialise — assign a value to a variable for the first time

Name	Meaning	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 - 0.1	33.9
*	Multiplication	30*30	900
/	Division	1.0/2.0	0.50
%	Remainder	20%3	2

Shorthand Operators

It is quite common to modify already existing variables rather than creating a completely new one. For example you may want to add 5 to the variable `x` using `x = x + 5`. Java allows you to combine the assignment operator (`=`) with the numeric operator (in this case `+`) to reduce complexity in your code.

Name	Meaning	Example	Equivalent
<code>+=</code>	Addition assignment	<code>i += 8</code>	<code>i = i + 8</code>
<code>-=</code>	Subtraction assignment	<code>i -= 8</code>	<code>i = i - 8</code>
<code>*=</code>	Multiplication assignment	<code>i *= 8</code>	<code>i = i * 8</code>
<code>/=</code>	Division assignment	<code>i /= 8</code>	<code>i = i / 8</code>
<code>%=</code>	Remainder assignment	<code>i %= 8</code>	<code>i = i % 8</code>

There are two additional operators that modify the value of a variable by 1. These are especially useful in loops (see section F) where you have to step through values sequentially. These operators are shown below where the variables `i` and `j` are considered to have been assigned the value 1 initially for the examples

Operator	Name	Meaning	Example
<code>++var</code>	preincrement	Increment var by 1 and return the new var value	<code>int j = ++i // j is 2 and i is 2</code>
<code>var++</code>	postincrement	Increment var by 1 and return the old var value	<code>int j = i++ // j is 1 and i is 2</code>
<code>--var</code>	predecrement	decrement var by 1 and return the new var value	<code>int j = --i // j is 0 and i is 0</code>
<code>var--</code>	postdecrement	Decrement var by 1 and return the old var value	<code>int j = i-- // j is 1 and i is 0</code>

Exercise 1.5 _____ (Formative: 4 Marks)

This exercise will be marked by the robot.

Copy the source file 'Division.java' from the course web site (Moodle). You don't need to type it in.

Add lines to this source file where indicated so that the program also prints out each of the following on a separate line

- the value of i minus j
- the product of i and j
- the value of i divided by j rounded to the lowest whole number
- the remainder of i divided by j
- the value of $((\text{double})i)/j$. This is explained below.

Submit your program to Moodle as a single .java file. The program will be automatically marked by a robot. The latter will compile your program, run it and analyse the output.

Make sure your program adheres strictly to the following rules:

- Your output values should be exactly in the order as they are specified in this exercise.
- Each line of output should contain one numeric value only and no additional text.
- Do not include *any* other text anywhere in your output.
- Do not delete any of the lines of code already in the program.

The robot can neither understand a human language nor is it capable of guessing your intention. Therefore make sure that your program does not print anything extra — this will confuse the robot and it will mark the results as incorrect. If in doubt then ask a demonstrator.

Please do *not* submit .class files. For this exercise there is also no need to 'zip' the files as there is only one of them. Please just submit the .java file.

Before you edit it the file `Division.java` will look like the listing below.

You do not have to type in this program. You can find it on Moodle.

```
/**
 * Exercise: 1.5
 * Purpose:  To demonstrate use of arithmetical operators and
 *           a common problem encountered with integer division.
 * @author Iain Bertram
 * @version 1.0
 * Date:
 */
public class Division
{

    public static void main(String[] args)
    {
        // declare and initialise variables
        int i = 5, j = 3;

        // print the numbers and their sum
        System.out.println("The values are: i is " + i +
                           ", j is " + j);
        System.out.println("The sum of these integers is " + (i + j) );

        // Add your lines below.  Don't edit anything above this line.
    }
}
```

One of the aims of this exercise is to demonstrate a potential difficulty with dividing one integer by another. If both the numbers involved are integers, an integer result is expected so the answer is rounded down to a whole number — thus $5/3$ gives 1. If using numbers in your program you could instead write $5.0/3$ or $5/3.0$ to get a decimal answer. The default type for a decimal number in your code is `double`, and when one number in the division is `double`, the result is of type `double` as well. Obviously if you are using a variable just adding `.0` is not much use, so instead there is a way of converting the value of an `int` variable into a `double` before the operation is carried out. This process is called ‘casting’. The piece of code `(double)i` that you have just used in exercise 1.5 ‘casts’ the integer `i` so that it is interpreted as a ‘double’ rather than as an ‘integer’.

E Input, mathematical functions and conditional branching

Keyboard input

It would obviously be much more useful to create a program which could add together any two numbers rather than just two specific numbers which are written in the program. Recompiling every time you wish to try a program with new values is not very satisfactory. An example of how to enter numbers via the keyboard which your program can use while it is running is given below. Input is slightly more complicated than output, so you are not expected to understand the details of the code at this stage.

```
/**
 * Purpose: Shows the lines which read input from the keyboard.
 * These include "import java.util.Scanner" in
 * addition to the lines indicated in the main method.
 * @author Iain Bertram
 * @version 2.0
 */
import java.util.Scanner;

public class InputExample
{
    public static void main(String[] argv) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a String with no spaces: ");
        String text = scanner.next();
        System.out.println("The entered String is: " + text);
        System.out.println();

    }
}
```

Just as in the example above, you will often find it useful to add a line like `System.out.println("Enter Input")` or similar whenever you use code like this so that it is obvious when the program is waiting for some input from the user. When running the code, enter the text and then press the return key to finish and let the program continue to execute.

Whenever you wish your program to read something from the keyboard, simply choose the most appropriate command from the following list:

```
Scanner sc = new Scanner(System.in);
String s = sc.next();           // Returns next "character string"
                                // without spaces
int i = sc.nextInt();           // Returns next integer value.
long l = sc.nextLong();         // Returns next long value
float f = sc.nextFloat();       // Returns next float value
double d = sc.nextDouble();     // Returns next double value.
boolean b = sc.nextBoolean();   // Return the next boolean value
```

This is the standard way to read data from the keyboard. By way of a brief explanation: `import` tells the compiler that you will be using an extra ‘package’ of software from outside your program. The name `java.util.Scanner` tells the compiler the name of the ‘package’ so that the compiler knows where to look for the necessary files.

There will be a chance to practice this input method in the next couple of exercises.

Exercise 1.6 _____ (Formative: No Marks)

Practice Exercise: This exercise will not be marked.

Download the program `IOintroduction`, compile and run it.

Modify the program so that you can enter an integer, a double, and a string. Print the three values entered to the screen.

Mathematical library

It may be necessary in a program to use common mathematical functions such as the sine, square root, or log of a number. There are some standard *methods* Java provides to do this. To calculate $y = x^2$ it is easiest to use the expression `y = x*x`, however for higher powers of `x` this becomes awkward, so in general $y = x^n$ is calculated using the expression `y = Math.pow(x, n)`.

The values or variables placed in the round brackets, `()`, of a method are known as *arguments*. In the case of the method `Math.pow` in the example above the arguments are `x` and `n`.

To access these ‘methods’ which represent mathematical operations and functions you may need to include the `java` package `Math` by including the statement `import java.lang.Math` before your `public class` statement e.g.:

```
import java.util.Scanner;
import java.lang.Math;

public class InputExample
```

The complete list of methods representing mathematical functions included in Java 8 can be found at <http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>.

On the next page is a short list of some of the most common.

```
Math.E           // e as a double
Math.PI          // pi as a double
Math.sin(x)      // sine of x
Math.cos(x)      // cosine of x
Math.tan(x)      // tangent of x
Math.asin(x)     // arcsine of x
Math.acos(x)     // arcsine of x
Math.atan(x)     // arctangent of x
Math.exp(x)      // exponential of x
Math.log(x)      // natural logarithm of x
Math.pow(x,n)    // x raised to the power n (both double)
Math.sqrt(x)     // square root of x
Math.random()    // a random double between 0.0 and 1.0
                 // (uniform distribution). It requires no argument
Math.abs(x)      // the absolute value of x. Works for int, long, float
                 // and double values, returning value of the same type
```

Except `Math.abs()` these mathematical functions all give result values of type `double`. They also all take variables or values of type `double` as their *arguments* — both `x` and `n` in the above are `double` variables. Note that `Math.pow()` requires two *arguments*, whereas `Math.random()` requires none, although the brackets must still be included. Using `Math.abs()` is slightly different — using it on a `float` variable will give a `float` value, on an `int` variable will return an `int` value, and so on.

Note that all the trigonometric functions work in radians.

Exercise 1.7 _____ (Formative: 4 Marks)

This exercise will be marked by the robot.

Write a program that reads in a number (x) of type `double` from the keyboard and then computes and writes out the values of the three following expressions:

- $\sin(x)$
- $\cos(x)$
- $\cos^2(x) + \sin^2(x)$

The value of $\cos^2(x) + \sin^2(x)$, should of course, be very close to 1.

Ensure that your code includes appropriate comments.

Submit your program to Moodle as a single `.java` file. The program will be automatically marked by the robot which will compile your program, run it, 'type' a number as input to your code and then analyse the output your code provides. You can use any name you like for the file as long as the file name matches the name of the class (obviously you should avoid special characters in the names as already mentioned earlier in the notes).

Make sure your program adheres to the following rules:

- The expected input should be a 'double' number corresponding to the value of x .
- The output values should be exactly in the order specified in this exercise.
- Each line of output should contain only one value.
- Don't include *any* other text in the output. E.g. don't have a prompt asking the user to input a value for ' x '. You would normally include a prompt, but in this case your user is a robot who cannot read...

Note that the robot can neither understand a human language nor guess your intention. Therefore make sure that your program does not print anything extra — this will confuse the robot and it will mark the results as incorrect.

Please submit only the `.java` file.

The `if` statement and comparison operators

We use `if` in a program so that a block of code may only be executed when a criteria, which we choose, has been satisfied. It is one way in which the *flow*, the order in which lines of code are executed when the program runs, can be controlled, (you may find it helpful to think of `if` and other forms of *flow control* by drawing a flow chart). A basic example of the code for an `if` statement is as follows:

```
if ( boo )
{
    x = y;
}
```

In this short piece of code, `x` is only set equal to `y` if whatever condition we have chosen to put in the round brackets `()` is true. This is where the boolean variable type briefly mentioned earlier may come in useful. If the value of a boolean variable `boo` is true, the code in the curly brackets `{ }` (in the above case `x = y`) is executed. If `boo` is false, the code in curly brackets `{ }` will **not** be executed.

Notice how within the curly brackets `{ }` code is indented. This makes no difference to the computer, but it makes it clearer to a human reader that the code is contained within an `if` statement ¹⁴.

We can write things like

```
if (w < z)
{
    x = y;
}
```

As you might expect `if (w < z)` means ‘if `w` is less than `z`’, and in the above code `x = y` only happens if it is true that `w` is less than `z`.

Less than, `<`, is an example of a *comparison operator* — it compares two variables, asking whether one is less than the other, returning a `boolean` result — *true* if the condition is satisfied and *false* if it is not. The other *comparison operators* are in the table below. Note the difference between `w == z` (i.e. ‘does `w` equal `z`?’) to `x = y`, i.e. ‘set the value of `x` equal to `y`’ which is the assignment operator we met earlier.

`<` Less than

E.g. `w < z` is true if the value of `w` is less than the value of `z`.

`>` Greater than

E.g. `w > z` is true if `w` is greater than `z`.

`==` Equal to

E.g. `w == z` is true if the values of `w` and `z` are equal.

`<=` Less than or equal to

E.g. `w <= z` is true if `w` is less than or equal to `z`.

`>=` Greater than or equal to

E.g. `w >= z` is true if `w` is greater than or equal to `z`.

¹⁴`jedit` should automatically indent your code to make it easier for someone to read

!= Not equal to

E.g. `w != z` is true if `w` and `z` have different values.

Here's a simple example where `if` is used to compute the modulus, i.e absolute value, of any double value entered.

```
/**
 * Purpose:  An application to demonstrate use of if
 */

import java.util.Scanner;

public class IfExample1
{
    public static void main(String[] args)
    {
        //Read in a value from the keyboard
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a positive or negative number...");
        double x = sc.nextDouble();

        if ( x < 0 )
        {
            x = -x;
        }

        System.out.println("The modulus of this number is " + x);
    }
}
```

Else and else if

There is also the option of adding `else` after an `if` — the `else` section should contain parts to be executed if the condition is *not* true. A trivial example of this is given below, where a value for `boo` is displayed on the screen.

```
/**
 * Purpose:  demonstrate how if ... else statements work
 */

public class IfExample2
{
    public static void main(String[] args)
    {
        boolean boo;

        // some code goes here which gives boo the value true or false
        boo = true;

        // Switch boo from true to false or false to true
        if ( boo )
        {
            System.out.println("if: boo is true");
            boo = !boo;
        }
        else
        {
            System.out.println("else: boo is false");
            boo = !boo;
        }

        // print value of boo to the screen directly
        System.out.println("boo is now " + boo);
    }
}
```

If `boo` is *true* it is changed to *false* otherwise the `else` part is executed and `boo`, which must be *false* is changed to *true*.

There is also an optional `else if` that can be added. This works like an `else` but you get to specify another test condition as well. Here is an example:

```
import java.util.Scanner;

/**
 * Purpose: An example of using 'If' and 'Else If'
 *
 */
public class IfExample3 {
    public static void main (String[] args){

        // Read in a number
        Scanner scanner = new Scanner(System.in);
        System.out.print ("Enter a number between 0 and 10 inclusive: ");
        double x = scanner.nextDouble();

        // check user input is in the required range
        if (x > 10) {
            System.out.println("The number you entered is too high");
        }
        else if (x < 0) {
            System.out.println("The number you entered is too low");
        }
        else {
            System.out.println("The number you entered is " + x);
            System.out.println("This makes me happy");
        }
    }
}
```

This program checks whether input is in the range 0-10. If the input is larger than 10 the first condition is *true* and the `if` block of code is executed. If `x` is less than 10 the second test is performed — `x<0`. If the second condition is true the `elseif` block of code is executed. Otherwise, if both conditions were *false* the final `else` block of code is executed.

You may use as many `else if`'s as you like after an `if`, allowing tests of many different conditions such as `x<0`. If an `else` is used it always comes at the end, being executed only if all of the conditions preceding it are found to be *false*.

Logical operators

It is also possible to test whether more than one condition is satisfied at the same time. E.g.

```
if ((w < z) && (a == b))  
{  
    x = y;  
}
```

This uses what is known as the *AND logical operator*, which is represented in the code above by the symbol `&&`. It does exactly what the name suggests — the code will set `x` equal to `y` only if **both** `w` is less than `z` **and** `a` is equal to `b`. Look carefully at the use of round brackets in the example above. These are important as the conditions within the innermost brackets are evaluated first. When these are found to be either *true* or *false*, the `&&` operator checks whether both sets of brackets are *true*.

There are also other logical operators, including OR and NOT which again do what their names suggest:

&& AND operator

E.g. `(boo1 && boo2)` returns *true* only if both `boo1` *and* `boo2` are *true*. Returns *false* otherwise.

|| OR operator

E.g. `(boo1 || boo2)` returns *true* if either `boo1`, `boo2`, or both are *true*. Returns *false* otherwise.

! NOT operator

E.g. `!boo` returns *false* if `boo` is *true* and vice versa.

Exercise 2.1 _____ (Formative: 4 Marks)

Look at the code given below. What is the output from this program when:

- (i) a = 6, b = 2, c = 2, and d = 3?
- (ii) a = 4, b = -1, c = 4, and d = 4?
- (iii) a = 9, b = 7, c = 3, and d = 1?
- (iv) a = -2, b = 3, c = -2, and d = 3?

- First, work it out by hand.
- Secondly write a program to check your answers. (You do not need to submit this to Moodle.)
- Finally, enter your answers on Moodle

```
String text;  
if ((a>b) && (c!=d))  
{  
    text="First if block of code executed";  
}  
else  
{  
    text="First else block of code executed";  
}  
System.out.println(text);  
if ((a==c) || (b<=d))  
{  
    text="Second if block of code executed";  
}  
else  
{  
    text="Second else block of code executed";  
}  
System.out.println(text);
```

Flow-Charts

One of the most important aspects of computer programming is the design of the program. A more complete discussion of code design will take place in the sections devoted to 'methods' and 'classes'. For now, we will look at the use of one of the tools used in design: flow-charts.

A flowchart (also spelled flow-chart and flow chart) is a schematic representation of a process. In addition to their use in computing they can be used in presentations to help visualise processes. In the case of


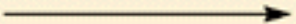

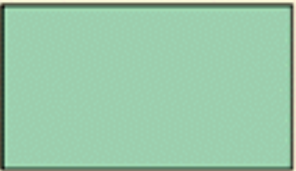
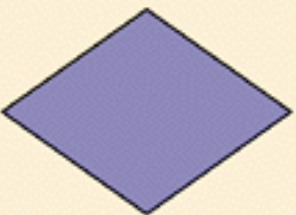
Name	Symbol	Use in flowchart
Oval		Denotes the beginning or end of a program.
Flow line		Denotes the direction of logic flow in a program.
Parallelogram		Denotes either an input operation (e.g., INPUT) or an output operation (e.g, PRINT).
Rectangle		Denotes a process to be carried out (e.g., an addition).
Diamond		Denotes a decision (or branch) to be made. The program should continue along one of two routes (e.g., IF/THEN/ELSE).

Figure E.3: *Some common symbols used in flowcharts.*

programming they are particularly useful with decisions (`if`) and iterations (`for`, `do`, `while`) which we introduce shortly.

The basic elements of a flowchart are illustrated in Fig. E.3. These elements are:

- **Oval**:- denoting the beginning or end of a program;
- **arrow**:- A flow line that shows the direction of logic in the diagram;
- **parallelogram**:- Denotes an input or output operation;
- **rectangle**:- the process to be carried out;
- **diamond**:- decision or branch to be made.

In some of the following exercises you may be required to produce flowcharts as part of the design of your programs. They will also be used to illustrate many of the processes used in this course.

Figure E.4 illustrates the procedure you might follow in attempting to fix a lamp. It starts with an oval representing the starting point, a lamp that does not work. The arrows show the direction you take through the process of repairing the lamp. In this case the first choice you have is to see if the lamp is plugged in and you are offered two choices on what to do (if it is not plugged in, plug it in...). Subsequent choices and decisions are then followed until you reach the end of the process.

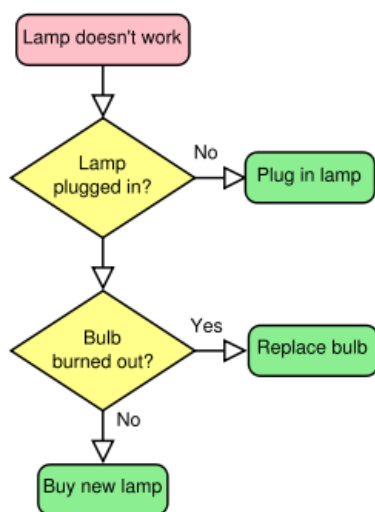


Figure E.4: A flowchart illustrating the process by which a lamp is repaired by someone lacking electrical skills.

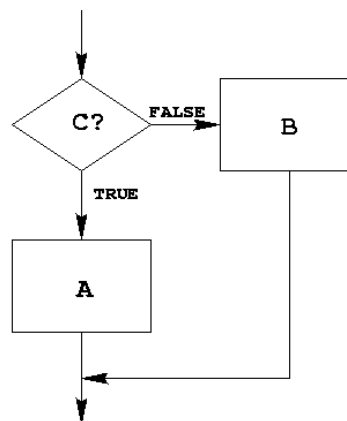


Figure E.5: A generic flowchart illustrating the `if` process. The statement `C` is evaluated and depending on the outcome either `A` or `B` is implemented.

Figure E.5 depicts the standard flowchart elements used in `if` statements where you make a single `if/else` statement while Figure E.6 shows how to implement multiple `if/else-if` statements. Don't worry, things will get more exciting later.

Practice Exercise: This exercise will not be marked.

Exercise 2.2 _____ (Formative: No marks)

If you haven't drawn flow charts before then draw a flow chart using the following elements to describe the process of making one move in a game of snakes and ladders.

- End
 - Climb up ladder
 - Landed on a snake's head?
 - Throw die.
 - Start
 - At bottom of ladder?
 - Move the counter by value shown on die.
 - Slide down the snake.
-

Exercise 2.3 _____ (Formative: 8 Marks)

This exercise will be marked by the robot.

Note that this exercise is worth more marks than the other exercises this week. Your goal is to create an application to solve quadratic equations.

- Assuming the form of the equation is $ax^2 + bx + c$, read in the three numbers a, b, c using a scanner (NB in your code create only one scanner and use it three times. Do **not** create three separate scanners. The latter may seem to work but it will cause lots of problems for the robot.)
- The input should be three doubles corresponding to the values of a, b , and c in this order. Do not print any text to the screen requesting input, as this program is for the robot not a human!
- If the equation has complex roots, your program should simply output “0” (zero) to the screen and nothing else.
- I.e. In the case of complex roots the output should be simply:
0
- If the equation has real roots, the program should print “2” to the screen followed by the two real roots (**even** in the case that it’s the same root repeated).
- I.e. in the case of real roots with values 3.1415 and 5.4567 the output should be :
2
3.1415
5.4567
- Do not include any other text in the output as it’s likely the robot will get confused.
- Test your code to see if it gives accurate results for the examples given below.
- Next, read the text below on “catastrophic cancellation”. Implement the revised quadratic formula and test it.
- The final version of your code (the one you are going to submit to Moodle) should be capable of giving accurate values for the roots of all the examples below.
- Ensure that your code is clear and contains appropriate comments as a human may check this.
- Finally submit your program to Moodle as a single .java file with no other files.

Your program will be tested using the following equations:

- (i) $x^2 - 4x + 4 = 0$
- (ii) $x^2 - 7x + 4 = 0$
- (iii) $2x^2 + 11x + 24 = 0$
- (iv) $x^2 - 1.786737601482363x + 2.054360090947453 \times 10^{-8} = 0$

Note: The standard solutions to the quadratic equation are not ideal for implementation in computer programs. There is the possibility of “catastrophic cancellation”¹⁵ between the terms in b and $\sqrt{b^2 - 4ac}$ which can result in the program giving incorrect solutions. This can be avoided by calculating the solutions x_1 and x_2 using the expressions:

$$q = -\frac{1}{2} \left(b + \text{sgn}(b) \sqrt{b^2 - 4ac} \right) \quad (1)$$

$$x_1 = q/a \quad (2)$$

$$x_2 = c/q \quad (3)$$

where $\text{sgn}(b)$ just signifies the sign of b . If you look at the information about the Maths package at <http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html> you should be able to find a suitable Java *method* to calculate the sign of a variable of type `double`.

