

Practical Physics: Scientific Programming in JAVA



This document contains the notes, checkpoint tasks and background information for SCIENTIFIC PROGRAMMING which forms part of the Practical Physics course.

Additional on-line information relating to this course, including example programs, can be obtained at:

<http://www.ph.ed.ac.uk/~cpalansu/teaching/Scientific-Programming/>

Contents

1	Introduction	5
1.1	Synopsis	5
1.2	Why learn a computing language	5
1.3	Uses of Java	5
1.4	What you will learn	5
1.5	Other Languages	5
1.6	Flavours of Java	6
1.7	Documentation	6
2	Background	7
2.1	Basic Structure of a Computer	7
2.2	The Computers being Used	7
2.3	The Linux operating system	8
2.4	Components of a program	8
3	Checkpoints	10
3.1	Introduction	10
3.2	The Checkpoints	10
3.3	Checkpoint Submission Dates	10
3.4	Checkpoint Grading	10
3.5	Checkpoint Requirements	11
4	Getting Started	12
4.1	Introduction	12
4.2	Preliminaries	12
4.3	Basic Control (Linux Systems)	12
4.4	Changing your password	12
4.5	Getting Course Documentation	13
4.6	Logging-off	13
4.7	Rules on use of Computers	13
5	Basics of Linux	14
5.1	Introduction	14
5.2	Entering Commands	14
5.3	The Directory Structure	14
5.4	File and Directory Names	15
5.5	Directory Manipulation	15
5.6	List of Files and Directories	15
5.7	Viewing and printing files	15
5.8	Manipulating files	16
5.9	Wild-cards in file names	16
5.10	e-Mail	16
5.11	The Shell	17
5.12	More Information	17
6	The emacs editor	18
6.1	Introduction	18

6.2	Starting up emacs	18
6.3	Simple Editing	18
6.4	Things that go wrong	19
7	Your First Java program	21
7.1	Your First Program	22
8	Checkpoint 1	23
9	Data Types, Variables and Operators	24
9.1	Introduction	24
9.2	Data Types	24
9.3	Variables	25
9.4	Variable Names and Declaration	25
9.5	Assignment	26
9.6	Basic Arithmetic	27
9.7	Arithmetic Assignment Operators	28
9.8	Strings	29
10	Basic Input and Output	32
10.1	Introduction	32
10.2	The Console Class	32
10.2.1	Writing data using the Console class	32
10.2.2	Reading data using the Console class	32
10.3	Nicer Output Formatting using printf()	33
11	Math Class and Constants	36
11.1	Introduction	36
11.2	The Java Math class	36
11.2.1	Trigonometric Functions	36
11.2.2	Exponential, Power and Hyperbolic Functions	36
11.2.3	Absolute values, Nearest Integers, Max/Mins	37
11.2.4	Random Numbers	37
11.3	Constants	37
12	Checkpoint 2	39
13	File Output and Input	40
13.1	Introduction	40
13.2	The Basics	40
13.3	Output with the PrintWriter class	40
13.4	Input with the BufferedReader class	41
13.5	Breaking up input lines with Scanner	42
13.5.1	Reading ints and doubles	43
13.6	Summary and Additional Features	43
14	Checkpoint 3	45
15	Conditional Statements	46
15.1	Introduction	46

15.2 The boolean Data Type	46
15.3 Conditional Statements	46
15.4 Double Conditionals	47
15.5 Multiple Conditional	48
15.6 The <code>System.exit()</code> Method	48
15.7 The switch Construct	48
16 Checkpoint 4	50
17 Loops	52
17.1 Introduction	52
17.2 The while Loop	52
17.3 The do/while Loop	52
17.4 The for Loop	53
17.4.1 The for-each Loop	54
17.5 Nesting of Loops	54
17.6 The break statement	55
18 Arrays and Strings	56
18.1 Introduction	56
18.2 One Dimensional Arrays	56
18.3 Initialisation	57
18.4 Addressing Arrays	57
18.5 Multi-Dimension Arrays	58
18.6 Warning	58
18.7 Strings	59
18.8 Declaration and Initialisation	59
18.9 Extending Strings	59
18.9.1 Useful String Methods	60
19 Plotting Graphs Using the <code>matplotlib</code> Package	62
19.1 Introduction	62
19.2 Plotting Multiple Datasets	63
19.3 Setting the Plot Range	63
20 Checkpoint 5	64
21 Introduction to Methods	66
21.1 Introduction	66
21.2 An Example static Method	66
21.3 Arrays and Methods	67
21.4 <code>main()</code> as a Method	69
22 Introduction to Objects	71
22.1 The Basics	71
22.2 A simple Point object	71
22.3 Putting it together	75
22.4 Why bother with objects and classes	76

23 Checkpoint 6	77
24 Checkpoint 7	79
A Finding and Fixing Bugs	82
A.1 Introduction	82
A.2 Types of Bugs	82
A.2.1 Syntax Errors	82
A.2.2 Runtime Errors	83
A.2.3 Working Program – Wrong Results	84
A.3 Problems with the Systems	84
A.4 Myths, General Miss-conceptions and Classic Excuses	85

1 Introduction