Whilst this type of solution solves the problem of loss of accuracy when subtracting b and $\sqrt{b^2 - 4ac}$ it still leaves the possibility of accuracy issues in the expression $b^2 - 4ac$ itself. These issues can be addressed by using higher precision (e.g. 128-bit rather than 64-bit) variables. If you have some experience coding and you want an extra exercise to do at this point that you can explore the java ‘BigDecimal’ class which allows you to configure the precision being used in calculations. If you find anything interesting then upload examples to the discussion forum on Moodle.

¹⁵This occurs when you take the difference between two large numbers which results in a truncation in the answer and a resultant loss of accuracy.

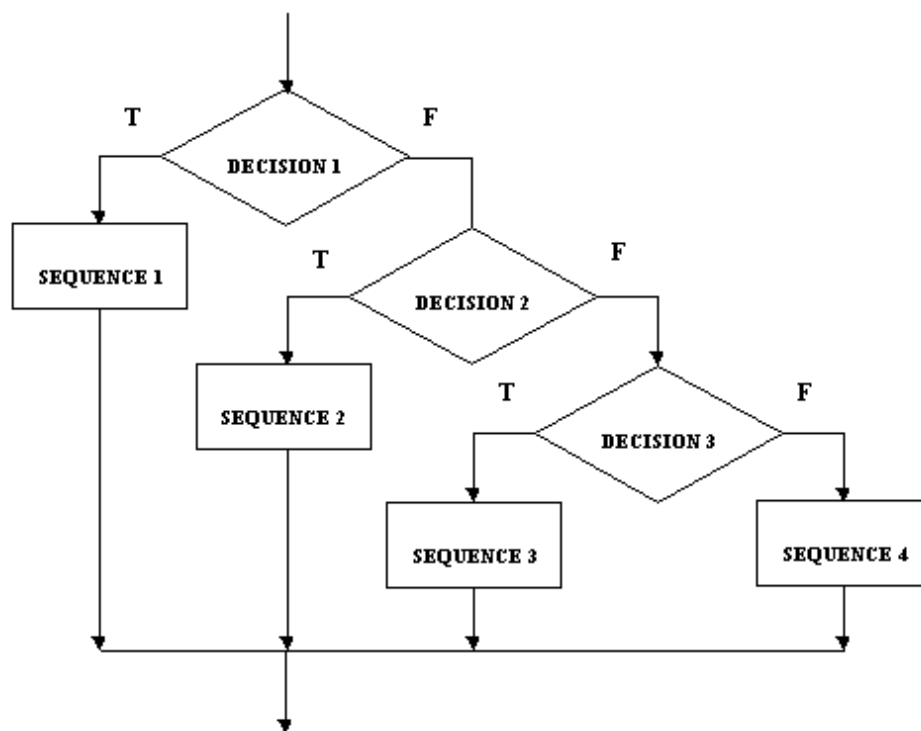


Figure E.6: An example of a flowchart representing an if/else if/else programming element.

F Iteration

One of the most common and powerful ways to control the flow of execution through a program is by using iteration, also known as *loops*. A *loop* is just a way of telling the computer to repeat some code many times. The number of iterations can be a predetermined number, or it can depend on the outcome of calculations made within the *loop*.

There are several types of *loop* available in Java. The first to be introduced is called a `for` loop.

For loops

Here is an example of a `for` loop:

```
for ( int i=0; i<9; i++)  
{  
    System.out.println("This is iteration "+i);  
}
```

The first line, starting with the `for` statement, defines the loop. The code to be executed inside the loop is contained within the curly brackets, `{}`. Code within these brackets is indented to make it easier to see that it is inside the loop.

There are three expressions within the round brackets after the `for` statement. The first (`int i=0`) is known as the *initial expression*. It is used to do things like declare and initialise variables before the *loop* starts. It is evaluated only once. The second expression within the round brackets (`i<9`) is the *test expression*. This is evaluated at the beginning of each iteration. It must be a *logical expression*, as described in the previous section on `if` statements. If it is found to be *true*, the iteration continues and the code within the *loop* is executed. If false, the *loop* ends. The final expression within the round brackets (`i++`) is the *update expression*. This is evaluated at the end of each iteration, after the code within the *loop* has been executed.

In the example, `i` is declared as an ‘`int`’ and initialised to 0. Then the test `i<9` is evaluated. This is clearly *true* since `i` is 0. So the line of code inside the *loop* is executed with the result that the value of `i`, 0, is printed to the screen. The *update expression* `i++` is short-hand for `i = i + 1`, i.e. the value of `i` is increased by one. This is repeated until it is no longer *true* that `i` is less than 9. At this point, the test fails and the *loop* ends.

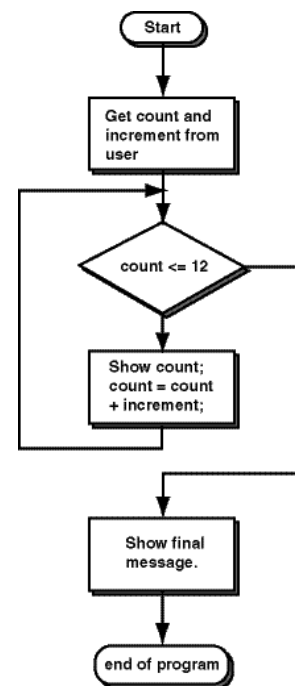


Figure F.7: Flowchart of a `for` loop.

Exercise 2.4 _____ (Formative: 4 Marks)

Answer online on Moodle.

- Predict exactly what numbers the code (quoted again below for convenience) would print out.
- Specifically, what are the first and last numbers that will be printed?
- If you are new to programming then you might want to sketch out a flowchart to help you understand what is going on.
- Answer the associated 'quiz' question on Moodle.

Here is the example of a `for` loop again:

```
for (int i=0; i<9; i++)  
{  
    System.out.println("This is iteration "+i);  
}
```

Note: the `++` operator is very useful in *loops*. There is also a `--` operator, which does the opposite, i.e. subtracts 1 from the variable.

You can put any valid Java code inside the `for` loop. For example, variable declarations, `if` statements, calculations, mathematical functions, and even more *loops*.

While loops

Below is an example of a `while` loop. Like the `for` loop, it contains a *test expression* (`i < 9`) and some code inside the *loop* to be repeated as long as this expression remains *true*. The variable `i` must be declared and initialised before the loop as you see on the first line, `int i = 0;`. The example below does exactly the same thing as the `for` loop above.

Example:

```
int i = 0;
while ( i < 9 )
{
    System.out.println("This is iteration "+i);
    i++;
}
```

The following example shows another simple use of the `while` loop. For each *loop* iteration, `x` is doubled and the new value is printed out. This continues until `i` exceeds 20. Hence powers of two are printed up to 1,048,576.

```
// Print powers of 2 up to 20th power.
float x = 1;
int i = 1;
while ( i <= 20 )
{
    x = x*2;
    System.out.println("This is the value of x: "+x);
    i++;
}
```

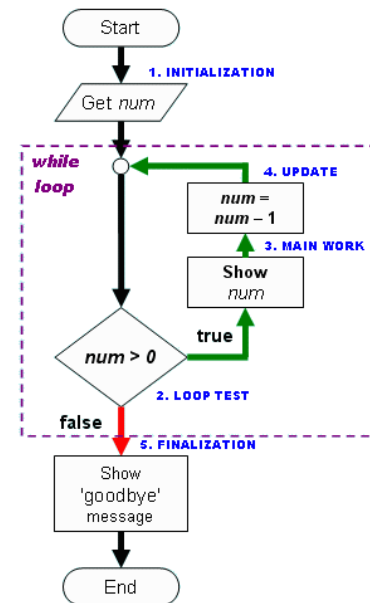


Figure F.8: Flowchart of a *while* loop.

So, which sort of loop should you use — `for` or `while`? They are both very flexible, so in almost any situation, either could be used to create the desired effect. The choice is therefore mainly a question of style and convenience. As a rough guide, `for` is well-suited to *loops* which have a definite number of iterations that you know in advance, as in the example above. It is also most appropriate when you need a variable inside the *loop* which keeps count of the number of iterations, like `i` in the above examples. A `while` loop is best when you don't know how many iterations there will be, but you can write a *logical expression* which tells you whether to continue for another iteration or not. If in doubt, a good rule is to use which ever type of *loop* seems simplest.

There is a third *loop* statement, `do`, which is very similar to `while`. Again, the example does the same thing as the two previous examples.

```
int i = 0;
do
{
    System.out.println(" The value if i is " + i);
    i++;
}
while (i < 10);
```

The only difference between `do` and `while` is that the *test expression* (`i<10`) is evaluated at the end of each iteration, instead of at the beginning. Therefore the code inside the curly brackets `{}` is always executed at least once.

Note: it is possible to create an *infinite loop*, i.e. one which repeats for ever. A simple example would be:

```
while (true)
{
    // never ends
}
```

If you have such a *loop* in your program, with no other means of escape the program will run for ever. If this happens, you can usually stop the program by pressing Ctrl-C.

Exercise 2.5 _____ (Formative: 4 Marks)

This exercise will be marked by the robot.

Take the example `IfExample3` which you can download from the course web site: Moodle. Modify `IfExample3` so the user is asked to re-enter a number if the input is not valid, i.e. outside the requested range.

- To aid in this you may want to draw a flowchart for your program.
- Do NOT modify or add any print statements in the program. An example of the expected output is given below.
- The only changes you should make are to add an appropriate loop around the input statements in the program and modify the if statements.
- Submit your working .java file to Moodle.

Example Output:

```
Enter a number between 0 and 10 inclusive: 25
The number you entered is too high
Enter a number between 0 and 10 inclusive: -100
The number you entered is too low
Enter a number between 0 and 10 inclusive: 5
The number you entered is 5.0
This makes me happy
```

Flow control

There will sometimes be points in the code inside the *loop* where it will be convenient to jump out of the *loop* completely, or skip the rest of the *loop* code and proceed directly to the next iteration. In Java, there are commands to do this. They are, respectively, `break` and `continue`. They allow finer control of the program flow, as the *loop* can be ended without waiting until the next time the *test expression* is evaluated. You should use these commands when it makes your code simpler or easier to understand. They are most useful when your programs become larger and more complex. The example below shows how they are used.

```
// generate random numbers and sum those that are <= 0.5,  
// until the running total exceeds 6.0.  
double sum = 0;  
int i = 0;  
while (true) // infinite loop, but break will be used to escape  
{  
    // get a new random number  
    double r = math.random();  
  
    // if r is outside the required range,  
    // skip to the start of the next iteration  
    if (r > 0.5)  
    {  
        continue;  
    }  
  
    // do something with the random number  
    sum = sum + r;  
    System.out.println("random number " + r + " running total " + sum);  
    i++;  
  
    // exit the loop when the running total exceeds the target  
    if (sum > 6.0)  
    {  
        break;  
    }  
}
```

The program is illustrated in the flowchart in figure F.9.

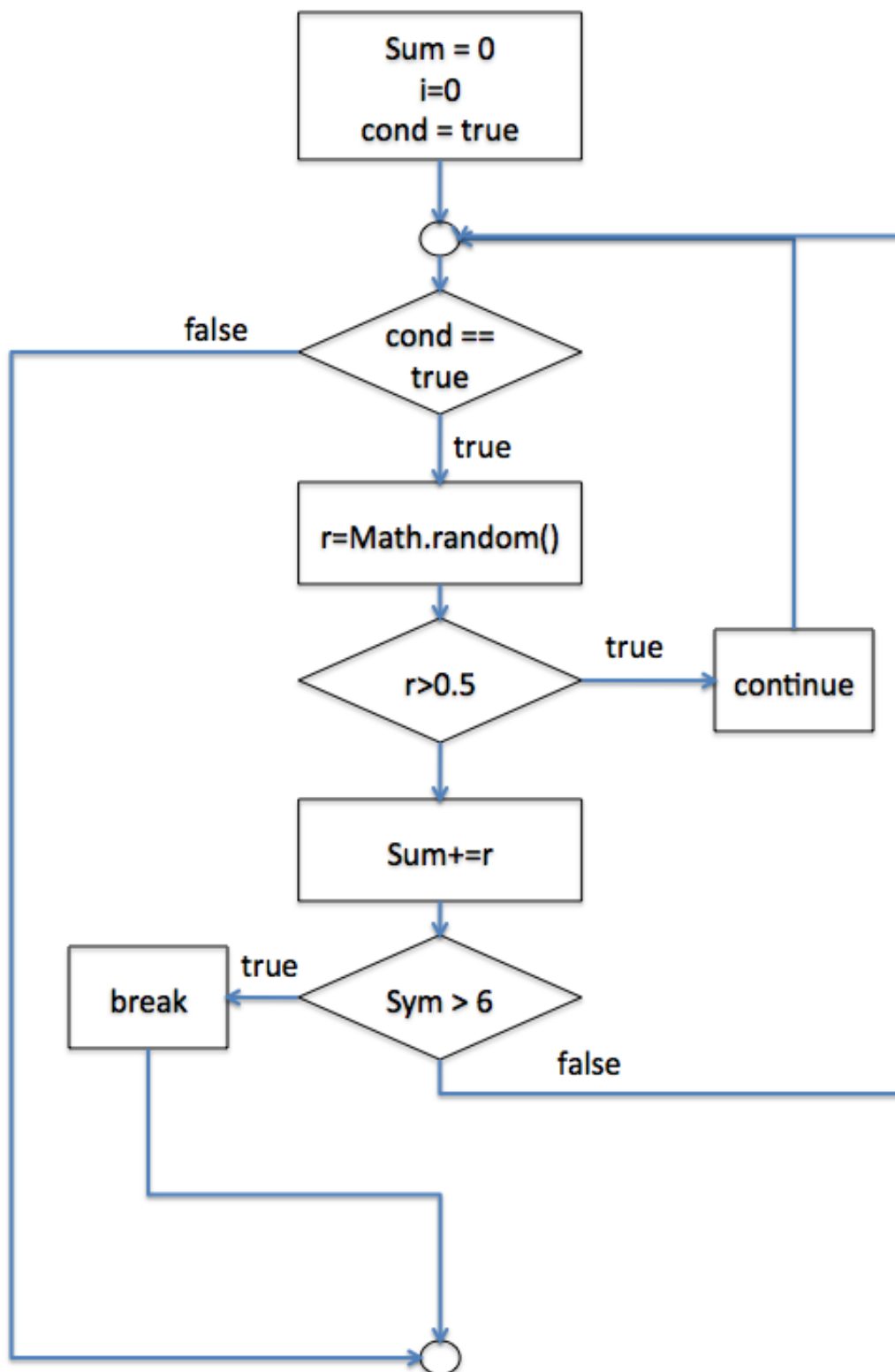


Figure F.9: Flowchart using *break* and *continue* statements.

Scope

Now that you have learnt about the main control statements `if`, `for`, `while`, you need to know about something called *scope*. The *scope* of a variable is the region of the program within which the variable can be referred to. A variable will only exist from the point at which it is declared until the end of the block of code it is in. A block of code is contained within curly brackets. If you declare a variable inside a *loop*, it will not be available outside the *loop*. At the end of the braces, the variable “goes out of scope” which is to say that it is no longer recognised by the compiler or available to use in your program. Study the examples below:

```
// Scope example 1 - this won't compile because of the last line.
// the scope of i is inside the loop
for (int i=0; i<10; i++)
{
    System.out.println("Intermediate i is " + i);
}
System.out.println("final i=" + i); // wrong, outside the scope of i
```

```
// Scope example 2 - this is a corrected version of scope example 1
// the scope of i is outside the loop, because i is declared before
// the for loop.
int i;
for (i=0; i<10; i++)
{
    System.out.println("Intermediate i is " + i);
}
System.out.println("final i=" + i);
```

```
// Scope example 3 - this won't compile either
boolean test = true;
if (test)
{
    boolean success = true;
    System.out.println("Test was " + success); // this is ok
}
System.out.println(success); // wrong - outside the scope of 'success'
```

If the third example is modified so that the boolean variable ‘success’ is declared before the `if` statement, then it will work.

In general it is a good idea to declare variables within the most limited scope that they need, and to declare them as near as possible to where they are going to be used. This means if you use the variable by mistake elsewhere, you are more likely to get a compiler error rather than a strange run-time behaviour, which is hard to debug.

So, if a variable is only needed inside a *loop*, it is best to declare it inside the *loop*. For example, if you like to use `i` as the iterator in `for` loops, you can declare it in the *initial expression* of the `for` loop, as shown in the example at the beginning of this section. Since `i` is now limited to the scope of the *loop*, you can declare it again in the next *loop* and be sure you are getting a different variable every time.

In addition to variables which are just used temporarily by the program, it is often the case that at the top of your main method you will declare some variables which are used throughout your program and/or have an impact on how the code operates. For example, if your code reads data from a file then you might want to store the name of this input file in a variable. Declaring and initialising this variable at the top of the main method makes it easy to locate and change it when editing the code.

Exercise 2.6 _____ (Formative: 4 Marks)

This exercise will be marked by the robot.

The following code extracts have scope-related errors.

- Attempt to compile the code and use the error messages to help you figure out what the problem is in each case.
- You do not need to type in the extracts as the full examples can be found on Moodle.
- Fix any scope problems with the examples, and upload them to Moodle.
- Fixing the problem should not involve moving the print statements.

```
// Example 1
// Calculate n factorial for some integer n. Read n from the keyboard.

Scanner keyboard=new Scanner(System.in);
System.out.println("Enter an integer.");
int n = keyboard.nextInt();

for (long i=1L, factorial=1L; i<=n; i++)
{
    factorial*=i;
}

System.out.println(n+"! = "+factorial);
```

The second example is given on the next page.

```
// Example 2
// Find the first n prime numbers. Read n from the keyboard.

Scanner keyboard=new Scanner(System.in);
System.out.println("Enter the number of prime numbers to calculate:");
int n=keyboard.nextInt();

System.out.println("\n Calculating the first "+n+" prime numbers:");

for (int num=2, nprime=0; nprime<n; num++)// loop over numbers
{
    boolean isPrime=true;
    // Check if the number is prime by looking for factors
    for (int ifact=2; ifact <= Math.sqrt(num) ; ifact++)
    {
        if (num % ifact == 0 ) {isPrime=false;}
    }
    if (isPrime) // A prime number has been found, so print it.
    {
        nprime++;
        System.out.println(num);
    }
}

System.out.println("\n The first "+nprime+
    " prime numbers have been calculated!");
```

G Multiple methods

Basics of using another method

Every programming language has different ways of partitioning up lines of code. In Java we will be using *classes* and *methods*. So far all the applications you have written have had the same basic structure. First your program must contain a `public class` for which you can pick any name, the contents of which are inside a set of curly brackets. Secondly, inside the class you've included a `public static void` *method* which must be called `main`, and must have the *argument* `String[] argv`¹⁶; there should be only one such *method* in each program, as the Java system looks for this as the first thing it runs. As with all *methods*, the contents of the `main method` are contained within a set of curly brackets `{ }`¹⁷. The basic structure is shown below.

```
/** * Purpose: To show the most simple Java program structure. */
public class Simple
{
    public static void main(String[] argv)
    {
        //some program code goes here
    }
}
```

We are now going to add to this program structure a second *method*. See that this second *method* is still inside the `class`, but unlike all the other application code you have made use of so far, it is outside of the `main method`.

¹⁶The name 'argv' is arbitrary.

¹⁷At the end of the last section we looked at the idea of *scope*, so you're already aware that variables defined within a set of curly brackets are not defined outside of the curly brackets. There will be more on this later.

```

/**
 * Purpose: To show structure of program with multiple methods.
 * The class can have any name you like but it conventionally
 * starts with a capital letter.
 */
public class LessSimple
{
    // Let's have another method in this class -
    // it can be called whatever you like,
    // but it's conventional to start the name with
    // a lower case letter.

    public static void doSomething() {

        // The code should go in here in the usual way -
        //e.g. variable declarations, ifs, loops etc.
        // It must end with a return statement...

        return ;
    }
    // Now let's have the main method as usual
    //(it can be above or below the other method)

    public static void main(String[] argv) {

        //some program code goes here
        //We should use the doSomething() method
        //this is done by a line like the one below

        doSomething ();
    }
}

```

The lines `doSomething()` and `return` will be explained shortly.

When the application is run, the *main method* is executed. Any other methods that exist will only be executed if they are used by the *main method*. The *main method* can be thought of as the top level of the program, from which other methods are invoked. It is good practice to keep *main* very short and simple by splitting your program up into different methods which can be called from *main*¹⁸.

The usual convention is to have the *main method* written as the last *method* in your source file, just so that it is easy to find. However, the order in which *methods* are written doesn't affect the order in which they are executed — even if it is at the end, the computer always begins by executing the *main method*.

Note that the additional *method* 'doSomething' is *declared* using `public static void` just as for the *main method*. Later we'll come back to discuss what `public static void` actually means.

¹⁸Those of you with programming experience may be aware of *Java applets*. In this special case you do not provide a *main* method because one is already provided elsewhere that calls `paint()` and the other methods that you provide in your applet code.

Here is a simple example application with exactly the same structure as above.

```
/**
 * Purpose: Prints the squares of the first 10 integers,
 *          as an example to show the use of two methods.
 */
public class Trivial
{
    // method to print squares of integers
    public static void squares()
    {
        int maxNum=10;
        for(int i = 1; i <= maxNum; i++) {
            System.out.println(i*i);
        }
        return;
    }
    // main method
    public static void main(String[] argv)
    {
        squares();        // calculate squares of integers
    }
}
```

The line of code `squares()`; is where the *main method* makes use of the *squares method*, asking it to carry out the set of instructions in the *squares method*. This is known as a *call* to the *squares method*. The `return` statement at the end of *squares* sends flow of the program back to the *method* that called it, and so execution of the *main method* continues.

You will soon find that writing programs where all your code is within the *main method* can become very long, which makes the source file look messy or unclear. It can also be awkward if you have similar groups of calculations or instructions that need to be performed and need to be typed in separately each time. Using additional *methods* can help with both of these issues, making code tidier and therefore easier to understand, and reducing the need to duplicate code. There is little obvious advantage in having separated the above program into two *methods* — the same could very easily be achieved with just a *main method* — it is just a simple example in how to write and call *methods*. However with larger programs there are significant benefits of separating your code into several *methods* in this way.

Passing information between methods

What if you want the behaviour of a *method* to vary depending on a value of a certain variable in your *main method*? It is possible to *pass* several values to *method*, and receive a value back. You may for example have several numbers you would like some calculation performed on. You could pass these values to a *method* which carries out the calculation and then returns the result to you. This should be familiar from when the earlier discussion of *methods* in the `Math` package, such as `Math.pow()`.

Below we show how to give, or *pass*, a value **to** a *method*. This is done by giving the *method* an *argument*, which goes between the round brackets `()` after the *method* name. Look at the following example, an explanation follows.

```

/**
 * Purpose: Example of program with two methods.
 *          The main method uses a method called printSquare.
 *          printSquare takes an 'argument' of type 'double'
 *          and prints the value of the argument squared
 *          to the screen.
 */
public class ArgExample
{
    // method to print square of a value to screen
    public static void printSquare(double y)
    {
        System.out.println(y*y);
        return;
    }

    // main method
    public static void main(String[] argv)
    {
        double x = 5.0;

        printSquare(3.0); // call to print the square of 3.0
        printSquare(x);   // call to print the square of x
    }
}

```

The *method* `printSquare` is *declared* as `public static void printSquare(double y)`. Unlike before, the brackets that follow the *method* name now contain a variable declaration, in this case for a variable `y` which is of *type* `double`. The variable `y` exists within this method and can be used here for calculations etc...

As the variable declaration for `y` is in the round brackets, whenever the *method* `printSquare` is called the *method* expects a `double` value to be in the round brackets in the call statement. So the call for `printSquare` may look like:

```
printSquare(3.0);
```

In the `printSquare` *method* `y` then takes this value, for the case above 3.0, so the value 9.0 is printed to the screen.

Within the *main method* a variable `x` is declared and given a value. When the call `printSquare(x)` is used, `y` takes whatever value `x` has in the *main method* at the time, in this case 5.0, so 25.0 is printed to the screen. There's nothing special about naming the variables `x` and `y` of course; you can name them anything you want. It is possible for a *method* to have several arguments. These should be separated by commas, e.g.


```

public static void calculate(double x, double y, int i, short n)
{
    //method code goes here
    return;
}

```

then to call the *method* simply enter values, or variables, of the expected *type*, e.g.:

```

calculate(3.0, result, 2, num) // where result and num are
                               // double and short variables respectively

```

So far all the additional *methods* you have seen have been declared as `void` and have not passed back, or *returned*, any values to the *method* that called them. This is the meaning of `void`, that the calling *method* should not expect any result to be passed back to it from the *method* being called. However *methods* can also be declared as `double`, `int`, `boolean`, `String`, or any other variable *type*, in which case the code for the *method* must return a value of that *type*. This is probably easiest to understand from an example. See below.

```

import java.util.Scanner;

public class NonVoidMethodExample
{
    /* First lets have a method that represents some mathematical
       function. */

    public static double function1(double x)
    {
        /* write some mathematical function of x here, e.g.  $x + 3x^2$  */
        return (x + 3*x*x);
    }

    // ...and then let's call the above method from the main method

    public static void main(String[] args)
    {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a value of x to evaluate f(x)");
        System.out.print("or enter a non-number to exit: ");
        // read input until a line without a double is entered
        while ( scanner.hasNextDouble()) // Is next value 'double'?
        {
            double x = scanner.nextDouble();
            System.out.println("f(" +x +") =" + function1(x));
        }
    }
}

```

The above example defines a *method* called `function1` which has the return *type* `double` and also takes

a `double` argument. The method simply performs a calculation using the argument and returns the result.

The use of this *method* is demonstrated in the `main method`. In this example, the user is prompted to enter a number, then there is a loop which takes each entered number, uses the `function1 method` to calculate another number, and prints them. The program ends when the user enters nothing.

Notice that the call to a `void method` exists on its own on a line. In contrast, in the example above, there is a value returned from the *method*, so this needs to be used somehow in a calculation or stored in a variable.

Other information about methods

A *method* can not alter the values of any of its arguments. Thus the following will compile but does not change the value of the variable being passed to the method through the argument list.

```
import java.io.*;
/**
 * Purpose:  An example of attempting to change the actual parameter of
 *           a method
 * @author Ian Bailey
 * @version 1.0
 */
public class ArgChange
{
    /* A method which increments its formal parameter and
    * then returns the value */
    public static int increment(int i)
    {
        i+=3;
        return i;
    }

    public static void main(String [] arv)
    {
        int j = 3;
        int k = 0;

        System.out.println(j);

        k=increment(j); // attempt to increment j

        System.out.println(j);
        System.out.println(k);
        return;
    }
}
```

In this example the value of 'j' is left unchanged whereas the value of 'k' printed out is 6.

You do not have to restrict yourself to calling additional *methods* just from the *main method*. If you have two or more additional *methods* one may call another.

Exercise 3.1 _____ (Formative: 20 Marks)

This exercise will be marked by the robot.

In this exercise you will start by writing a class called `MultMethEx1` which uses a *method* called `printTriangleArea` to calculate the area of a triangle, given the lengths of the three sides as arguments.¹⁹ It should print out the result.

Start by writing a *main method* which requires the user to enter three values for the sides of a triangle and then uses the `printTriangleArea` *method* to calculate and print the area.

Your final program should not prompt the user to enter values. It should only print the area to the screen. For example, if you enter

```
3.0
4.0
5.0
```

the code should only print

```
6.0
```

If the values entered do not correspond to a valid triangle then your program should print:

```
The values entered do not form a triangle.
```

Test your code by calculating the areas of several test triangles. When devising the lengths of the sides of your test triangles, bear in mind the constraints on the lengths of the sides of the triangle²⁰

Read Carefully

- (i) Copy your version of `MultMethEx1` to create a new class called `MultMethEx2` and edit it as described below.
- (ii) Add a new *method* called `calcTriangleArea` that performs the same calculation as `printTriangleArea` did, but instead of printing the area to the screen, it returns the area as a double.
- (iii) Modify the *main method* to use both `calcTriangleArea` and `printTriangleArea` to show that they produce the same result as each other.
- (iv) Change the code in the *method* `printTriangleArea` so that it now uses the new *method* `calcTriangleArea` instead of calculating the area itself. I.e. at this point, *main* should call `printTriangleArea` and `printTriangleArea` should call `calcTriangleArea`.
- (v) No changes to *main* should be necessary at this point.

¹⁹ The area of a triangle with sides a , b , c is given by Heron's formula:

$$\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$$

where $s = (a + b + c)/2$.

²⁰ In order for a , b and c to form a triangle, two conditions must be satisfied: all side lengths must be positive; the sum of any two side lengths must be greater than the third side length.

- (vi) Check that your program produces the same results as in part iii).
- (vii) Now you can remove any calls to `calcTriangleArea` from `main`. At this stage, your `main` method should just be calling `printTriangleArea`, and `printTriangleArea` should be calling `calcTriangleArea`. The `main` method should print out the triangle area once and only once. As usual, as the robot is going to test this code so don't print anything else.
- (viii) Add a *method* `testIfValidTriangle` which takes the lengths of the three sides as arguments and checks that they form a valid triangle, according to the criteria given in the previous exercise. This *method* should return a boolean value: *true* if the sides do make a valid triangle, or *false* if not. Use the `testIfValidTriangle` *method* in your `main` *method* to validate the input before trying to calculate the area.
- (ix) Read the next few pages (on documentation). Create javadoc documentation `import java.io.*;` for `MultMethEx2`. Ensure that you describe fully all arguments and return types you use in your methods. Also ensure that you describe any calculations and checks made in the code.
- (x) In addition to the javadoc comments, include any other comments that help add context to your code, and ensure that you are using a clear coding style with indentation, etc.
- (xi) Submit a zip file (.zip format only please) containing your version of `MultMethEx2.java` and the javadoc documentation to Moodle.

Note that in answering question (ii) you have benefited from having *encapsulated* a well-defined part of your program inside a separate *method* (`printTriangleArea`). This allows you to modify the internal workings of this *method* without requiring any changes elsewhere in the program where the *method* is used. This is considered good programming technique.

Documentation

The Java language provides a standard documentation tool called `javadoc`. This tool takes the standard comments in a `.java` file and converts them into a `.html` file describing the behaviour and methods of the class. This tool allows you to describe the use of all methods contained within a class including the parameters the method is called with and the return value. Your comments have to be in standard locations and special “tags” are used and called in the code using `@tag`. A list of commonly used tags is given in Table 1

For example if we compile the class `MaxTest.java` shown below and run the command `javadoc -author -version -use MaxTest.java` it will produce web pages in the current directory you are working in ²¹. You can open these files in a web browser. You will probably want to start with the file called `MaxTest.html` which will be linked to all the other pages.

```
/**
 * Purpose: Example of using a method to determine the larger
 *           of two values. Includes javadoc 'tags'.
 *           Includes an example of 'overloading' a method.
 *
 * @author Ian Bailey
 * @version 1.0
 */
public class MaxTest{

    public static void main(String[] argv) {
        int i=5;
        int j=2;
        int k=max(i,j);

        System.out.println(
            "The maximum between " + i +
            " and " + j + " is " + k);

        double x=0.543;
        double y=-0.123;
        double z=max(x,y);

        System.out.println(
            "The maximum between " + x +
            " and " + y + " is " + z);
    }
}
```

The program is continued over the page.

²¹If you use the command `javadoc -author -version -use -d doc MaxTest.java` the documentation files will be created in the subdirectory `doc`.

```

/**
 * This method returns the higher of two values.
 *
 * @param num1 the first number to be compared
 * @param num2 the second number to be compared
 * @return the higher of num1 and num2
 */
public static int max(int num1, int num2){
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}

/**
 * This method returns the higher of two values.
 *
 * @param num1 the first number to be compared
 * @param num2 the second number to be compared
 * @return the higher of num1 and num2
 */
public static double max(double num1, double num2){
    double result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
}

```

Tag	Description
@author <i>name-text</i>	Adds an "Author" entry with the specified name-text to the generated docs when the -author option is used. A doc comment may contain multiple @author tags.
@param parameter-name description	Adds a parameter with the specified parameter-name followed by the specified description to the "Parameters" section. When writing the doc comment, you may continue the description onto multiple lines. This tag is valid only in a doc comment for a method, constructor or class.
@return description	Adds a "Returns" section with the description text. This text should describe the return type and permissible range of values. This tag is valid only in a doc comment for a method. Omit @return for methods that return void and for constructors.
@version version-text	Adds a "Version" subheading with the specified version-text to the generated docs when the -version option is used. This tag is intended to hold the current version number of the software that this code is part of

Table 1: Most commonly used javadoc tags ^a

^a <http://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/>

Class variables, method variables and scope

Look again at the example that was used earlier, reproduced below. There are two variables x and y.

```
/**
 * Purpose: Example of program with two methods where one method
 *          takes an argument. The program prints the value of
 *          x squared to the screen.
 */
public class ArgExample
{
    // method to print square of a value to screen
    public static void printSquare(double y)
    {
        System.out.println(y*y);
        return;
    }
    // main method
    public static void main(String[] argv)
    {
        double x = 5.0;

        printSquare(3.0);    // prints the square of 3.0 to the screen
        printSquare(x);      // prints the square of x to the screen
    }
}
```


Variable `y`, which is declared at the same time as the `printSquare` *method*, only exists within this *method*. Each time the *method* is called, the variable is recreated, and then ceases to exist when flow returns to the *main method*. Code outside of the `printSquare` *method* cannot use `y`. That is, the *scope* of `y` is the *method* `printSquare`. The concept of scope was introduced in section F, but now it has to be extended to include methods and classes.

Similarly `x` is defined inside the *main method*, and may only be used in there. The *scope* of `x` is the *main method*. So `x` and `y` exist independently of each other, within their respective *methods*.²² They are known as *method variables*. It is not possible for `printSquare` to use the variable `x`, or for the *main method* to use `y`, because the variables do not exist there.

Before this section on multiple methods, all the variables you have used have either been declared in the *main method* (in which case their scope is from their declaration to the end of the *method*), or within a `for` loop, `if` statement or similar block of code (in which case their scope was from declaration to the end of the code block, denoted by the closing curly bracket `}`). Now you have also created variables in other *methods*, but the principle is just the same.

There is one further level at which variables can be declared. These are *class variables*. They are variables declared outside of any *methods*, but within the class, and as such exist for all *methods* in the class. Their basic declaration is the same as with any variable, however like *methods* they should have `public static` added to the beginning. Here is a simple program which does something similar to the previous example, but where the scope of `x` is different.

²²Because the variables exist independently in their separate regions, there is no reason why they should not be given the same variable name if you wish. Even if both were called `x` the two variables would exist completely independently in their respective scopes.

```

/**
 * Purpose: Example of program with a class variable
 *          The program prints the value of x squared
 *          to the screen.
 */

public class ScopeExample {
    // declare class variable x
    public static double x;

    // method to print x squared to screen
    public static void printXSquare()
    {
        System.out.println(x*x);
        return;
    }

    // main method
    public static void main(String[] argv)
    {
        // give x a value and print its square to screen
        x = 5.0;
        printXSquare();

        // change value of x and print square to screen
        x = 2.0;
        printXSquare();
    }
}

```

Since `x` exists within the entire class, both *methods* can make use of it.

There is no need to pass the value of `x` to the *method* `printXSquare`, so there is no need for the *method* to take an argument. Note that because `printXSquare` has been written to print the value of x^2 to the screen, it can not be used to print the square of any other value. It is not in general a very useful *method*. Class variables should generally be used sparingly.

To summarise these ideas of variable scope:

Class Variables An example is the variable `a` in the code below. For now you should declare them using `public static`. The variable is declared in the class, but outside of the *methods*. It is visible to all the the class, so all the *methods* of the class may use it.

Method Variables Examples are `b` and `c` in the code below. *Method* variables are declared inside a *method* (`c`), or as an argument in a *method* declaration (`b`). The scope of `c` is from its declaration to the end of the *method*. The scope of `b` is the entire *method*.

Variables declared in a block of code E.g. the variables `d` and `e` in the code below. They are normally declared within a block of code (code within a pair of curly brackets `{ }`) and their scope is from the declaration to the end of that block of code. In the case of a `for` loop the declaration is in the round brackets following `for` and the variable exists for the duration of the loop.

```

/**
 * Purpose:  Program that does nothing useful, but using variables
 *           that have a variety of scopes.
 */

public class ScopeSummary
{
    // scope of a is the entire class
    public static double a;

    // scope of b is all of method alpha
    public static double alpha(int b)
    {
        double c;                // scope of c from here to method end
        c = a*b;
        return c;
    }

    public static void main(String[] argv)
    {
        // scope of d is all of for loop
        for (int d=1; d<100; d*=2)
        {
            if ((d%3) != 0)
            {
                a = d%3.0 + 1.234;
                double e = alpha(d);    // scope of e starts here
                System.out.println(e);
            }                          // scope of e ends here
        }
    }
}

```

Exercise 3.2 _____ (Formative: 4 Marks)

With reference to the example code above, categorise the following statements as true or false.

- (i) The variable `a` is visible in any method of the class.
- (ii) The variable `d` is visible inside the method `alpha`.
- (iii) The variable `c` is not visible inside the method `main`.
- (iv) The variable `e` is visible everywhere inside the method `main`.

Enter your answers on Moodle.

Methods and Flowcharts

Methods do not introduce any new concepts to the use of flowcharts. The only change is that all of the operations contained within a method can be represented as a single action, i.e. in a flowchart as a single rectangle. This simplifies most flowcharts greatly.

H Arrays

Creating an array

Arrays are useful when an ordered group of values need to be stored provided that

- The values are all of the same *type*. For example, they could all be of type `int` or they could all be of type `double`, but not a mixture of the two.
- There is a fixed number of values which doesn't vary during the execution of your program.

The entries of a vector or matrix are good examples of values that could be stored in an array.

There are similarities in the behaviour of arrays and variables, but also some differences — here are some examples of how to create an array that will contain entries that are of *type* `double`.

```
// create an array and fill with some numbers
double[] vector1 = {1.2, 0.0, 5.3, 1.4};

// create an array but fill in values later
double[] vector2;
vector2 = new double[4];           // where 4 is the size (no. of entries)
```

The first piece of code creates the arrays `{1.2, 0.0, 5.3, 1.4}` called `vector1` and the second piece of code creates the array `{0.0, 0.0, 0.0, 0.0}` called `vector2` — note that *unlike a variable*, if you do not specify initial values, all entries are automatically initialised to zero. The first part of both examples looks like declaring a variable apart from the square brackets. In the example we've created arrays of *type* `double`, but you can equally well use other data *types* as desired. To get arrays of integer values you would just replace `double` with `int` in the examples²³.

The second parts of both examples determine the length or size of the array, for `vector1` this was done by explicitly giving the values of each *element*. In the case of `vector2`, space in the computer's memory was reserved for four values of *type* `double`, but the values will be filled in later in the code. As with variable declarations the two lines of code can be reduced to one, e.g.

`double[] vector2 = new double[4]`. However, unlike a variable, you cannot split the first example into two pieces as shown below.

```
// Trying to create an array and fill with some numbers
// This won't work... You can only use the {num1, num2, ...} syntax
// as part of the line where you declare the array.
double[] vector1;
vector1 = {1.2, 0.0, 5.3, 1.4};
```

Figure H.10 illustrates the creation of an array, which results in the values in the array occupying four adjacent words in the computer's memory. The name of the array gives you a way of referring to this data. We say that the name of the array is a *reference* to the data.

²³You can even create arrays of *objects* such as the `Scanners` we used earlier.

```
double[] a = {1.0, 2.0, 3.0, 4.0};
```

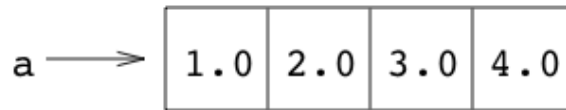


Figure H.10: Example of an array of doubles in memory.

Array manipulation and references

To make use of the values stored in arrays you should refer to each *element* individually. The numbering of each *element* starts at zero, thus for `vector1` above, the number 1.2 is the zeroth *element* `vector1[0]`, 0.0 is the first *element* `vector1[1]`, and 5.3 is the second *element* `vector1[2]`, and so on. Individual array *elements*. e.g. `vector1[2]`, can then be used in just the same way as any other double variable would be. For example you could assign a number to it or use it in a calculation, like this:

```
System.out.print(vector[1]);  
vector1[0] = 7.0;  
vector1[2] = vector1[0]*(-7.0) + vector1[3];
```

Often though you will want to perform the same or similar operations on all the entries of an array in turn. This can be done using loops, and shows the major advantage of an array over using many individual variables²⁴, as in this example:

```
// Create an array called 'list' and then print out the elements  
// to the screen  
int[] list = {1, 4, -3, 66, 19};  
for (int i=0; i<5; i++)  
{  
    System.out.println("Element " + i + " is " + list[i]);  
}
```

This saves us typing out `System.out.println(...)` five times. Note that here the loop variable `i` runs from 0 to 4 as the *elements* of the array are numbered from zero. It is a very common mistake to let the loop variable increase to too high a value. For example it would be easy to type `i<=5` by mistake, by thinking about the array having five entries. If a variable does go beyond the end of an array, the error will not be picked up by the compiler, but instead results in an `ArrayIndexOutOfBoundsException` when the code is executed. Also be careful to start with `int i = 0` rather than `int i = 1` or the code will skip the first entry in the array.

²⁴ There are also other more efficient ways of looping over the elements of an array. If you are interested then look up `for each` loops in the lecture slides, or on the Java website.

Consider this example:

```
// Create an array of general length filled with random numbers
int N=42; // Set N to some value as an example
double[] randomNos = new double[N];
for (int i=0; i<N; i++)
{
    randomNos[i] = Math.random();
}
```

where the loop not only saved us typing `randomNos[...] = Math.random();` many times (N could be 10, 1000, 100000 or more!), but in fact we don't even need to know whether the array contains 10 elements or 100000 elements when we write the code apart from where we assign a value to N.

If you wish to find out the length of any array in your code, append `.length` to the end of the name of the array. In the code above, the value of `list.length` would be 5, and the value of `randomNos.length` would be whatever the value of N is (42 in our example). In the former case it would clearly be better to use `list.length` instead of typing 5. Using `list.length` makes the code more general and requires less alteration if the length of the array were altered in a second version of the program.

Copying Arrays

What if you want to create a new array which contains the same values as an existing array? You may be tempted to type `double[] b = a` where `a` is the array to be copied to the new array `b`. This is valid code, but it has a slightly different effect to the one desired. In order to create a copy of an array you should use a `for` loop and set each value individually:

```
// Create an array b of the same length as array a
double[] b = new double[a.length];

// set each value of b equal to the equivalent value of a
for (int i=0; i<a.length; i++)
{
    b[i] = a[i];
}
```

Exercise 4.1 _____ (Formative: 4 Marks)

Consider the following program snippet

```
double[] a = {5,7,9,11};
double[] b = new double[a.length];

b=a;

for (int i=1; i<a.length; i++){
    b[i]+=2;
}

for (int i=1; i<a.length; i++){
    System.out.println(a[i] + " " + b[i]);
}
```

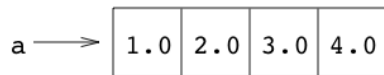
Firstly try to predict what the output will be. Now write a program which includes this piece of code where you add the appropriate class declaration, main method declaration, etc. Use your program to help you answer the questions on the Moodle quiz.

For usual variables, a line of code like `double b = a;`, copies the value of variable `a` to the variable `b`. With arrays, this is not the case. Instead, `b` is set to refer to the same place in memory that `a` refers to. There is therefore still only one copy of the data in the computer's memory. Any change to the values of `b` is also a change to the values of `a`.

This difference is due to arrays being *reference data types*. This means that they are handled *by reference*, unlike `int`, `double`, `boolean` etc. which are *primitive data types*, handled *by value*.

Figure H.11 illustrates what happens when you assign one array to another.


```
double[] a = {1.0, 2.0, 3.0, 4.0};
```



```
double[] b = a;
```

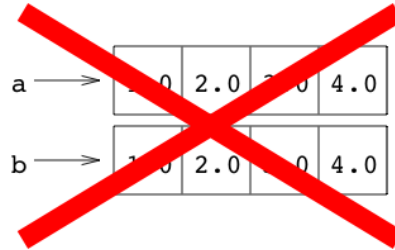
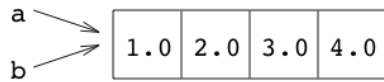


Figure H.11: What happens when array *a* is defined and then array *b* is set equal to *a*? Since *a* and *b* are references, they both point to the same array data (bottom left). It is important to understand that *b* does **not** get its own copy of the array data (bottom right).

For now it may be hard to see the advantage of arrays behaving in this way, but one becomes apparent if you consider how to pass a long list of numbers to a method. Rather than have a long argument list with individual variables, you can just let the method know where the array of numbers is stored in the computer's memory. In other words, we pass a *reference* to the data, not the data itself which would require using twice as much memory (the original data and a complete copy). This is discussed in the next section.

The only other strange behaviour of *reference data types* to be aware of at this stage is that of the `==` comparison operator as shown below.

```
double[] a = {1.0, 2.0, 3.0, 4.0}
double[] b = {1.0, 2.0, 3.0, 4.0}

if (a == b)
{
    // code here not executed
    // even though the contents
    // of the arrays are equal...
}
```

Rather than comparing the values inside *a* and *b*, their *references* are compared — that is the code asks whether *a* and *b* refer to the same area of memory. So even though they have the same values the test `a == b` will be false and the code within `if` never executed.

Conversely, if a and b are both references to the same place in memory, then they are considered to be equal, so the comparison will be true as shown below.

```
double[] a = {1.0, 2.0, 3.0, 4.0}
double[] b = a

if (a == b)
{
    // code here always executed, a==b true!
    // both references refer to same array
}
```

Exercise 4.2 (Formative: 10 Marks)

This exercise will be marked by the robot.

- Write a program, 'Arrays.java', that creates an integer array of length N, where N is an `int` you declare in your program.
- The program should get a value for N from the user using a `Scanner`. The number must be greater than 3, and your program should loop until a number greater than 3 is entered. Don't print anything to the screen yet.
- You should now fill the array with the first N numbers of the *Padovan sequence*. This might be better handled in a separate method, but for now just do this inside the main method.
- The first three numbers in the sequence are 1 (i.e. $x(1)=1$, $x(2)=1$, $x(3)=1$).
- The rule to compute the rest is

$$x_n = x_{n-2} + x_{n-3} \quad (4)$$

- The resulting sequence begins 1, 1, 1, 2, 2, 3, 4, 5, 7, ...
- After the program has calculated the first N numbers in the Padovan sequence and filled the array, add a separate loop to your main method which prints out the contents of the array.
- Try generating the first 79 numbers in the sequence. Are you getting the correct answer? (Hint: Compare the size of your answer to the maximum value allowed for an `int` <http://download.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>).
- If you are not getting the correct answers what modification to the code could you make? Adjust your code if necessary so that it can reliably generate at least the first 100 Padovan numbers.
- NB Before making the changes below it would be a good idea to make a copy of your working directory in case things go wrong!
- Firstly read the following section on how to pass an array to a methods as an argument.
- Write a method called `printArray` that can take an array of integers as an argument and print out the contents of the array.
- In your main method you should currently have a loop which prints out the contents of the array. Replace this with a call to your new method `printArray`.
- Your program should print the first N entries of the Padovan series separated by spaces. It should not print anything else. Your output can be on more than one line. E.g.

```
1 1 1 2 2
3 4 5 7
```

- Make sure your code is clear. We won't be marking the javadoc documentation on this exercise, but it's a good habit to include the javadoc comments before each method anyway.
- Finally, upload your `Arrays.java` file to Moodle.

Passing arrays to methods

Arrays are *reference variable types* and as such their behaviour is slightly different from that of variables of *primitive data types* such as `double`, `int`, etc. Passing arrays to or from *methods* can be very useful. However it is the most obvious area where *reference data types* behave slightly differently from *primitives*.

To pass an array to a *method*, the *method* declaration should look similar to this:

```
public static void takeArrayMethod(double [] numberList){
    // method code
    return;
}
```

and then the *method* call might look like:

```
takeArrayMethod(vector1);    //or
takeArrayMethod({1.0, 3.4, 5.2});
```

where `vector1` would be an array of double values already created earlier in the program.

For a *method* to return an array the *method* should be declared as you might expect:

```
public static int [] returnArrayMethod(){
    // method code which should include
    // creating an int array called vector2
    return vector2;
}
```

When a *primitive data type*, e.g. `double`, is passed to a *method*, its value is copied into the space in memory referenced by the new *method* variable. For *reference data types*, a new *reference* is created, but unlike for *primitives*, the data that is referenced is **not** copied to a new area of memory. Instead the new *reference* is set to refer to the original area of memory storing the data. We have *passed a reference*.

As was mentioned earlier, it is not possible for a *method* to change the value of its arguments. E.g. if you have a main method that contains

```
int i, j;
i=5;
j=increase(i);
```

and a method called `increase` that looks like this

```
// method that tries to change argument value
public static double increase(double d)
{
    d += 10;
    return d;
}
```

then the value of `i` in the main method will not be changed.

The equivalent for an array is that the variable (i.e. the array name) can not have its reference changed. E.g. if we have this in the main method

```
int [] vector1= {4,5,6};  
int [] vector2;  
vector2=change( vector1 );
```

and a method called `change` which looks like this

```
// method that tries to change argument reference  
public static int [] change(int [] vector3)  
{  
    vector4 = {1,2,3};  
    vector3 = vector4;  
    return vector3;  
}
```

then `vector1` still refers to the original space in memory containing 4, 5, 6.

However it is possible to change individual values in the array, in the area of memory that `vector3` refers to. This is in contrast to primitive data *types* where the value of an argument can not be changed. E.g.

```
// method that changes the values of an array  
public static void valueChange(int [] vector3)  
{  
    for(int i=0; i<vector3.length; i++)  
    {  
        vector3[i] = i+1;           // this IS allowed  
    }  
    return;  
}
```

Note that in this particular case, since any changes are made directly to the original data, there is no need to have `return vector3;` thus the *method* is declared as `void`.

Let's look at one more example for completeness. Assume in some method we have this piece of code

```
int [] num = {2,4,8,16};  
int total=sumArray(num);
```

where somewhere else in the `class` we've defined a method `processArray` that could look like this

```
public static int sumArray(int [] values)  
{  
    int sum=0;  
    for (int i=0; i<values.length; i++)  
    {  
        sum+=values[i];  
    }  
    return sum;  
}
```

here we have a method that uses the elements of the array to calculate a return value but doesn't change the elements themselves.

Note that anything created with a `new` command (we've seen some examples already), not just arrays, is a *reference data type* and it is the reference, not the values in memory, that are passed to a method.

Multi-dimensional arrays and nested loops

It is also possible to create arrays with two or more dimensions. Two-dimensional arrays are useful for storing matrices for example. In Java, arrays of more than one dimension are created by making arrays of arrays. This is known as *nesting*. The follow examples should make this more clear. Note that arrays in Java do not have to be square or rectangular, although you will usually want them to be.

```
// ways of creating two dimensional arrays
int [][] matrix = {{0, 1, 2, 3},
                  {4, 5, 6, 7},
                  {8, 9, 10, 11}}; // Create and fill a 3 by 4 matrix

int [][] square = new int [5][5];    // Creates 5 by 5 matrix with
                                     // values set to zero

int [][] notRect = new int [2][];    // Create references to two arrays
int [] temp = {1,2,3};               // The values stored look like this
notRect[0] = temp;                   //      {{1, 2, 3}
notRect[1] = new int [5];             //      {0, 0, 0, 0, 0}}
```

Figure H.12 illustrates the non-rectangular nested array structure `notRect` created by this code.

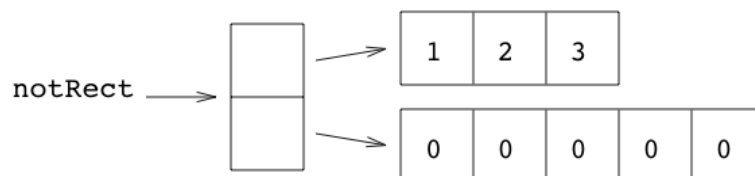


Figure H.12: *Non-rectangular nested arrays. See text for sample code which creates these arrays.*

Taking the above 2D array `matrix` as an example, in order to perform operations on each value of a multi-dimensional array, it is necessary to nest several `for` loops. The example code below increments each entry of `matrix` by 1.

```
for (int i=0; i<3; i++)
{
    for (int j=0; j<4; j++)
    {
        matrix[i][j]++;
    }
}
```

You can think of the outer loop moving down the rows of the `matrix`, then the inner loop moves along each row. Note that when you nest loops, the loop variable of each loop (`i` and `j` in the example above) must be different for this to work properly.

With two-dimensional arrays such as matrices, it can be helpful to print the entries to screen in columns and lines as they would normally appear in a matrix. Firstly, creative use of `System.out.println()` and `System.out.print()` (the latter does not move to the next line after printing the output) can help. Secondly, there are some special characters: `"\t"`, which produces a tab and `"\n"` which moves to the next line. These can be included anywhere in a string.

Flow-charts involving arrays and loops

Figure H.13 shows the flow-chart for calculating a polynomial that has been stored in an array. The code used to calculate this is given below. If a polynomial:

$$f(x) = A_n x^n + a_{n-1} x^{n-1} + \dots + A_1 x^1 + A_0$$

is rewritten as

$$f(x) = A_0 + x \times (A_1 + x \times (A_2 + x \times (\dots x \times A_n) \dots))$$

which can again be rewritten as

$$f(x) = (((A_n \times x) \times x + A_{n-1}) \dots + A_1) + A_0$$

then we get a much quicker computer calculation of the polynomial and the calculation can be coded using:

```
polyValue = A[n];
for (int j=n-1; j>=0; j--) polyValue = polyValue*x+A[j]
```

where somewhere earlier in the code the values of the coefficients must have been set. Alternatively, we could use code like this if we want to be very concise:

```
polyValue = A[j=n];
while (j>0) polyValue = polyValue*x+A[--j];
```

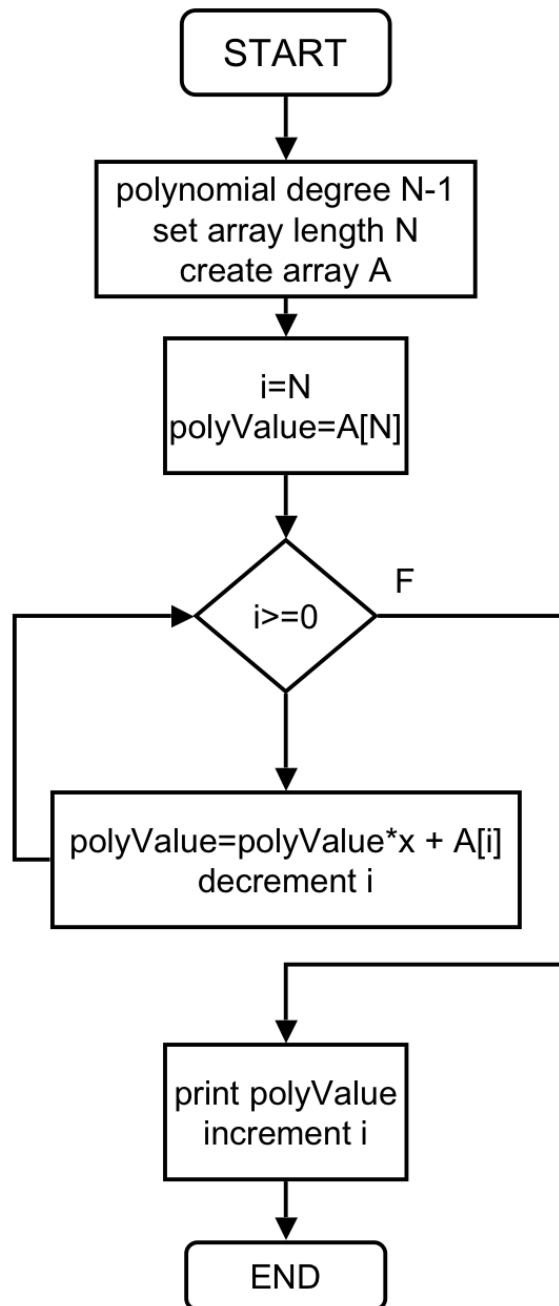


Figure H.13: *Example of flow-chart for evaluating a polynomial*

Exercise 4.3 (Formative 10 Marks)

This exercise will be marked by the robot.

The two-dimensional rotation matrix looks like

$$\begin{pmatrix} \cos(t) & -\sin(t) \\ \sin(t) & \cos(t) \end{pmatrix}$$

for some angle t .

- Write a class `RotationMatrix`.
- In the main method use a `Scanner` to read in the angle t in radians. Don't print anything to the screen at this point.
- Declare a 2 by 2 array of `doubles` and fill it with the values of the rotation matrix corresponding to the angle t . Note that the `Math` methods in Java assume that their arguments will be in radians, so you won't need to convert t to degrees.
- In the main method use the `Scanner` you set up above to read in an integer, $nRotate$ that represents the number of times the rotation matrix is going to be applied to a vector.
- Now initialise an array of `doubles` that represents a 2-d vector, and use your `Scanner` to read in the x and y coordinates of this vector (one at a time).
- Write a method which takes a 1-dimensional array of `doubles` representing an n-dimensional vector as input and prints out the contents (one number per line, with no extra text or symbols). Name this method `printVector`. Check that it works.
- Write a method that takes a 2-dimensional array of `doubles` representing a square matrix as input and prints out the contents sequentially row by row (each line of output should contain one row of the matrix where each number is separated by a single tab with no extra text or symbols). Finish the method by printing a blank line for clarity. Name this method `printMatrix`. Check that it works.
- Write a method that takes a 1-dimensional array of `doubles` representing an n-d vector as input and a 2-dimensional array of `doubles` representing an n by n matrix as input and returns a 1-dimensional array of `doubles` with values equal to $R.n$ where R is the matrix and n is the initial vector. Name this method `applyMatrix`. Use this together with the `printVector` and `printMatrix` methods to check that it works.
- Write a method that takes a 2-dimensional array of `doubles` representing a square matrix as input and returns a 2-dimensional array of `doubles` with values equal to the transpose of the initial matrix (i.e. $N_{ij} = M_{ji}$ where N is the transposed matrix and M is the original matrix). Name this method `transposeMatrix`. Use this together with the `printMatrix` method to check that it works.
- Now that you have all the methods you need, we can use them.
- In your main method, first print out the rotation matrix.
- Next, calculate the transposed rotation matrix and print it.
- Next, apply the rotation matrix $nRotate$ times to the initial vector. After each rotation use `printVector` to print out the new values of the vector.

- Next, apply the transposed rotation matrix $nRotate$ times to the vector you obtained at the end of the last step. After each rotation use `printVector` to print out the new values. You may or may not already know that the rotation matrix is an example of an ‘orthogonal’ matrix. This means that the ‘transposed’ matrix you’ve created in your program represents the ‘inverse’ rotation. I.e. if the matrix you started with represents a rotation by ‘t’ radians clockwise then the transpose matrix represents a rotation by ‘t’ radians in an anti-clockwise direction.
- Your code should apply the original matrix and the inverse (transpose) matrix an equal number of times. So, in principle you should end up with a vector that is the same as the one you started with. Note that the more rotations you apply, the less true this will be due to the finite precision with which numbers are stored.
- We won’t be marking commenting or javadoc on this exercise but you should include it in case we need to look at your code manually.
- When you are confident your code works submit the `RotationMatrix.java` file only to Moodle. Below is an example of some input and the expected output.

Example of input:

```
0.1
2
1.0
1.0
```

Example of expected output generated from the input above:

```
0.9950041652780258 -0.09983341664682815
0.09983341664682815 0.9950041652780258

0.9950041652780258 0.09983341664682815
-0.09983341664682815 0.9950041652780258

0.8951707486311977
1.094837581924854
0.7813972470461805
1.178735908636303
0.8951707486311977
1.094837581924854
1.0000000000000002
1.0000000000000002
```

This last exercise is in some sense our first physics simulation in this course. You could imagine that it represents a calculation of the position of a rotating object which has constant angular momentum. Each time we rotate the object we’re taking a step forward in time and calculating the new position of the object. We then run the simulation backwards in time as a cross-check. Of course, this isn’t a very interesting simulation. It would be much more interesting if the angular momentum wasn’t constant. If we were trying

to solve the equations of motion for such a system analytically we'd need to write them down as differential equations and then integrate them. How do we go about integrating equations of motion numerically on a computer? That's one of the main themes of the rest of the course.

I Numerical integration

Numerical methods are a collection of techniques for solving mathematical problems which lend themselves to implementation as computer programs. They exist to solve all sorts of problems, for example differentiation, minimisation, sorting data and solving differential equations.

Many physical systems can be represented by differential equations, but solving them analytically is only possible in special cases as most non-linear systems of equations do not have analytical solutions. In order to make numerical predictions it is often necessary to take an approximate numerical approach to integrating them. These techniques are based around the fact that an integral can be written as a limit of a ‘Riemann sum’, like this:

$$\int_a^b f(x)dx \equiv \lim_{N \rightarrow \infty} \sum_{i=1}^N f(x_i) \left(\frac{b-a}{N} \right) \quad (5)$$

Some basic techniques for numerically-integrating functions of one variable to obtain approximations to the true value of the integral are shown in the following sections. Remember that all of these techniques are *approximations* based on the Riemann sum above but using only a finite number of terms in the sum.

I.1 Left-Endpoint Rectangle Rule

The left-endpoint rectangle rule is a very simple method for estimating the integral of a function of one variable (see figure I.14). The area under the curve is estimated using the value at the lower limit of the integration region, i.e.

$$\int_a^b f(x)dx \approx (b-a)f(a) \quad (6)$$

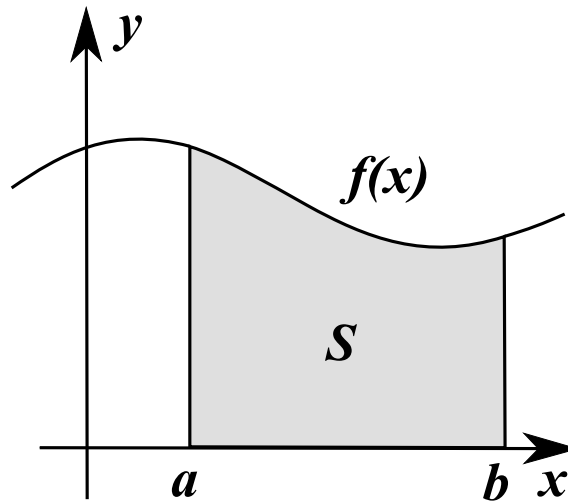


Figure I.14: An illustration of a generic curve $f(x)$ between a and b .

The accuracy can be improved by simply dividing the interval into N equidistant sub-intervals and summing up the results from each sub-interval, giving the expression

$$\int_a^b f(x)dx \approx \sum_{i=1}^N h f(x_i) \quad (7)$$

where $h = (b - a)/N$ and $x_i = a + h(i - 1)$.

This is a very simple approximation of an integral and often gives poor results. For example, if $f(x)$ is a ‘monotonically increasing’ function, e.g. $f(x) = 1 + 0.5x$, then the left-endpoint rectangle rules will always underestimate the integral no matter how many sub-intervals the function is divided into.

I.2 Mid-Point Rectangle Rule

The left-endpoint rectangle rule can be easily improved for many functions by using the value in the middle of the interval rather than a point on the edge of the interval:

$$\int_a^b f(x)dx \approx (b - a)f(a + (b - a)/2). \quad (8)$$

as before, subdividing the interval into many sub-intervals will generally improve the approximation giving an expression similar to equation 7. As always, the number of sub-intervals needed to obtain an accurate result will depend on how rapidly the value of the function itself varies.

I.3 Trapezium Rule

The trapezium rule states that you can approximate the area under a curve by a trapezium, as shown in figure I.15. The area is therefore calculated like this:

$$\int_{x_1}^{x_2} f(x)dx \approx h \left(\frac{1}{2}f(x_1) + \frac{1}{2}f(x_2) \right) \quad (9)$$

This method is exact for polynomials up to degree 1, i.e. straight lines, because such functions can be exactly represented by a trapezium. It is clearly an approximation for higher order polynomials (x^2, x^3 , etc.) and other functions. The error will also be small if h is small and vanish in the limit that h tends to zero.

Note that the area of a trapezium is its average height multiplied by its width, i.e. $\frac{(f(x_1)+f(x_2))}{2}h$.

As with the rectangle rules, to make this rule useful we need to break the curve to be integrated into many small intervals as shown,

$$\begin{aligned} \int_{x_1}^{x_5} f(x)dx &\approx h \left(\frac{1}{2}f(x_1) + \frac{1}{2}f(x_2) \right) \quad \dots \text{region A} \\ &+ h \left(\frac{1}{2}f(x_2) + \frac{1}{2}f(x_3) \right) \quad \dots \text{region B} \\ &+ h \left(\frac{1}{2}f(x_3) + \frac{1}{2}f(x_4) \right) \quad \dots \text{region C} \\ &+ h \left(\frac{1}{2}f(x_4) + \frac{1}{2}f(x_5) \right) \quad \dots \text{region D} \end{aligned}$$

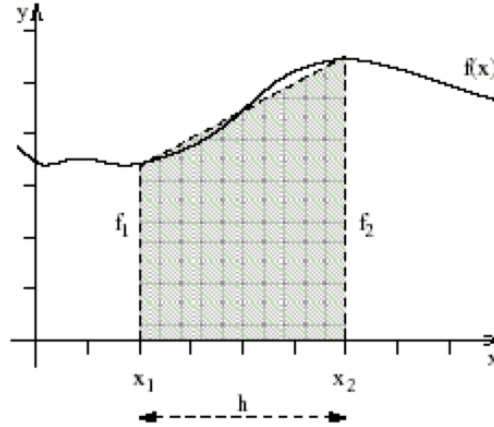


Figure I.15: A straight line is used to approximate the curve $f(x)$ between x_1 and x_2 . This forms a trapezium (shaded area) which can be used to calculate approximately the area under the curve between these bounds.

where $x_1 = a$ and $x_5 = b$ the limits of the integral. Figure I.16) illustrates this approach.

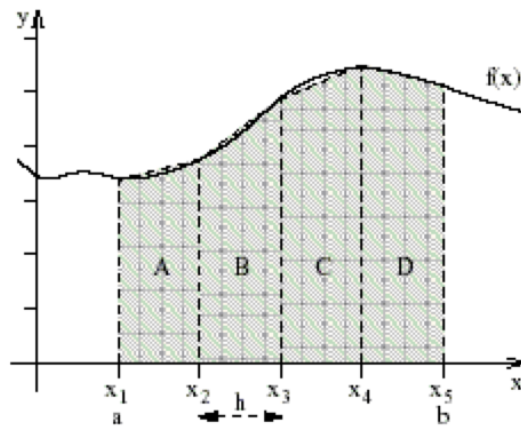


Figure I.16: The curve $f(x)$ is now approximated by a series of short straight lines which each form a trapezium. Summing the area of these trapezia gives an approximation of the total area under the curve between a and b .

We can generalise this ‘extended trapezium’ or ‘compound trapezium’ rule to N points giving

$$\int_{x_1}^{x_N} f(x)dx \approx h \left[\frac{1}{2}f(x_1) + f(x_2) + f(x_3) + \dots + f(x_{N-1}) + \frac{1}{2}f(x_N) \right] \quad (10)$$

$$\approx h \left[\frac{1}{2}(f(x_1) + f(x_N)) + \sum_{i=2}^{N-1} f(x_i) \right] \quad (11)$$

where $x_1 = a$ and $x_N = b$ the limits of the integral.

Note that h is still the width of a single interval, so

$$h = \frac{b - a}{N - 1} \quad (12)$$

and there are $N - 1$ trapezia not N .

I.4 Accuracy of Numerical Integration

Consider an arbitrary function, f , which we want to integrate numerically. We can expand it as a Taylor series around a ‘sampling’ point, x_i to obtain

$$f(x) = f(x_i) + (x - x_i)f'(x_i) + \frac{1}{2}(x - x_i)^2 f''(x_i) + \dots \quad (13)$$

where the apostrophe denotes differentiation with respect to x .

If we now integrate this expression with respect to x between x_i and the next sampling point x_{i+1} we obtain

$$\begin{aligned} \int_{x_i}^{x_{i+1}} f(x) dx &= f(x_i) \int_{x_i}^{x_{i+1}} dx + f'(x_i) \int_{x_i}^{x_{i+1}} (x - x_i) dx \\ &+ \frac{1}{2} f''(x_i) \int_{x_i}^{x_{i+1}} (x - x_i)^2 dx \\ &+ \dots \end{aligned} \quad (14)$$

the difference between x_i and x_{i+1} is h , so with a little bit of algebra we obtain

$$\begin{aligned} \int_{x_i}^{x_{i+1}} f(x) dx &= hf(x_i) + \frac{1}{2} h^2 f'(x_i) \\ &+ \frac{1}{6} h^3 f''(x_i) \\ &+ O(h^4) \end{aligned} \quad (15)$$

where the big ‘O’ notation denotes all of the missing higher-order terms starting with the term proportional to h^4 .

So we see that in general we can represent our integral as an infinite series in h which we have to truncate in order to be able to write a practical expression which we can use in our computer program (or indeed on paper).

If we look at the formulas for the rectangle rule and the trapezium rule we see that they are both linear in h . By comparison to our integrated Taylor series (equation 15) we might assume that the error in all cases (left-hand rectangle, midpoint and trapezium) will be of order h^2 . Actually, this is incorrect. Both the rectangle and trapezium rules when applied to N sub-intervals contain a sum from 1 to N , and N is inversely proportional to h . Hence we would actually expect the error in all cases to be linear in h . I.e. for the rectangle rules we expect that

$$\int_{x_1}^{x_N} f(x) dx = \sum_{i=1}^N hf(x_i) + O(h) \quad (16)$$

and for the trapezium rule we expect

$$\int_{x_1}^{x_N} f(x)dx = h \left[\frac{1}{2}(f(x_1) + f(x_N)) + \sum_{i=2}^{N-1} f(x_i) \right] + O(h) \quad (17)$$

In fact, equation 16 is correct for the left-hand rectangle rule but not for the mid-point rule, and equation 17 is incorrect for the trapezium rule. For the trapezium rule and for the mid-point rule the dominant error term is actually of order h^2 , despite the sum over sub-intervals that we've just discussed. This arises because the coefficient of the term which is linear in h cancels out in the case of these two methods.

To see why this is true, we can expand our function as a Taylor series as before, but this time around the point x_{i+1} , to give

$$f(x) = f(x_{i+1}) + (x - x_{i+1})f'(x_{i+1}) + \frac{1}{2}(x - x_{i+1})^2 f''(x_{i+1}) + \dots \quad (18)$$

When we integrate this expression with respect to x over the same interval as before we obtain

$$\begin{aligned} \int_{x_i}^{x_{i+1}} f(x)dx &= f(x_{i+1}) \int_{x_i}^{x_{i+1}} dx + f'(x_{i+1}) \int_{x_i}^{x_{i+1}} (x - x_{i+1})dx \\ &\quad + \frac{1}{2}f''(x_{i+1}) \int_{x_i}^{x_{i+1}} (x - x_{i+1})^2 dx \\ &\quad + \dots \end{aligned} \quad (19)$$

It's hopefully obvious that when we evaluate this, some terms will pick up negative signs due to $(x - x_{i+1})$ being negative in the interval x_i to x_{i+1} . In terms of h the result is

$$\begin{aligned} \int_{x_i}^{x_{i+1}} f(x)dx &= hf(x_{i+1}) - \frac{1}{2}h^2 f'(x_{i+1}) \\ &\quad + \frac{1}{6}h^3 f''(x_{i+1}) \\ &\quad + O(h^4) \end{aligned} \quad (20)$$

The trapezium rule can be formed from taking the average of equation 15 and equation 20. When we form this average and introduce the sum over sub-intervals we find that due to the negative sign in equation 20 the term linear in h will cancel out, and we obtain

$$\int_{x_1}^{x_N} f(x)dx = h \left[\frac{1}{2}(f(x_1) + f(x_N)) + \sum_{i=2}^{N-1} f(x_i) \right] + O(h^2) \quad (21)$$

It can be shown that this error term can be written out (approximately) as

$$\frac{1}{12}h^2 (f'(x_1) - f'(x_N)) \quad (22)$$

Recall again that N here is the number of sampling points, not the number of trapezia. Beware that other texts may use the other convention where N is the number of trapezia and there are $N + 1$ sampling points. Don't mix up the two conventions!

The trapezium rule is an example of a ‘first-order’ numerical integration technique. A similar demonstration can be made for the midpoint rule which is also ‘first-order’. First-order techniques have errors of the form $O(h^2)$. In principle one can form higher-order approximations with errors of the form $O(h^n)$ for higher values of n . Such techniques should in general be better at approximating an integral compared to the techniques we’re presented above, for the same number of sampling points.

One such higher-order approximation is Simpson’s rule (which combines aspects of the trapezium and midpoint rules), which we quote here for completeness without derivation. It takes the form

$$\begin{aligned} \int_{x_1}^{x_N} f(x)dx \approx & \frac{1}{3} \left[f(x_1) + f(x_N) \right. \\ & + 4 \sum_{i=1}^{N/2} f(x_1 + (2i-1)h) \\ & \left. + 2 \sum_{i=1}^{N/2-1} f(x_1 + 2ih) \right] \end{aligned} \quad (23)$$

In general when you are using numerical integration you need to be aware of two sources of error. The first is the intrinsic error in the technique you are using. This is determined by the ‘order’ of the technique, and we have seen above that different techniques will have errors which depend on different powers of h and hence on different powers of N (the number of sampling points). If the number of sampling points gets very large then we might also have to worry about ‘rounding errors’ such as those we have come across in earlier parts of these notes. For the exercise in the next section, you can probably neglect rounding errors as long as you are using variables of type `double`.

I.5 An Exercise in Numerical Integration

Here you will use the trapezium rule to numerically integrate a function. This exercise will allow you to put into practice many of the aspects of Java programming that you have covered so far.

Exercise 5.1 _____ (Summative: University Scale, 10%)

You need to write a program that numerically integrates a mathematical function. You are going to use this program to investigate the properties of numerical integration techniques, and then use the results to help you write a short report. Follow the steps below.

1. Firstly, write a method called `f` with a `return` value equal to the value of the function you want to integrate. For example, this is a definition of $f(x) = 2x^2$:

```
public static double f(double x){
    return 2*x*x;
}
```

2. Write a method which implements the left-endpoint rectangle rule. It should take as arguments the range over which to integrate (`a` and `b`) and the number of sampling points used (`N`). Choose appropriate data types for `a`, `b` and `N`. Your method should use these parameters to compute the integration using the rectangle rule and return the result. It should invoke the method `f` you defined in the previous step²⁵. You might be tempted to use arrays to store the values of `f` at each of the sampling points. Although there are times when this might be useful, you shouldn't use arrays in this exercise as you may find that the computer will run out of memory (stack overflow)!
3. Write a main method to prompt the user to type in values of `a`, `b`, and `N`, and use a `Scanner` to read these values. Remember not to create more than one `Scanner`! Your main method should then invoke your left-endpoint rectangle rule method and print the result.
4. Test your program with a second order polynomial for several values of `N`. In this case you can easily integrate the polynomial analytically, so do this first and then compare the numerical results from your program to the analytical result. Check what happens to the difference between the expected result and the numerical result as `N` increases. *Quantitatively*, how does this difference (i.e. the error) depend on `N`? Is this what you expect from the formula for the rectangle rule? Hint: if you have data which you expect is of the form N^d for some constant d , how would you go about finding the value of d ?
5. Write an additional method which implements the midpoint rectangle rule. Again, test this method for a range of values of `N`. Compare the 'convergence' rates of the midpoint and left-endpoint rectangle rules (i.e. the rates at which the numerical values of the two methods tend to the correct value as `N` varies.) You may want to try a couple of different polynomials as your function `f` to make sure that any conclusions you are making are valid generally.

²⁵Those of you who are experienced programmers, or just curious, may wonder if there is a way of passing the method 'f' to the integration method in the same way that a variable can be passed to a method through the argument list. Some computer languages do allow this, but in Java you would normally do this using 'interfaces' which aren't covered in this course, but are discussed in the recommended textbook.

6. Finally, write an additional method which implements the trapezium rule. Compare its convergence rate to that of the midpoint and left-endpoint rectangle rules. Again, you are likely to want to try this with a range of different polynomials or other functions to check whether any trends you are seeing are valid generally. Consider whether there are cases where either of the rectangle rules give more accurate results than the trapezium rule for the same number of sampling points. If not then why not? If so then why?
7. You will need to write a report describing the results of your work above. You should use the template document on Moodle as a starting point for your report. You should use LaTeX for this report unless you are a Natural Sciences or exchange student who has not used LaTeX previously in which case you can use an alternative tool, but your submission should be a .pdf document (Word documents, etc can be saved as a .pdf document easily). Please let us know ahead of time to expect the submission not to be from LaTeX.
8. Submit your report (as a single .pdf file) on Moodle. The report should contain a discussion of the convergence rates of the various integration methods and any tables or plots that support the discussion. You are welcome to extend the scope of the report by exploring other numerical integration methods but *the maximum length of the report is 5 pages*. The marking guidelines for the report are available on Moodle.

Exercise 5.2 _____ (Summative: University Scale, 10%)

1. Now that you have established enough confidence in the program you used in the previous exercise, you can use it on a function which cannot be integrated analytically.
2. Make a copy of your program at this stage and continue to work with the copy in case you still need the original to complete your report.
3. In your new program change the method `f` to represent the function:

$$f(x) = \sin(\ln(1 + x^c))$$

where `c` is an integer. You will need to modify the method `f` so that the value of `c` can be passed to the method as a second argument in its argument list. Notice that the argument of the `sin` function is expected to be in radians and the Java method `Math.sin()` also expects its argument to be in radians so you do not need to implement any kind of conversion from degrees to radians or vice versa. The method in Java which represents the natural logarithm function is `Math.log()`.

4. Test your program with different values of `N` and different values of `c`. You can use the results from this exercise as part of the report from the previous exercise if you wish to.
5. Again, you will probably want to make a copy of your code at this point in case you need to revert back to his old version later.
6. Implement the following logic in the main method in the order given below:
 - A user enters the value of `a`. This is as required in previous parts of the exercise and so you may not need to change anything.

- A user enters the value of b . This is as required in previous parts of the exercise and so you may not need to change anything.
- A user enters the value of c . This is the integer which is used as part of the function in the method f .
- A user enters the value of N . This is as required in previous parts of the exercise and so you may not need to change anything.
- A user enters the value of the *tolerance*, ϵ . See below to understand what this is.
- Your program should carry out the numerical integration using a , b and N in the trapezium rule (only). **Don't** delete the methods you've implemented for the rectangle rule, etc. You'll want these to be in your program when you submit it so that the marker can see them.
- Having evaluated the numerical integration using the trapezium rule once, your program should then carry out the numerical integration again using a , b and $2N$ in the trapezium rule. I.e. the number of sampling points should be doubled.
- Your program should calculate the difference between the two answers just obtained (the trapezium rule with N and the trapezium rule with $2N$).
- If the absolute value of this difference is larger than ϵ then the program should double the value of N again and repeat the numerical integration with the trapezium rule.
- Keep doubling N and repeating the numerical integration with the trapezium rule until convergence is reached (i.e. the difference between the last two answers obtained from the trapezium rule is equal to or less than ϵ). Hint: use a while loop.
- While developing and debugging the code, it might help to print out the results at each iteration so that you can see what is going on.
- It is very important that in the final version of the code (the one you are going to submit) the last line printed by your program should consist of two numbers separated by a space. The two numbers should be the final result of the numerical integration as obtained from the trapezium rule and the final value of N that was used to calculate the numerical integration result.
- For example the last line printed should look something like:

```
1.5321329564702857 256
```

7. After printing out the final result, the program should simply end as normal (i.e. you **don't** want the program to loop back to the top of the main method to ask for new values for a , b , N and ϵ .)
8. Make sure your code is adequately commented and clear to read. You don't need to generate javadoc documentation for this exercise but you **should** include javadoc-style comments for all methods so that the human reading the code can understand what each method does.
9. Submit your program (just the final .java file) to Moodle. The program functionality will be automatically marked by a robot. Humans will check the code commenting and style.

The mark allocation for this exercise:

- program functionality (robot marked): 60%
- quality of the code (human marked): 40%

Pitfalls and tips

Here are some (but not all) of the common problems associated with the program for this week's exercises.

- Data types — make sure your choice of `int` or `double` is appropriate for the variable. Remember the effects of integer division!
- Loop counters should always be `int` or `long` to avoid rounding errors.
- Don't forget to initialise or re-initialise variables whenever necessary.

Here are some tips for choosing what functions to work with when experimenting with numerical integration techniques and writing the report.

- Understand the behaviour of the function before trying to integrate it.
- Choose N so that you obtain a reasonably accurate result first. Vary N to understand the accuracy quantitatively.
- The function must not be sharply concentrated in peak(s) (i.e. f'' shouldn't be large) or you will wait a long time to get accurate results.

J Multiple classes

J.1 Static classes

For the last couple of weeks the exercises have required adding extra methods to the basic program structure in addition to the `main` method, but it is also possible to add extra *classes* too. Up until now we've had one *class*, which was declared `public`. Since a *class* which is declared `public` must always have the same name as the file in which it is contained, any file can only contain one `public class` declaration. To include an extra `public class` declaration in your program you'll need to put the declaration in a separate file which will have the name of this second *class*. These additional *classes* do not need to contain a *main* method. A program can use as many classes as you like and include as many extra *Java* source files as needed.

One advantage of using additional *classes*, is that they allow you to take *methods* that you've already developed for one program and use them in another program.

By writing these *methods* in a separate *class* (in a separate file) any program may make use of them. To invoke a *method* which is in another *class* you just add the name of the *class* containing the *method*, to the beginning of the *method* name, i.e.

```
ClassName.methodName (argument) ;
```

Here's a short example, based on the `printSquare` example from section G, page 55. The following code should be in a file called 'MainClass.java'.

```
/**
 * Demonstrate the use of the printSquare method in ExtraClass.
 */
public class MainClass
{
    public static void main (String [] args)
    {
        double x = 3.0;
        ExtraClass.printSquare(x); // print the square of x
    }
}
```

The above code uses the method `printSquare` which is in the separate file, 'ExtraClass.java'. The contents of 'ExtraClass.java' is

```
/**
 * An example of a class for use by code in
 * other classes. This one just contains a
 * method to calculate the square of a number.
 */
public class ExtraClass
{
    /** Prints the square of a number
     *
     * @param y the value to be squared
     */
    public static void printSquare(double y)
    {
        System.out.println(y*y);
        return;
    }
}
```

Both of these files can be downloaded from Moodle. Compile and run the code to make sure you understand the idea of using multiple classes before we move on to the much larger topic of using *non-static classes*. When compiling the file containing the `main` method, the compiler will automatically detect if any other files/classes are used by the program and compile these too.

Splitting code up like this means that any other program you write may also make use of the method `printSquare` without you needing to retype it. Obviously if you are going to do this, it pays to make your code as general as possible and to comment it as fully as possible, to avoid problems if you end up re-using it some time after it was originally written. Javadoc-style comments are especially useful. If the comments are well-written you don't need to understand the details of the code itself - indeed you don't have to see the code at all (which is the point of *javadoc*).

Splitting up programs over several files also makes it easier for more than one person to work on a project at the same time. This is useful if you are programming in a team as is common in some areas of physics such as particle physics. One person can be busy writing a `main` method, and as long as he/she knows what the *class* names, *method* names and their *arguments* are, they can make use of them in the `main` method. Another programmer can worry about the details of how to perform the necessary calculations etc. within these additional *methods*.

You've probably noticed by now that the form of the call to a *method* of another *class* looks quite familiar. This is because you have been using it already, e.g. `System.out.println()` and the mathematical functions such as `Math.sin(x)`. These *classes* and their *methods* are a standard part of the language and are available for you to call in any Java program²⁶.

²⁶In the case of these pre-existing *classes* (`Math`, `System`, etc) you also need to use an import statement such as `import java.lang.Math;` as the classes are in *packages*. You can also quite easily group together the *classes* you write into your own *packages* but this is beyond the syllabus of this course.

J.2 Non-static classes and objects

Reading for Week 6-10 Liang: Chapters 9 – 10

Introduction to object orientation (OO)

By now you have mastered all the basics that enable you to write programs that will perform all the calculations you need. However we have not yet touched upon the style of programming that Java was really designed for, *object orientation*, or *OO*. There can be great advantages in writing your programs in an *object-oriented* way. This section should serve as an introduction to *OO* programming in Java. If after this you wish to find out more, try chapter 9 – 10 of [3], chapter 2 of [5], and chapter 8 of [4].

This section will require you to start thinking about classes in a different way. However, despite the different style, you will still need the Java covered previously in the course. Also bear in mind that, regardless of how a program is written, when running the program the computer always starts with the *main* method in the *public class* contained in the file you are running.

In the following sections, we will start by using objects, and then move on to designing and creating our own objects.

J.3 What is an Object?

An *object* is a generalisation of the *primitive data types* that you are already familiar with. It's a way in which we can represent a collection of data and in addition provide some methods that can manipulate the data. In very general terms an *object* can represent any “thing” (for example a ball, a book, a gravitational field, a student, ...) that has certain properties (a ball may have mass, colour, size, location), and actions (it can move, change size, rotate, etc.). In object-oriented programming the “thing” is represented by the *object*.

Specifically an *object* will contain

1. member variables (also known as data members or fields) which represent the data associated with the object.
2. methods that access and/or manipulate the data members in predefined ways.

Hopefully, the idea of an *object* will become clearer after reading and working on the following examples.

In Java, a *class* can be thought of as a blueprint or recipe from which *objects* are created. If you had a recipe for a cake then in principle you could make as many cakes as you want. Likewise, once you have a *class* you can create as many *objects* from the *class* as you want. A specific *object* is called an *instance of the class*.

We have been using *objects* already in this course. For example, if you create a variable of type `String` then you are actually creating an instance of the `String` class. The `String` class contains a data member which is an array of type `char` where the contents of the `String` is stored, and it also contains various *methods* which let you manipulate `String` objects.

Objects are similar to primitive data types or arrays in some ways. Like arrays they are *reference data types*, and so may be passed to and from *methods* in the same ways as arrays. The major difference is that the programmer can define the form that the *object* takes, such as how many values it has, and can decide

what the rules are for manipulating the *object*. To see how this works, read carefully through the following example.

J.4 Creating an object

To recap, before creating any *objects*, we must tell the computer how to make the *objects*. We do this by giving the computer a blueprint or recipe that describes the *object*, i.e. we must write a *class*. Previously you have been using *classes* really only as a way of grouping together *methods* in a program, but they can be much more than this.

A *class* that act as blueprint for *objects* behaves differently from the ones we have used earlier in this section. This difference occurs when we omit the word `static` when declaring a member variable and/or method²⁷.

Let's consider an example. It is possible to imagine many scenarios when you might want to perform calculations using complex numbers. *Java* does not contain a *primitive data type* or a *class* which represents complex numbers, so we're going to have to make our own²⁸. Once we have written the *class* we will be able to create objects which represent complex numbers. Here is a class `Complex` that defines a complex number.

```
/**
 * Purpose: A very basic class which represents complex numbers.
 * @version 1.0
 */
public class Complex
{
    // two non-static member variables
    // representing the real and imaginary parts of a complex number
    public double realpart;
    public double imagpart;
}
```

An *object* created from this class can be given a name of your choosing just in the same way as with any variable you've created previously. To create an *object* of type `Complex` you could write

```
Complex a;           // creates a reference "a" (c.f. arrays)
a = new Complex();   // creates a new object of class Complex
                    // and sets "a" to refer to it

// often this is all written on one line
Complex b = new Complex();
```

in a method of some other class.

Each *object* created from this class will have its own variable called `realpart` and its own variable called `imagpart`. The contents of each *object* will have different scopes and will be stored in a separate space in the computer's memory. You can refer to these in your program by adding `.realpart` or

²⁷The main method for the program should still always be declared `public static void` and should be in a separate `public class` that does **not** contain **any** non-static *methods* or non-static member variables.

²⁸It's worth noting that there is a complex number *class* and other useful maths *classes* in the Apache Commons Mathematics Library [8].

.imagpart²⁹ to the end of the *object* name. Thus a program that created a *Complex object*, gave it values and printed them to the screen could look like this.

```
/**
 * Purpose: Create and Print a Complex Number
 */

public class ComplexDemo
{
    public static void main(String[] args)
    {
        // create an object of type Complex
        Complex a = new Complex();

        // give object values
        a.realpart = 1.2;
        a.imagpart = -5.9;

        // print to screen
        System.out.println(a.realpart + " + " + a.imagpart + "i");
    }
}
```

As with any programs that involve more than one *class* (and therefore more than one file), compiling the file that contains the `main` method will automatically compile any other files needed for the program to work.

As we've seen previously, variables that are declared inside a *class* but outside of any *methods* are known as *member variables* or *fields*. If a *member variable* is declared as being `static` then the variable is called a *class variable* and only one copy of the variable will be made in the computer's memory; it is shared by all *objects* made from the same *class*. If a *member variable* is not declared as being `static` then it is called an *instance variable* and each *object* will get its own copy of the variable.

²⁹You can only access these variables if they are declared to be `public` as they are in this example. It is more common to make these variables `private` and to access them via methods. We will see this later.

Exercise 6.1 _____ (Formative: No marks)

Download the Complex class given above (from Moodle) and save the file as 'Complex.java'. Write your own short program, in a separate file (main method only), similar to 'ComplexDemo' that creates two Complex objects and gives them values. Next your program should multiply the two complex numbers together by referring to the real and imaginary parts. For example, if you've created a complex number *object* which you've named *a* then you'd refer to the real and imaginary parts of the number using *a.realpart* and *a.imagpart* respectively. Finally your program should print out the result.

For example here is some code which adds together two complex numbers.

```
public class AddComplex
{
    public static void main( String [] args )
    {
        Complex u = new Complex ();
        Complex v = new Complex ();
        u.realpart = 4; u.imagpart = 3;
        v.realpart = 2; v.imagpart = -2;

        Complex z = new Complex ();
        z.realpart = u.realpart + v.realpart;
        z.imagpart = u.imagpart + v.imagpart;

        System.out.println("u=" + u.realpart + "+" + u.imagpart + "i");
        System.out.println("v=" + v.realpart + "+" + v.imagpart + "i");
        System.out.println("u+v=" + z.realpart + "+" + z.imagpart + "i");
    }
}
```

This kind of approach may seem a very long-winded way of adding or multiplying two complex numbers, and indeed it is. Fortunately, using *objects* there is a simpler way to do this using *methods* that is considerably more versatile. This is discussed in the next section.

J.5 Static & Non-static methods

A `static` method is one that is shared between all members of a class. `Non-static` methods act directly on a single instance of an object and often modify the object's data members.

The idea of a *method* is that it gives an object some capability. For example, the rules of addition, multiplication, etc of complex numbers differ from the rules of addition, multiplication, etc for real numbers or for matrices or for vectors. *Java* has no idea how to add or multiply complex numbers unless you tell it how to do so. You can't just write

```
Complex u = new Complex();
Complex v = new Complex();
Complex z = new Complex();

z=u+v;
```

as *Java* does not understand how the '+' operator relates to *objects* of *type* `Complex` even if you do. It would therefore be useful to be able to define a *method* which introduces the concept of addition into the `Complex` class.

Below, a simple (`static`) method for complex number addition has been added to the `Complex` class which we used previously.

```
public class Complex
{
    public double realpart;
    public double imagpart;

    /**
     * This method adds two complex numbers and returns the result.
     *
     * @param z the first complex number to add
     * @param w the second complex number to add
     * @return the result of z+w
     */
    public static Complex add(Complex z, Complex w)
    {
        Complex sum = new Complex();
        sum.realpart = z.realpart + w.realpart;
        sum.imagpart = z.imagpart + w.imagpart;
        return sum;
    }
}
```

This first *method* `add` is similar in form to those you are already used to writing. When in a program it is necessary to add two `Complex` *objects* together, the *method* will be invoked by code like this:

```
// where a, b and c are each objects of the class Complex
c = Complex.add(a,b); // add a and b, result stored in c
                       // (values of a and b remain unaltered)
```

as you may expect from having used methods in other classes in Section G.

Here's another *method* called `increaseBy` which can also be used to carry out addition of two complex numbers. Note that this *method* is **not** declared as *static*. This *non-static method* also goes inside the `Complex` class.

```
// method to add another complex number object to this complex number object
public void increaseBy(Complex z)
{
    realpart += z.realpart;
    imagpart += z.imagpart;
    return;
}
```

To invoke a *method* like `increaseBy` which *isn't* declared as *static* requires a slightly different syntax from the one we've used before. Rather than adding the *method* name to the *class* name as in `Complex.add(...)` the *method* name needs to be added on to the end of the *object* name. For example, if we've created two objects, `a` and `b`, from the `Complex` class then we could use the `increaseBy` method to increment the values stored in object `a` like this

```
a.increaseBy(b);           // add b to a (result stored in a)
```

or we could use the same method to increment the values stored in object `b` like this

```
b.increaseBy(a);           // add a to b (result stored in b)
```

You need to specify which *object* you want the *method* to be applied to because `increaseBy` uses `realpart` and `imagpart` which can take different values for each *object* created from the *class*.

Note that (unlike `add`) the method `increaseBy` acts on one of the *objects* directly and changes the contents of the *object*. In the `increaseBy` method the variables `realpart` and `imagpart` refer to the real part and imaginary part of the `Complex` *object* on which the *method* acts. However, `z.realpart` and `z.imagpart` are the real and imaginary parts of the `Complex` *object* which is passed as an argument to the method. So using `increaseBy` in this way is analogous to

```
int a=3;
int b=5;
a = a+b; // the value of a is changed
```

or

```
int a=3;
int b=5;
b = b+a; // the value of b is changed
```

where we've used the familiar `int` data type to illustrate the principle. Using the static `add` *method* we introduced earlier is more like the following.

```
int a=3;
int b=5;
int c;

c = a+b; // The values of both a and b are unchanged
```

Methods like `increaseBy` in the example above are known as *non-static* and are defined without the `static` keyword because they act on data which belong to each specific *object*. In this case, the data are the variables `realpart` and `imagpart` which clearly have different, independent values in each *object* of type `Complex`.

J.6 Constructor methods

Constructor methods are a particular type of *non-static* method. There is often more than one *constructor* in a *class* and they are always given the same name as the *class* itself. They are automatically invoked when an *object* is first created using the keyword `new`. They are declared slightly differently from usual *methods* as you do not specify the return type of a *constructor* whereas with a normal *method* you always need to specify the return type even if it is `void`. The *constructors* should be the first methods in the class. A *constructor* method is often used to automatically initialise an *object's* data members to hold the values of your choice. Here is an example:

```
public Complex(double x, double y)
{
    // initialise variables to values specified
    realpart = x;
    imagpart = y;
    return;
}
```

Note that the name of the constructor is also `Complex` (the name of the *class*). If this constructor is in the `Complex` *class* then we can now create `Complex` *objects* with any initial values we want, e.g.:

```
Complex z = new Complex(1.2, -5.9);
```

We have used the syntax `new Complex()` before to create `Complex` *objects* (instances of the `Complex` *class*). The `new` keyword creates an *object* from a *class* and invokes a *constructor* which *initialises* the member variables in this new *object*. If there is no *constructor* supplied inside the *class* then *Java* provides a default *constructor* which takes no arguments. This default *constructor* sets all the numerical member variables in the *object* to be zero. However, if you write your own *constructor* like the one above, this default *constructor* will not be invoked. Therefore you could add your own *constructor* with no arguments in its argument list to a *class* if you wish and it will act as a default *constructor* replacing the one that *Java* would otherwise provide. It is generally considered to be good practice to create your own default constructor so that you control the initialisation behaviour of *objects* created from your *classes* ³⁰.

³⁰If you do not want to use the default *constructor* directly then you can make it `private` so it is not available to programs. See the following sections.

It is possible to write more than one *constructor* for a *class* provided that each of them takes a different number/type of *arguments* (i.e. as long as they have different *signatures*). For example, in addition to the `public Complex(double x, double y)` above, the *Complex class* could also contain:

```
// default constructor
public Complex()
{
    realpart = 0.0;
    imagpart = 0.0;
    return;
}
```

J.7 Using private member variables

So far you have been using *public member variables*. The word `public` makes the variables visible to the other *classes* not just inside the *class* itself. Outside of the *class* in which they are declared the variables can be referred to using the `a.realpart` syntax. This is not generally considered a good way of designing a *class*. To see why, consider a situation in which you wanted to change the way complex numbers were stored inside the *Complex class* from Cartesian notation to polar notation. You could replace the *public variables* `realpart` and `imagpart` with two new variables `argument` and `modulus` inside the *Complex class*. Now you have a problem because anywhere in any other program that *Complex* objects have been used, you will have to change them to use the `argument` and `modulus` variables instead of the `realpart` and `imagpart` variables. This could be a lot of work.

It would be much better to somehow hide all of the internal workings of the *Complex class*, so that any alterations to the *class* did not require changes to be made to any of the other *classes* which create and use *Complex objects*.

This is possible if you declare the *member variables* of your *class* to be `private` rather than `public`. In addition you'll need extra *methods* that can be invoked to find out the values of these *variables*. *Member variables* that are declared as `private` can not be referred to from other classes, they are only visible within their own *class*. It is considered better programming practice to use `private` rather than `public member variables`, and you should aim to do this in the remainder of the course. Here is the *Complex class* rewritten using only `private member variables`.

```

/**
 * A class from which objects might be created, which behave like
 * complex numbers. Private variables are used to demonstrate good
 * programming practice.
 *
 *
 * @author Ian Bailey
 * @author Iain A. Bertram
 * @author Roger Jones
 * @version v2
 */

public class Complex
{
    // private member variables
    private double realpart;
    private double imagpart;

    /**
     * default constructor, real and imaginary parts
     * :are initialised to zero.
     *
     * (0 + 0i)
     */
    public Complex()
    {
        realpart = 0;
        imagpart = 0;
        return;
    }

    /**
     * Constructor that sets the complex number to (x + yi)
     *
     * @param x the real part of the complex number
     * @param y the imaginary part of the complex number
     */
    public Complex(double x, double y)
    {
        realpart = x;
        imagpart = y;
        return;
    }

    /**
     * method to find out the value of the real part of the complex number
     * @return the real part of the complex number

```



```

    */
    public double getReal()
    {
        return realpart;
    }

    /**
     * method to find out value of the imaginary part of the complex number
     * @return the imaginary component of the complex number
     */
    public double getImag()
    {
        return imagpart;
    }

    /**
     * method to add complex number "z" onto this complex number
     *
     * complex number is increased by  $z=(c+di)$  so that the new
     * number is given by  $((a+c)+(b+d)i)$ .
     *
     * @param z increase the complex number by z
     */
    public void increaseBy(Complex z)
    {
        realpart += z.getReal();
        imagpart += z.getImag();
        return;
    }

    /**
     * static method to add together two complex numbers and return
     * the result,  $z = (a+bi)$ ,  $w = (c+di)$  then return  $y =$ 
     *  $((a+c)+(b+d)i)$ 
     *
     * @param z one of the numbers to be added
     * @param w one of the numbers to be added
     * @return the sum
     */
    public static Complex add(Complex z, Complex w)
    {
        Complex sum = new Complex();
        sum.realpart = z.getReal() + w.getReal();
        sum.imagpart = z.getImag() + w.getImag();
        return sum;
    }

    /**

```

```

    * method to print out in usual complex number form
    * print out a + bi
    */
    public void print()
    {
        System.out.print(realpart);
        if (imagpart < 0)
        {
            System.out.print(" - " + (-1*imagpart) + "i");
        }
        else if (imagpart > 0)
        {
            System.out.print(" + " + imagpart + "i");
        }
        System.out.println("");
        return;
    }
}

```

See how the values of `realpart` and `imagpart` may now be obtained by calling the *methods* `getReal` and `getImag`. This type of method, which gives information on the values of the member variables are sometimes called *accessors* or *getters*. Any calculations etc. that might need to be done with complex numbers should be achieved by writing static methods such as `add` or non-static methods like `increaseBy`. Both types of method are useful, although if there is a choice between the two then using a non-static method is often preferable. This class could be extended and used as in the following example exercise.

Exercise 6.2 _____ (Formative: No marks)

The `Complex` class listed above uses `private` member variables, constructors, accessors, the `increaseBy` method and the `print` method.

- (i) Write a program, in a separate class, that makes use of the `Complex` class to create two `Complex` objects, u and v , gives them values, adds v to u using the *non-static* `increaseBy` method and prints the new value of u to the screen. When you have checked that it works, Use it to calculate the result of $(4 + 3i) + (2 - 7i)$.
- (ii) Now adapt 'Complex.java' by adding a new *non-static* method that turns a complex number into its complex conjugate. Test it thoroughly by getting your program to check all cases for the original number: imaginary part positive, imaginary part zero and imaginary part negative. Show the results for $(4 + 3i)$, $(2 - 7i)$, and 2.
- (iii) Add a new *static* method that calculates the product of two complex numbers. Test it thoroughly by getting your program to multiply several different complex numbers. Show the results for multiplying $(4 + 3i)$ and $(2 - 7i)$.
- (iv) Finally, comment the new methods in `Complex.java` using the javadoc style.
- (v) Written examples of the solutions to this exercise are given below.

An example of code for a *non-static method* that turns a complex number into its complex conjugate and another *method* that calculates the product of two complex numbers is given here.

```
/**
 * Transform a complex number into its conjugate:
 * a +b<b>i</b> which
 * is a - b <b>i</b>
 *
 */
public void conjugate(){
    imagpart *= -1;
    return;
}

/**
 * static method to return the product of two complex numbers
 * the result, z = (a+bi), w = (c+di) then return y =
 * ((a*c - b*d) +(a*d + b*c)i)
 *
 * @param z one of the numbers to be multiplied
 * @param w one of the numbers to be multiplied
 * @return the calculated product
 */
public static Complex product(Complex z, Complex w){
    Complex product = new Complex();
    product.realpart = z.getReal()*w.getReal() - z.getImag()*w.getImag();
    product.imagpart = z.getReal()*w.getImag() + z.getImag()*w.getReal();
    return product;
}
```

An example of the code that makes use of the Complex class:

```
public class ComplexDemo {  
  
    public static void main(String[] arg){  
  
        Complex u = new Complex(4, 3); // u = (4 + 3i)  
        Complex v = new Complex(2, -7); // v = (2 - 7i)  
  
        System.out.print(" u = ");  
        u.print();  
        System.out.print(" v = ");  
        v.print();  
  
        u.increaseBy(v);  
        System.out.print("\n u + v = ");  
        u.print();  
  
        Complex a = new Complex(4,3);  
        System.out.print("\n\nThe complex Conjugate of ");  
        a.print();  
        a.conjugate();  
        System.out.print(" is ");  
        a.print();  
  
        Complex b = new Complex(2,-7);  
        System.out.print("The complex Conjugate of ");  
        b.print();  
        b.conjugate();  
        System.out.print(" is ");  
        b.print();  
  
        Complex c = new Complex(2,0);  
        System.out.print("The complex Conjugate of ");  
        c.print();  
        c.conjugate();  
        System.out.print(" is ");  
        c.print();  
    }  
}
```

Inheritance and Polymorphism

This section has only scratched the surface of *object-oriented* programming. Another important aspect of *OO* is *inheritance* which leads to the concept of *polymorphism*, but these are beyond the scope of this introductory Java course. It is a large subject which encompasses both the design and implementation of software. You are encouraged to read more about it using the references in the bibliography if you are interested. It is also covered in detail in the optional PHYS389 module.

Exercise 6.3 _____ (Formative: 10 marks)

- Complete the quiz on Moodle which covers the material for the whole section on using multiple classes in your programs.
-

K Designing and Implementing Classes

The last section looked at the attributes of *classes* and *methods*, and you tried some simple examples to illustrate the point. The remaining exercises in the notes are related to designing classes that can be used in a simulation of the motion of particles under the influence of gravitational fields. This can become quite complicated, so it is important that before you start coding you can think about the design and describe in detail the variables and methods that are required by a class and whether they should be declared as being `static` or not, and whether should be declared as `public` or `private`. Getting the balance right between designing a class that is too general or one that is too specific takes experience, so it helps us to help you if you can show us your design.

K.1 Program & Class Design

The first step in designing a program is to describe what it has to do, this is often called a *Use Case* or *Requirements Capture*.

Your *use case* should break the problem down into smaller parts, identify all of the entities (*objects*) that will be required and the way they will be modified or changed (*methods*). You can then make use of simple diagrams or flow-charts to illustrate the way the *objects* will be used.

Once you have identified all of the classes you may find it useful to create a *Class-Relationship* diagram. The standard components of a class-relationship diagram are given in Fig. K.17. The use of these diagrams will be illustrated in the following examples.

If we want to model a car ³¹ then we have to describe the properties of a car and the things it can do. At a very simple level a car has one motor, and more than one wheel (the number of wheels is not set). We can also say that the motor and wheels belong to a car. This relationship is illustrated in Fig. K.18. This design is very simple and does not include a list of the `methods` that can be used.

Another, more sophisticated example, is a cash machine (ATM). To model this we need to consider several components:

- Keypad (for entering information)
- Deposit Slot (for depositing cash)
- Cash Dispenser
- Screen
- Bank Database (that contains information about accounts and users)
- Accounts
- Withdrawal (is a object that combines many of the above, to complete a specific action)

The resulting diagram for the ATM machine is given in Fig. K.19. At this point we have a visual description of the classes required for a simulation of the cash machine but have not described the actual data types or methods that we will require to implement each of the classes.

³¹For example, write some software to control the brake system etc.

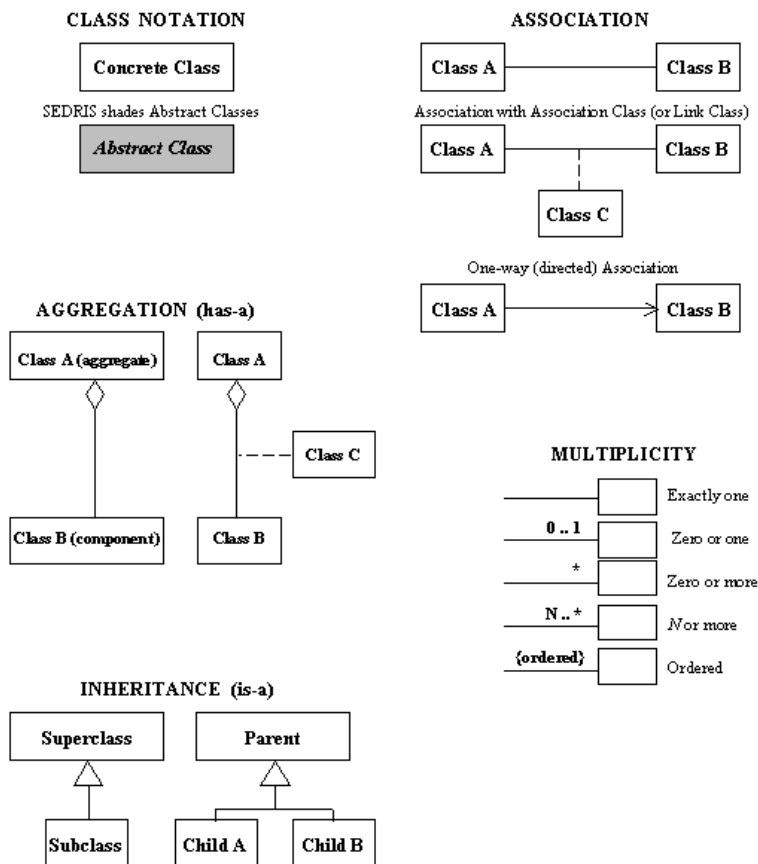


Figure K.17: Standard components of UML class relationship diagrams

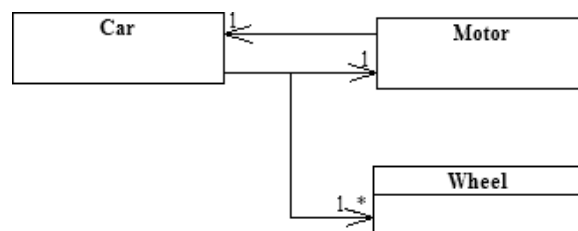


Figure K.18: Illustrative class-relationship diagram for a very simple car.

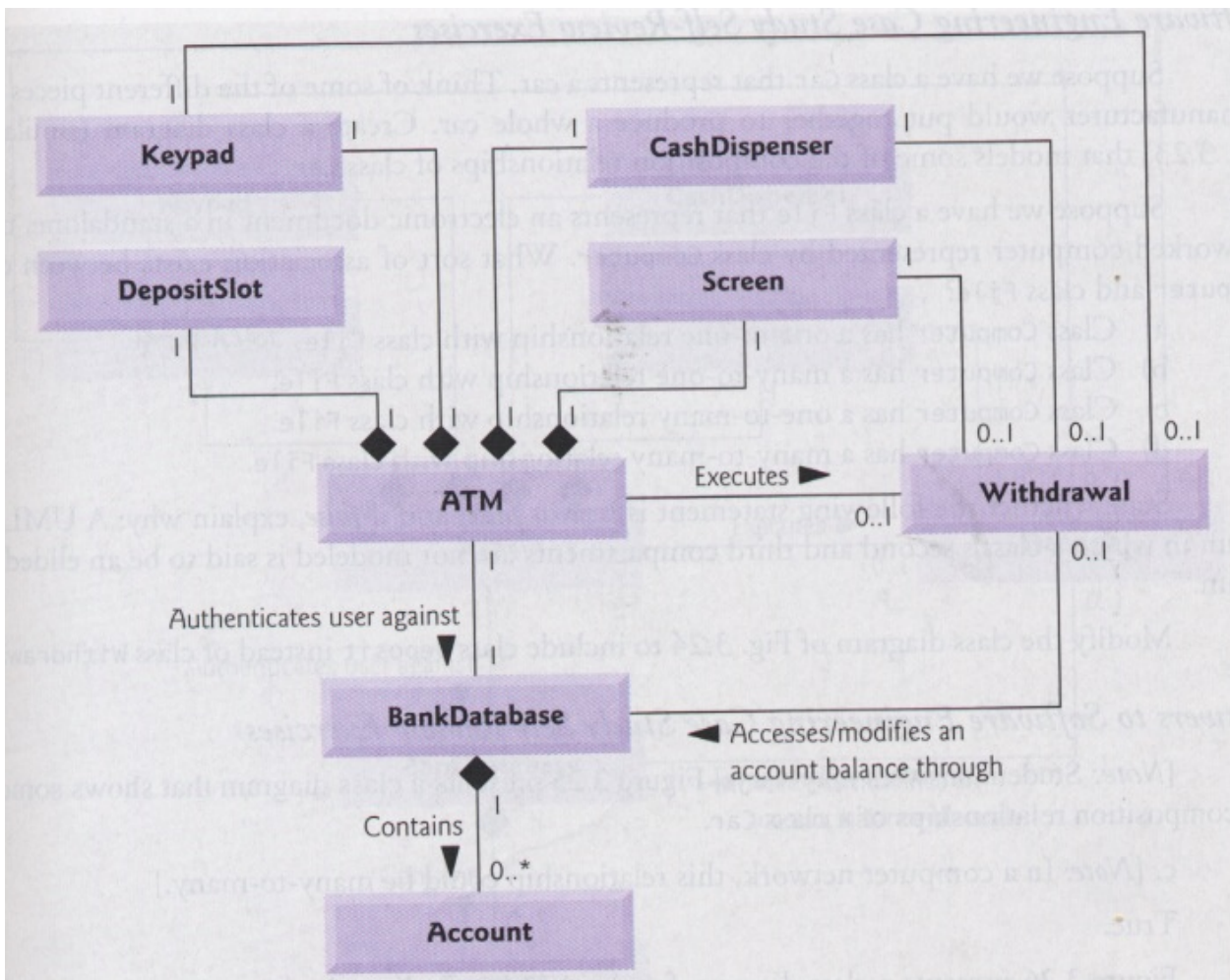


Figure K.19: *Class-Relationships* diagram for an ATM.

K.2 Class Diagrams

The next step in carrying out a formal class design is the creation of a class diagram. This requires you to fill in each of the boxes in the class-relationship diagram with the *data members* and *methods* that belong to each class. Depending on how complicated your final project becomes for this course you may or may not need a class-relationship diagram, but a basic class diagram is very useful and is similar to what is produced by the 'javadoc' command.

It is easiest to illustrate this stage of the design process by considering the `Complex` class used in the previous section. A use case for this class could be written so that it can do the following:

- set the values of a complex number
- return the real and imaginary parts of a complex number
- multiply two complex numbers
- calculate the ratio of two complex numbers
- print complex numbers

We can then produce a class diagram to describe the required data types and methods, this diagram is given in Table 2. The symbols in the diagram are as follows: "+" is a public variable or method, "-" is private and can only be accessed by methods that belong to the class, "#" represents a protected variable or method, "<<>>" signifies a constructor of a class and an underlined method or variable is *static*. The standard format of a class diagram is to give the name of the class at the top, followed by the variables or data contained within the class, and then the methods. The constructors are typically listed first. Each method is listed by its name. The arguments and their type are given in brackets, followed by the method's return type.

Complex
-realpart : double -imagpart : double
<<constructor>> Complex() <<constructor>> Complex(x: double, y: double) +setComplex(x: double, y: double): void +getReal(): double +getImag(): double +increaseBy(z: Complex): void <i>add(z: Complex, w: Complex) Complex</i> +conjugate(): void +print(): void

Table 2: Complex Class Diagram

In addition to the class diagram we need a complete description of the class as well, including tests to check that the class is working as it should. For the example Complex class we could write:

Complex is a class representing complex numbers given by $(a + bi)$ where a and b are stored in two doubles `realpart` and `imagpart`.

Constructors:

Complex(): is the default constructor where `realpart=0` and `imagpart=0`.

Check that the object of type complex is created and its components are set to zero.

Complex(x: double, y: double): is a constructor where `realpart=x` and `imagpart=y`.

Check that the complex is created and its components are set to x and y for the cases $x = 1, y = 1$ and $x = 0, y = 1$ and $x = -1, y = 1$ and $x = -1, y = 0$

Methods:

+setComplex(x: double, y: double): void: sets the complex number so that `realpart=x` and `imagpart=y`.
Create a complex number $5 + 3i$ and set it to be $-4 + 5i$ and check that it is changed correctly.

+getReal(): double: return `realpart`

+getImag(): double: return `imagpart`

+increaseBy(z: Complex): void: Let $z = a + bi$ and $c = \text{realpart}$ and $d = \text{imagpart}$ then the method will modify the complex number such that `realpart = c + a` and `imagpart = d + b`. *Check this with $1 + 1i$ and $z = 0, z = 1, z = i, z = 1 - 1i, z = -1 + 1i$*

+add(z: Complex, w: Complex): Complex: return $(a + c) + (b + d)i$ where $z = a + bi$ and $w = c + di$.
(Static Method)

Check this with $z = 1 + 1i$ and $w = 0, w = 1, w = i, w = 1 - 1i, w = -1 + 1i$

+conjugate(): void: sets `imagpart = -imagpart`.

Check this by setting the complex number equal to several different values including zero and i and checking that the complex number has been correctly set to be the complex conjugate.

+print(): void: print the Complex number with the format $a + bi$ to standard output.

Set the complex number to several different values including zero and check that the output is correct.

Note: The tests that we have constructed should attempt to cover all possible kinds of occurrences with the class. I.e. we need to check standard and non-standard cases such as the complex number being zero or

purely real or purely imaginary.

K.3 Vectors

In almost all areas of physics you will be required to make use of vectors. In this section we will be designing a class to represent standard 3-dimensional vectors:

$$\vec{A} = A_x \hat{i} + A_y \hat{j} + A_z \hat{k} \quad (24)$$

which can be used to represent quantities such as position, momentum, vector fields, etc. The vectors will be easily extendible to n -dimensions. A partially completed design document is provided for the class.

PhysicsVector
-N final array size (=3) -components: double[N]
<<constructor>> PhysicsVector() <<constructor>> PhysicsVector(x: double, y: double) <<constructor>> PhysicsVector(x: double, y: double, z: double) <<constructor>> PhysicsVector(x: double[]) <<constructor>> PhysicsVector(v: PhysicsVector) +setVector(x: double, y: double, z: double): void +setVector(x: double, y: double): void +setVector(v: PhysicsVector): void +increaseBy(v: PhysicsVector): void +decreaseBy(v: PhysicsVector): void +magnitude(): double +getX(): double +getY(): double +getZ(): double +getUnitVector(): PhysicsVector +scale(x: double): void +scale(x: double, v: PhysicsVector): PhysicsVector +print(): void +print2D(): void +returnString(): String +returnSimpleString(): String +return2DString(): String +returnSimple2DString(): String +add(v: PhysicsVector, u: PhysicsVector): PhysicsVector +subtract(v: PhysicsVector, u: PhysicsVector): PhysicsVector +dot(v: PhysicsVector, u: PhysicsVector): double +vectorProduct (v: PhysicsVector, u: PhysicsVector): PhysicsVector

Table 3: PhysicsVector Class Diagram

The following describes each of the methods. Some quick (non-exhaustive) tests are given in italics after each method.

PhysicsVector is a class designed to represent 3-D vectors which will be given by $x\hat{i} + y\hat{j} + z\hat{k}$ where x , y and z are stored in an array: `components[i]`, $i=0 \dots 2$. I.e. the vectors are made up of real numbers. The following methods are required.

Constructors:

PhysicsVector(): is the default constructor where `component[i] = 0`.

Check that a vector is created and can be printed out

PhysicsVector(x: double, y: double) is the constructor where `component[0] = x`, `component[1] = y` and `component[2] = 0`

create a (3,4,0) vector and print it out to see if it works.

PhysicsVector(x: double, y: double, z: double): is the constructor where `component[0] = x`, `component[1] = y` and `component[2] = z`

create a (3,4,5) vector and print out to see if it works.

PhysicsVector(x: double[]): is the constructor where `component[0] = x[0]`, `component[1] = x[1]` and `component[2] = x[2]`

create a (3,4,5) vector from an array and print out to see if it works.

PhysicsVector(v: PhysicsVector): where the new vector is created with `component[i] = v.component[i]` where `i = 0 .. 2`

This type of constructor is called a copy constructor.

create a vector from a (2,1,-1) vector and print it out.

Methods:

+setVector(x: double, y: double, z: double): void: sets the vector so that `component[0] = x`, `component[1] = y` and `component[2] = z`

Create a default (0,0,0) vector and set vector equal something different (2,1,3) and print

+setVector(x: double, y: double): void: sets the vector so that `component[0] = x`, `component[1] = y` and `component[2] = 0`

Create a default (0,0,0) vector and then set vector equal to (2,1,3) and print.

+setVector(v: PhysicsVector): void: sets the vector so that `component[i] = v.component[i]` where `i = 0 .. 2`

create a default vector (0,0,0) and a non default vector (2,1,3) and set the original vector to be the same as the new one.

+increaseBy(v: PhysicsVector): void: the original vector has vector \vec{v} added to it so that `component[i] += v.component[i]` where `i = 0 .. 2`

add (3,4,1) to (-3,4,1) and check we obtain (0,8,2)

+decreaseBy(v: PhysicsVector): void: the original vector has vector \vec{v} subtracted from it so that `component[i] -= v.component[i]` where `i = 0 .. 2`

subtract (-3,4,2) from (3,4,0) and check we obtain (6,0,-2).

+magnitude(): double: returns the length of the vector. Given by $\sqrt{\sum_0^2 \text{component}[i]^2}$.

Check that the magnitude of a (3,4,0) vector is 5, also check with (0,1,0) vector and (0,0,0) vector to obtain 1 and 0 respectively.

+getX(): double: returns the value contained in `component[0]`.

check it does return the x-component of (1,2,3) vector

+getY(): double: returns the value contained in `component[1]`.

check it does return the y-component of (1,2,3) vector

+getZ(): double: returns the value contained in `component[2]`.

check it does return the z-component of (1,2,3) vector

+getUnitVector(): PhysicsVector: returns a unit vector with magnitude 1 with the direction of the original vector. $xComponent = this.component[i] / this.magnitude()$. Return zero vector if original vector is zero vector.

Check zero vector (0,0), check unit vectors (1,0,0), (0,-1,0). Finally check (3,4,0) vector which should give (0.6,0.8,0)

+scale(x: double): void scales the vector by a scalar, $component[i] *= x$ where $i = 0 \dots 2$.
Multiply (3,4,1) by 10 to obtain (30,40,10)

+scale(x: double, v: PhysicsVector): PhysicsVector scales the vector v by a scalar and returns the result u , $u[i] = v[i]*x$ where $i = 0 \dots 2$.
Multiply (3,4,1) by 10 to obtain (30,40,10)

+print2D(): void: Will print the vector to the screen in the following format: $component[0] \ i + component[1] \ j$. *tested by printing out several 2d vectors including some with negative values*

+print(): void: Will print the vector to the screen in the following format: $component[0] \ i + component[1] \ j + component[2] \ z$. *tested by printing out several vectors including some with negative values*

+returnString(): String: returns a string representing the vector as $component[0] \ i + component[1] \ j + component[2] \ z$. *tested by printing out returned string from several vectors including some with negative values*

+return2DString(): String: returns a string representing the vector as $component[0] \ i + component[1] \ j$. *tested by printing out returned string from several vectors including some with negative values*

+returnSimpleString(): String: returns a string representing the vector as $component[0] \ component[1] \ component[2]$. *tested by printing out returned string from several vectors including some with negative values*

+returnSimple2DString(): String: returns a string representing the vector as $component[0] \ component[1]$. *tested by printing out returned string from several vectors including some with negative values*

+add(v: PhysicsVector, u: PhysicsVector): PhysicsVector: standard vector addition, returns a vector with $component[i] = u.component[i] + v.component[i]$ where $i = 0 \dots 2$.
add vector (1,0,0) to (0,1,0) and get (1,1,0).

+subtract(v: PhysicsVector, u: PhysicsVector): PhysicsVector: standard vector subtraction, returns a vector with $component[i] = v.component[i] - u.component[i]$ where $i = 0 \dots 2$.
subtract vector (1,0,0) from (0,1,0) and get (-1,1,0).

+dot(v: PhysicsVector, u: PhysicsVector): double: standard vector scalar product, returns a double given by $\sum_0^2 u.component[i] \times v.component[i]$
Check that (0,1,0) dot (0,0,1) is zero, check that (0,0,0) dot anything is zero, check (1,2,3) dot (2,3,4) is equal to 20.

+vectorProduct (v: PhysicsVector, u: PhysicsVector):PhysicsVector standard vector (cross) product in 3d. Returns a PhysicsVector with components $(v[2]u[3]-v[3]u[2], v[3]u[1]-v[1]u[3], v[1]u[2]-v[2]u[1])$
As this is part of the exercise, devise your own test for this method.

Exercise 6.4 _____ (Formative: 10 marks)

Download the `PhysicsVector` class from Moodle and complete the following tasks.

The class already partially implements the design in Table 3.

- (i) The following methods are unfinished. Find them in the `PhysicsVector` code and complete them.
 <<constructor>> `PhysicsVector(x: double, y: double, z: double)`
 `+decreaseBy(v: PhysicsVector): void`
 `+dot(v: PhysicsVector, u: PhysicsVector): double`
 - (ii) In addition the following method is completely missing and you should write and add this method to the class
 `+vectorProduct (v: PhysicsVector, u: PhysicsVector):PhysicsVector`
 - (iii) Test that the methods do behave as intended by writing a main method in some other class which uses the `PhysicsVector` class.
 - (iv) Note that some of the *methods* in this *class* use the keyword `this`. When used in a non-static *method*, `this` refers to the current *object*. It isn't strictly necessary for this course, but it is useful.
 - (v) Once it is tested, upload your completed `PhysicsVector` class to Moodle.
-

L Gravity simulation

The final project is to develop a physics simulation in which a physical system evolves with time. An interesting type of system to consider is the motion of one or more particles or extended bodies under the influence of gravitational fields. This is the example that we'll use as the starting point of the final project for this module.

There is plenty of scope to develop the simple gravitational field simulations described here into simulations of more complicated systems and you are welcome to extend your project to involve different physical systems, forces, etc. Initially though you should start by following the steps presented below. You may find that doing this takes all of the available time or you may find that you have plenty of extra time to simulate other systems. Either way you will need to write up your findings in a report as detailed at the end of this section so make sure that you leave enough time for this.

As in the case of numerical integration, most physics simulations will require some amount of approximation which will introduce an error (i.e. an inaccuracy) in the results from the simulation. There are various ways to establish the size of this error:

- Apply the simulation to a system where there is an analytical solution to the equations of motion so that the predictions of the simulation can be directly tested against the known results. This approach has the advantage of allowing you to unambiguously calculate the size of any error in the simulation due to approximations, etc. It is equivalent to applying the numerical integration techniques of section I to a polynomial. However, it does not guarantee that the simulation will have the same size of error when it is applied to a regime for which there is not an analytical solution.
- Apply the simulation to a system for which there is experimental data so that the simulation can be directly tested against reality. In some ways this is of course what physics is all about and so is a good approach. The difficulty here is that experimental data often contains many additional subtle effects which may not be in your simulation so it may not be easy to understand where any discrepancies come from.
- For a range of systems, consider whether the simulation conserves quantities that you would expect to be conserved, e.g. energy. Although conservation of energy, etc does not guarantee that the details of the simulation are accurate, it provides a quick global check of the simulation's properties.

You are going to want to consider carefully how to use these approaches to assess the accuracy of your own simulations.

Let's consider the various steps in developing a general simulation of particle motion in gravitational fields.

L.1 A 'Particle' in The Earth's Gravitational Field

To start off with, we want to simulate the trajectory of a particle in a cannonball-like trajectory close to the surface of the Earth. We need to bear in mind from the start that we are going to want to create a simulation that is easily extendable to the more general case where the acceleration due to gravity is not constant and may depend on other bodies involved in the simulation. It would be easy to forget this and just write a simulation which can only simulate the parabolic trajectories of a particle close to the surface of the Earth, but with a bit of thought we can save a lot of time later on.

For simplicity, our simulation can be considered initially to be in 2 dimensions: the x -direction which is horizontal to the ground, and the y -direction which is perpendicular to the ground. We are of course

inherently here assuming the surface of the Earth can be treated as being approximately flat as long as we consider fairly short trajectories. Later on we can remove this assumption.

In this case we can represent the approximate acceleration by:

$$\vec{a} = g = -9.81\hat{j} \text{ ms}^{-2} \quad (25)$$

To characterise the motion of particles in this uniform gravitational field we will need to be able to calculate the changes in the particle's position \vec{x} , and velocity \vec{v} as a function of time, t .

For many (although not all) physical systems we can write the velocity and position in the form

$$\vec{v}(t) = \int^t \vec{a}(\vec{v}, \vec{x}, t) dt \quad (26)$$

$$\vec{x}(t) = \int^t \vec{v}(\vec{x}, t) dt \quad (27)$$

where $\vec{a}(t)$ is the acceleration of the particle and we have restricted ourselves here to considering a single particle.

In this simple example the acceleration is constant and we can of course just solve the equations analytically to give the basic equations of motion for constant, uniform acceleration (i.e. there is no spatial or temporal dependence). The familiar equations much beloved of A-level syllabuses are:

$$\vec{v} = \vec{v}_0 + \vec{a}t \quad (28)$$

$$\vec{x} = \vec{x}_0 + \vec{v}_0t + \frac{1}{2}\vec{a}t^2 \quad (29)$$

where the initial position and velocity at time $t = 0$ are given by \vec{x}_0 and \vec{v}_0 respectively. As long as the acceleration is constant and uniform, these equations will be accurate and hence our simulation will not be very interesting.

We would instead like an approximate numerical solution to equations 26 and 27 so that we can consider the general case where the acceleration is not constant/uniform and analytical solutions are therefore not always available. In addition, we'd like to track the evolution of the system as time increases, so we don't just want the positions and velocities at a time t , we want to know them at intermediate times too. The simplest approach to this is known as the Euler method (or Euler forward method). It is an iterative algorithm which allows us to calculate the approximate position and velocity at time $t + \Delta t$ given that we know them at the slightly earlier time t . It has the form

$$\vec{v}_{n+1} \approx \vec{v}_n + \vec{a}_n \Delta t, \quad (30)$$

$$\vec{x}_{n+1} \approx \vec{x}_n + \vec{v}_n \Delta t \quad (31)$$

where we assume that the acceleration is approximately constant for the small duration Δt and the label n denotes the start of this time step and $n+1$ the end of this time step. As when we were considering numerical integration methods, this approach is actually just the first couple of terms of a Taylor series expansion and we expect that there will be an error of order $(\Delta t)^2$ each time we apply this iterative formula, so that the cumulative error over a fixed time will be of order Δt . Alternative algorithms are discussed later, but for now the Euler algorithm will suffice.

If we want to consider a slightly more complicated case of motion in the Earth's gravitational field then we can take into account the variation of acceleration due to gravity with distance from the Earth's centre r

using

$$\vec{g} = \frac{-GM_E}{r^2} \hat{r} \quad (32)$$

for values of r greater than the Earth's radius $R_E = 6380$ km. The mass of the Earth is given approximately by $M_E = 5.974 \times 10^{24}$ kg and G is the gravitational constant.

Care must be taken when defining the direction of the field or you could easily end up simulating anti-gravity!

Exercise 7.1 _____ (Summative: 10%)

This exercise forms the first part of your final project, but is due to be submitted earlier than the rest of the project and has clearly-defined goals. If you are an experienced programmer or have done a lot of independent learning of Java then you may realise that the design of the code described in this exercise has some limitations. You can develop a different design if you wish, as long as your code fulfills the requirements specified below.

For this exercise you need to write a simulation of a projectile travelling in the Earth's gravitational field as described in section L.1. The simulation should use the Euler algorithm to approximate the motion of the projectile. Do not use anything other than the Euler algorithm as this stage, even if you know of better algorithms.

Specifically the code should:

- Use a `Scanner` as usual to read in the size of the time step to be used in the simulation from the keyboard. This will be Δt in the Euler algorithm.
- Use the same `Scanner` to read in the x component of the initial velocity of the projectile, where x is horizontal (i.e. tangential to the Earth's surface).
- Use the same `Scanner` to read in the y component of the initial velocity of the projectile, where y is vertical (i.e. radially outwards from the Earth's surface).
- Assume the projectile starts at the origin of the Cartesian coordinate system which is a point on the Earth's surface. The z coordinate can safely be ignored for this simulation.
- Track the motion of the projectile by applying the Euler algorithm so that the position and velocity of the projectile are calculated at each time step in the simulation. You will clearly want to use a loop of some sort to do this.
- Use equation 32 to represent the Earth's gravitational field. You'll need to be careful as the origin of the Cartesian coordinate system in which the projectile is moving is not the same as the origin used in the equation for the gravitational field (which has its origin at the centre of the Earth).
- Stop as soon as the simulated projectile is at a negative vertical position, i.e. $y < 0$. If the length of the trajectory is short then the projectile will now be approximately back on the Earth's surface and if the trajectory is very long then the projectile will stop some way above the Earth's surface (due to the curvature of the Earth).
- Print out to the screen (using `System.out` methods as usual) the final value of the x -coordinate. I.e. print out the maximum horizontal displacement of the projectile. You can have the program print out anything else you like **but the last line printed should be this number without any units or text as it has to be read by the robot.**
- Assume that SI units are used throughout the simulation.
- Use the values of the Earth's mass and radius given in the notes (see above).

Clearly there are various ways of tackling this problem. You can solve it in any way you wish and still get marks for this exercise as this code will be marked only for functionality (unlike your final code submission which will be marked for style, etc). You should of course endeavour as always to make your code clear and easy to read with good commenting, etc in case a human marker has to read through the code to try to

correct any mistakes. In order to maximise your project marks, and in order to be able to re-use this code for the more complicated simulations you are likely to want to write next, you are strongly-recommended to use object-oriented techniques to implement this simulation. However - it is not essential to do this if you find it confusing.

Here is our strong suggestion for how you should proceed unless you know about *inheritance* or other more sophisticated object-oriented programming techniques:

- First start by designing a `Particle` class which you are going to save in a file called `Particle.java`.
- The `Particle` class should represent a point-like object which can move under the influence of forces. In order to represent an object of this kind, the class will need to have data members which represent quantities such as mass, position and velocity. These last two are best represented as `PhysicsVector` objects using the `PhysicsVector` class from week 6.
- In addition to storing the current position and velocity of a particle, the `Particle` class should contain methods which allow the position and velocity to be set or changed or retrieved. In particular there should be a non-static method which ‘updates’ the position and velocity of the particle using the Euler algorithm.
- In order to work, the method which implements the Euler algorithm will have to somehow know what the value of the gravitational field is at the current position of the particle. Putting the gravitational field inside the same class as the particle is not a good design choice, as the field is caused by the Earth not the particle itself.
- We suggest that the you introduce another class called `GravField` which will be in a file called `GravField.java`. This class can represent external gravitational fields such as the one caused by the Earth. It should contain one or more methods which return the acceleration due to gravity as a function of position.
- Your main method should then create a `Particle` object which is going to represent a projectile, and a `GravField` object which is going to represent the Earth’s gravitational field.
- The main method can now use a loop to repeatedly obtain the current value of the gravitational field (as experienced by the projectile) using the methods in the `GravField` class, and then move the projectile (one time step at a time) using the ‘update’ method(s) inside the `Particle` class.

From this point on you are entirely free to develop your simulation in any way you wish but do skip ahead and read the project description thoroughly first! The next two sections detail one way in which you could develop your code with a view to developing a simulation of the Solar System, but you can pick another goal.

L.2 Particle traversing a tunnel through the Earth’s core

In principle, we already know everything we need to know in order to simulate the general motion of a system of particle’s moving under the influence of each other’s gravitational fields. However, rather than

jump immediately to this scenario we could also consider the following. It is clear that the acceleration at the surface of the Earth is given by:

$$\vec{g} = -\frac{GM_E}{R_E^2} \hat{r} \quad (33)$$

We could simulate the path of a projectile dropped from the Earth's surface into a hole that runs through the centre of the Earth in a straight line, i.e. in a radial direction.

Assuming that the projectile has no component of velocity tangential to the Earth's surface, this is a one-dimensional problem (which simplifies matters considerably). The gravitational field strength will depend on the distance of the object from the centre of the Earth. To calculate the gravitational field we need the mass of the part of the Earth contained within a sphere of radius r such that:

$$\frac{m}{m_E} = \frac{\frac{4}{3}\pi r^3}{\frac{4}{3}\pi R_E^3} = \frac{r^3}{R_E^3} \quad (34)$$

So the gravitational field will be given by:

$$\vec{g} = -\frac{Gm}{r^2} \hat{r} = -G \frac{M_E}{r^2} \frac{r^3}{R_E^3} \hat{r} = -G \frac{M_E}{R_E^3} r \hat{r} \quad (35)$$

Hopefully it is clear from this result that we would expect the motion of a projectile dropped into this hole to be simple harmonic motion. You may want to use this as optional test of your code. You will clearly have to add new methods to represent the gravitational field inside the Earth. At this point it is quite likely you will start to see significant limitations of the Euler algorithm. You should read ahead to the discussion of alternative algorithms as you are likely to want to use the Euler-Cromer algorithm. Depending on how much time you have, you may want to skip this simulation entirely and move on with your own project plans, but either way you are likely going to want to replace the Euler algorithm with something more robust.

L.3 Orbiting Massive Particles (E.g Planets)

Consider the case of 'N' massive particles moving under the influence of each others' gravity. The net acceleration of a particle with mass m_i will be given by

$$\vec{g}_i = \sum_{j \neq i}^N \frac{-Gm_j}{r_{ij}^2} \hat{r}_{ij} \quad (36)$$

where \vec{r}_{ij} is the displacement vector from the j^{th} mass to the i^{th} mass. In this case, to determine the field that is affecting a given particle we need to know the locations of all the other particles in our simulation. Notice that the sum of all forces acting on all particles should be zero at all times. This result is just due to Newton's third law in the absence of an external field. Note that if we move one particle before calculating the effect of that particle on all the other particles in our simulation then (unless great care is taken) our code will violate Newton's third law. This is unlikely to be a good idea.

You could now think about redesigning your code so that each `Particle` object has its own gravitational field.

L.4 Different methods of simulating kinematics

Regardless of the details of your final project you are going to need to have a way of approximating the motion of your system. We have assumed earlier that for a small interval of time Δt that the change in

acceleration is small and that the resulting change in position and velocity is given by

$$\vec{v}_{n+1} \approx \vec{v}_n + \vec{a}_n \Delta t, \quad (37)$$

$$\vec{x}_{n+1} \approx \vec{x}_n + \vec{v}_n \Delta t \quad (38)$$

which we've already referred to as Euler's Method. As stated, this is based on a Taylor expansion of the standard equations of motion to give

$$\vec{v}_{n+1} = \vec{v}_n + a_n \Delta t + \mathcal{O}\left((\Delta t)^2\right), \quad (39)$$

$$\vec{x}_{n+1} = \vec{x}_n + \vec{v}_n \Delta t + \mathcal{O}\left((\Delta t)^2\right) \quad (40)$$

where $\mathcal{O}\left((\Delta t)^2\right)$ signifies contributions of higher order. If Δt has a small enough value these higher-order contributions will be small and can be safely ignored. This is the assumption made by Euler's Method. The error in any given step is given by the truncation of the expansion and in this case is of order $(\Delta t)^2$. This error will accumulate each time the iteration is applied and hence the error in a simulation of fixed duration will be of order Δt . In the following paragraphs other algorithms are introduced. The errors in each of these approximations is not discussed but an investigation of this could form a part of your project.

As well as not being very accurate, for oscillatory systems the *Euler* method can be unstable. An alternative method called the *Euler-Cramer*, uses the velocity at the end of the step rather than the beginning of the step and should give more stable results

$$\vec{v}_{n+1} \approx \vec{v}_n + \vec{a}_n \Delta t, \quad (41)$$

$$\vec{x}_{n+1} \approx \vec{x}_n + \vec{v}_{n+1} \Delta t. \quad (42)$$

It is also possible that it may be better to compute the velocity in the middle of the interval Δt . This calculation is called the *Euler-Richardson* algorithm. This is particularly useful for velocity-dependent forces. This algorithm requires use of the *Euler* method to calculate the intermediate position x_{mid} and velocity v_{mid} at time $t_{\text{mid}} = t + \Delta t/2$. The force and acceleration are then computed for this mid-point.

$$\vec{a}_n = \vec{F}(\vec{x}_n, \vec{v}_n, t_n) / m \quad (43)$$

$$\vec{v}_{\text{mid}} \approx \vec{v}_n + \frac{1}{2} \vec{a}_n \Delta t \quad (44)$$

$$\vec{x}_{\text{mid}} \approx \vec{x}_n + \frac{1}{2} \vec{v}_n \Delta t \quad (45)$$

$$\vec{a}_{\text{mid}} \approx \vec{F}\left(\vec{x}_{\text{mid}}, \vec{v}_{\text{mid}}, t + \frac{1}{2} \Delta t\right) / m \quad (46)$$

so that

$$\vec{v}_{n+1} \approx \vec{v}_n + \vec{a}_{\text{mid}} \Delta t, \quad (47)$$

$$\vec{x}_{n+1} \approx \vec{x}_n + \vec{v}_{\text{mid}} \Delta t. \quad (48)$$

Finally we can look at a simpler alternative known as the *Verlet*- algorithm that is similar to the familiar equations of motion for constant acceleration. This uses the acceleration calculated at the end position to calculate the updated velocity which helps smooth out any changes in the acceleration. It is given by

$$\vec{x}_{n+1} \approx \vec{x}_n + \vec{v}_n \Delta t + \frac{1}{2} \vec{a}_n (\Delta t)^2, \quad (49)$$

$$\vec{v}_{n+1} \approx \vec{v}_n + \frac{1}{2} (\vec{a}_{n+1} + \vec{a}_n) \Delta t. \quad (50)$$

Clearly in this case we need some way of estimating \vec{a}_{n+1} . This can be achieved by using any of the algorithms/methods mentioned above first and then applying Verlet.

Beyond these algorithms you can also consider higher-order algorithms such as the well-known Runge-Kutta algorithms which are very widely used in physics simulations. The Runge-Kutta algorithms aren't discussed in detail here as it can be troublesome to correctly turn them into code, but you are welcome to implement them if you wish. The algorithms listed above are likely to be sufficiently accurate for most projects you are likely to attempt in this module. In fact, you will probably get acceptable results by just using the simple Euler-Cromer algorithm.

M Physical Simulation Project

The final project work should use exercise 7.1 as a starting point but is open ended and will require the submission of your code and a project report. In total the project makes up 60% of your final mark, where 10% is used for the projectile simulation of exercise 7.1. The remaining 50% is divided equally between the code and the report you are going to write over the last few weeks of the module.

Exercise 8.1 _____ (Project Code. Summative: 25% of the 281 mark)

- Your simulation should be a numerical simulation of a physical system for which energy is conserved. The code should use the techniques outlined in the course notes (e.g. Euler, etc). NB solving equations of motion analytically and then implementing the solutions will not obtain good grades.
- Remember that you are going to want to investigate a simplified situation initially so that you can check the validity of your simulation before applying it to a system that does not have analytical solutions. Following the steps outlined in the notes will help with this but you may have your own ideas too.
- Whichever physical system you simulate, you should think about how you are going to estimate the accuracy of the simulation results. Quantities such as position, period of orbit, etc can be compared to real experimental data or to analytical calculations. In addition you should be aiming to calculate the total energy of the system. Consider if there are any other conserved physical quantities you could calculate.
- We recommend you consider simulating the solar system as a possible (but challenging) goal, but you may not get anywhere near this or you may go much further. Clearly you would like your simulation to be able to deal with a complicated situation, but it is much more important that your code works!
- To reiterate the previous point, it's more important that you get a chance to analyse the results of your simulation than that you simulate something complicated. Obviously you don't want the system to be too simple either. To obtain a grade of C or above you would like your simulation to be able to predict something which could not easily have been calculated analytically.
- You could choose to make little use of object-oriented techniques, but this will make it unlikely that you will get a code mark above a C grade. You would ideally like to make good use of object-orientation by using well-designed classes that could easily be re-used in other simulations without any major changes.
- Your code will be marked for style and design as well as functionality and coding technique, so you will want to make the code clear to read with appropriate commenting, including javadoc-style comments for all methods.
- Your submission should consist of a single zip file containing all of the .java files needed to run your simulation. It should also contain a plain text file called README or README.txt that describes the contents of each .java file and how to run the simulation.
- The recommended format is that you'll have one .java file for each class that you design and use, and one .java file containing the main method which runs the simulation. It's only necessary to submit the final version of your code. In some cases you might have a couple of very different simulations that form part of your work - if you are unsure what to submit then please ask.

- Make sure that the zip file you submit contains ALL of the files needed to run your simulation including for example the `PhysicsVector.java` file (which it is recommended you use in your simulation).
- Read the appendices for details on how to write your output to a file so it can be imported into Qtplot or another plotting package.

Exercise 8.2 _____ (Project Report. Summative: 25% of the 281 mark)

- You should base the general form of this report on the previous report which you produced in week 5 of this module.
 - Your report should seek to quantitatively demonstrate how accurate your simulation is and to identify the main factors which limit the accuracy. Note that this does not simply mean listing the ways in which your simulation is a simplification of reality. Although this is relevant, it is more critical to identify how closely your simulation provides an exact solution to the equations of motion you are modelling.
 - The report will be marked using the same marking grid as for the report in week 5. Please also read the departmental report writing guidelines.
 - The recommended length for this report is approximately 12 pages with a maximum length of 15 pages (not including any bibliography or appendices if included).
 - Any reports that are over the 15 page limit will be subject to the equivalent of a late penalty (reduction of the report mark by one letter grade).
 - Your report should
 - Describe the physical process you're simulating and the equations of motion and numerical techniques used to model the process. You don't need to include a long historic background discussion of e.g. planetary motion - but do include the key equations and their physical meaning and main properties.
 - Describe the design of your program and the testing procedure. You should not document the `PhysicsVector` class which can be assumed to be understood. Document all of your own code and mention any external classes that you use. For each class you've designed you'll want to include a class diagram. A class-relationship diagram is optional but might be appropriate if you have many classes. A flowchart or something similar may help describe the main flow of your program.
 - Describe the results of your simulation using appropriate tables and figures. Be as quantitative as possible.
 - Discuss the accuracy of your simulation. In what situations does it fail? In what situations does it work well? How does the accuracy depend on Δt ?
 - Discuss the extent to which your simulation conserves physical quantities which you expect to be conserved.
-

References

- [1] <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>
- [2] <http://www.physics.lancs.ac.uk/docs/report-writing-guidelines.pdf>
- [3] Y. Daniel Liang, Introduction to Java Programming (Brief or Comprehensive), 10th Edition, Pearson Education, 2015
- [4] Deitel, Java, How to Program, 6th Edition, Pearson/Prentice Hall, 2005.
- [5] Mary Campione, Kathy Walrath, and Alison Huml, The Java Tutorial, Third Edition, Addison-Wesley, 2001.
- [6] Online Resources for Java Programmers (a list of useful links)
<http://download.oracle.com/javase/tutorial/index.html>
- [7] Java Platform, Standard Edition, v7 'AP8' Specification
<http://download.oracle.com/javase/8/docs/api/>
- [8] <http://commons.apache.org/proper/commons-math/>

Appendix 1 Dealing with errors

It is inevitable that code you write will initially suffer from errors. Many will be detected by the compiler which prints details of the errors to the screen. However some will only become evident when you try and run the program (runtime errors). This may be an ‘exception’ in which case some details will be written to the screen, or you may be testing your program and have discovered that it is not giving the results you expect. Either way, you will have to *debug* your program; that is, work out why it is going wrong and how to fix it.

There are three major types of errors that you will have to address (see Liang Chapter 2.17). These are

- **Syntax Errors** are errors that occur during compilation and are also called compilation errors. These result from errors in code construction such as mistyping a keyword, leaving out a semi-colon, a missing curly brace, or a missing type identifier. These are the easiest to find as the compiler usually indicates the line number associated with the error and gives a hint to what the problem is.
- **Runtime Errors** are errors that occur during the running of the program that result in its termination. These errors are usually the result of the program trying to process data that does not make sense. For example, you may have entered a string where an integer is required. Another typical example is attempting to divide by zero. In this case you get output from the program indicating where the problem is.
- **Logic Errors** occur when the program runs but does not do what you intend it to do. These errors are the hardest bugs to track down as there are no technical programming errors, rather, you have made a mistake in the logic or planning of your program. In this case you will probably have to trace through your code to work out where the mistake is made. This is where well-constructed test cases will help you along with print statements.

There are several techniques you can use to find out what the problems are.

- With errors detected by the compiler always start with the first and work through them in order, recompiling each time. You may find that solving the first error will eliminate the rest as sometimes a simple error at the beginning of a program will cause many *cascading* errors later on (the exception to this is a missing brace at the end of the code, if you have a very long list of errors re-indent your code and check for missing braces).
- If the compiler detects an error then look at the details it gives about which line in the code the error was found and what the error was. Look carefully at the line of code in your source file — you may simply have spelled something wrong or omitted a semicolon. Forgetting a semicolon will tend to result in an error message that refers to the following line (as it is the following line that then fails to make sense to the compiler), so you should also check the lines around the one mentioned by the compiler.
- If the compiler gives an error “cannot resolve symbol” when you are using some Java feature that you are sure you have spelled correctly, then the problem could be that you have not got the necessary `import` statement at the start of the file.
- If an exception occurs when you try and run the program, again, look at the details written to the screen — the line number and the type of exception thrown — then examine this area of your source code. You may have written code that tries to go beyond the end of an array for example.

- If you suspect a particular block of code may be causing problems try surrounding it with the longer form of comment `/* . . . */` which effectively removes it from your program without lots of unnecessary deleting. Then compile (and run) the program again to see if the error/exception still occurs — if it doesn't then the problem lies within the code that was commented out. This method can be tried with both compiler and runtime errors.
- So long as the program compiles okay, you can add `System.out.println(...);` to your program at strategic points to determine how far the program gets before the exception is thrown. This allows you to narrow down on where the problem code must be.
- When we get to later points in the course and you are defining and using objects you may run into 'null pointer exception' errors at runtime. This cryptic-sounding message is telling you that you have tried to use an object that is not instantiated, i.e. you need to invoke a constructor.

Appendix 2 More advanced input and output

Formatting output – DecimalFormat

Sometimes you will want to control the way in which the output of a program is written. For example, for some numbers not all the significant figures will be required, or in fact they may get in the way if you wish to write more than one result on each line. It can be helpful then to be able to *format* output.

There are three steps to printing formatted output. The first is to create the format³² that you wish to use. This is done using

```
DecimalFormat myFormat = new DecimalFormat("###.000");
```

where `myFormat` is what you decide to call the format. The format is defined using the string `"###.000"` where `#` denotes a digit and `0` denotes a digit or a zero (if there is not a digit to go in this place a zero is printed instead). The decimal point has its usual meaning. E.g. the number 23 in this format would be printed as `23.000`.

For comparison, if the format was `"000.000"`, the output would be `023.000`. This shows the difference between `#` and `0`.

Note that to use `DecimalFormat`, you will need to add `import java.text.DecimalFormat;` at the top of your program, so that the compiler understands what you mean by it. If you forget this, you will get a compiler error “cannot resolve symbol”.

Once a format has been created in this way it may be used again and again for different numbers being printed out. To output a number, here 23, in the format `myFormat` use the code:

```
// creates a string (called sOutput),  
// which is the number in the desired format  
String sOutput = myFormat.format(23);  
  
// print out as usual  
System.out.println(sOutput);
```

The formats created may be used on all number variable types in exactly the same way. Despite the name `DecimalFormat` the value may be an integer, as demonstrated above, or formats need not include a decimal place, thus printing numbers truncated to integer form.

As well as `#`'s and `0`'s an `E` may be used to specify the exponential form of a number. For example `"0.##E0"` would give a number in scientific format, with a maximum of two decimal places — any extra digits being rounded off.

```
DecimalFormat mySciForm = new DecimalFormat("0.##E0");  
String sOutput = mySciForm.format(157.98642);  
System.out.println(sOutput);
```

The above code outputs 157.98642 in scientific format with two decimal places. The output from the program is therefore `1.58e2`

³²In the example, `myFormat` is actually an *object* of type `DecimalFormat`. *Objects* are discussed in section J.2.

As a shortcut, there is no need to create the intermediate `String`. The following code does exactly the same as the previous example.

```
DecimalFormat mySciForm = new DecimalFormat("0.##E0");
System.out.println(mySciForm.format(157.98642));
```

Formatting Output – printf

Formatting can be easily done by using the command `System.out.printf` which provides direct access to the Java formatting classes. The standard syntax of this command is:

```
System.out.printf(format, item1, item2, ..., itemk)
```

where `format` is a `String` that may consist of sub-strings and format specifiers. The `item`'s are the variables to be included in the output. Each will correspond to a format specifier.

The format specifiers are a `%` symbol followed by a *conversion* as given in Table 4.

Table 4: Common Format Specifiers

<i>Specifier</i>	<i>Output</i>	<i>Example</i>
<code>%b</code>	a boolean value	<code>true</code> or <code>false</code>
<code>%c</code>	a character	<code>'a'</code>
<code>%d</code>	a decimal integer	<code>200</code>
<code>%f</code>	a floating point number	<code>45.621000</code>
<code>%e</code>	a number in standard scientific notation	<code>4.562100E+01</code>
<code>%s</code>	a string	<code>"my Java string"</code>

An example of using `printf` is

```
int count = 41;
double pi = 3.145;
String name = "Bilbo";
System.out.printf(
    "My name is %s. I have %d Nobel prizes.\n"+
    "By the way, pi is approximately %f\n",
    name, count, pi);
```

and will output

```
My name is Bilbo. I have 41 Nobel prizes.
By the way, pi is approximately 3.145000
```

By default 6 significant figures are displayed after the decimal point for a floating point number (whether using scientific notation or not). Alternatively, you can specify the width and precision of the output by modifying the format specifier. For example, to display a field five spaces wide with three decimal places after the decimal point you would use the following code:

```
System.out.printf("%5.3f\n\n",x );
```

The symbol `%M.Nf` means that the formatted number will be a minimum of `M` spaces wide with `N` significant figures after the decimal point. A similar formalism can be used with strings. e.g. `%10s` is a string format specifier with a width of at least 10 spaces.

Table 5: Examples of specifying the width and precision in printf statements

<i>Example</i>	<i>Output</i>
<code>%3c</code>	prints a character with two spaces to the left of the character.
<code>%-3c</code>	left justifies the character so that the two additional spaces are printed to the right
<code>%5d</code>	prints an integer in with a width of at least five spaces. If the width is greater than five spaces then it will automatically be increased to the correct number of spaces.
<code>%9.5f</code>	prints a floating point value with a width of at least nine spaces including the decimal point. There will be five significant digits after the decimal point and the width will be increased if the total width is greater than nine spaces.
<code>%-9.5f</code>	left justifies the floating point number with additional spaces to the right.
<code>%10.2e</code>	A number in scientific notation with a minimum width of ten spaces and two significant figures after the decimal point.
<code>%15s</code>	a string with a minimum width of fifteen spaces.

Here is an example program making use of printf:

```
public class printf{
    public static void main(String[] args) {

        int count = 41;
        double pi = 3.145;
        String name = "Bilbo";

        System.out.printf(
            "My name is %s. I have %d Nobel prizes."+
            "By the way, pi is approximately %f\n", name, count, pi);
        //examples of setting the minimum width precision
        // %5.3f is a minimum of 5 spaces wide with 3 spaces
        // after the decimal point
        double x = 1/3.;

        System.out.printf("%5.3f\n\n",x );
        x=50/3.;
        System.out.printf("%5.3f\n\n",x );
        long Sum = 0;

        for(int i = 9; i > 0; i--)
        {
            Sum += i*Math.pow(10,9-i);
            // print integers with a width of 10 spaces
            // the %-10d is left justified
            System.out.printf("%10d %-10d\n",Sum,Sum);
        }

    }
}
```


Input from a file

As with reading input from the keyboard, you can use a `Scanner` to read text from a file. The following example illustrates this, and also introduces the idea of using `try` and `catch` blocks to deal with errors that can arise when the code is running (*exception handling*).

```
import java.util.*;
import java.io.*;

public class FileInput{

    public static void main(String[] argv){

        File file = null;
        file = new File ("textOutput.txt");

        Scanner scanner = null;
        try {
            // Create a scanner to read the file
            scanner = new Scanner (file);
        } catch (FileNotFoundException e) {
            System.out.println ("File not found!");
            // Stop program if no file found
            System.exit(0);
        }

        // try reading the file as though it where integers
        // and throw an exception if it fails
        try{

            // check that there is a next entry of any type
            // can use hasNextInt to check if is an int
            while (scanner.hasNext()){
                int i = scanner.nextInt();
                System.out.println(i);
            }
        }
        // catch the type miss match exception if it exists
        catch (InputMismatchException e) {
            System.out.println ("Mismatch exception:" + e );
        }

    }
}
```

Output to a file

Printing output to a file is quite similar to getting input from a file, although the file need not already exist. At the beginning of your source file `import java.io.*` is needed. If you aren't using `try` and `catch` you'll need to include `throws IOException` as part of your method declaration (see below).

To *open* the file, here called 'file2.out', we can use the following code:

```
// Open file to print output to
PrintWriter q = new PrintWriter("file2.out", true);
```

This line is only needed once, and must be used before you first try to send output to the file. The file will be closed (with the contents saved) automatically when the program ends. Don't forget the `true` when creating the `PrintWriter`, or this will not be the case and you could easily lose some output!

If the output file already exists when you run the program, e.g. from the last time you ran it, **it will be overwritten and lost**, so copy or rename files containing output you want to keep, before re-running a program.

Whenever you want to write data to the file you can use:

```
q.println("This will be written to the file.");
```

The above line can be treated in the same way as `System.out.println()`, so you can output variable values etc. as usual. As with writing to the screen, `println()` will cause data to be written to a new line each time.

For example, a complete program could look like the one below. Note that rather than open the file in one step, we've used two separate steps in this example so that we can check explicitly whether the file already exists. In addition, we've used the `PrintWriter` constructor which takes only one argument, whereas in the previous section we used a constructor that took a second Boolean argument which makes the `PrintWriter` 'auto-flush' (check the Java web pages for details). For this reason we have to explicitly 'close' the file in this example.

```
import java.io.*;

public class FileWrite{
    public static void main(String[] args) throws IOException{

        java.io.File file =
            new java.io.File("output_example.txt");

        if(file.exists()){
            System.out.println("The file already exists");
            System.exit(0); // quit the program
        }

        java.io.PrintWriter q = new PrintWriter(file);

        q.println("Output to be written to file");

        q.close();

    }
}
```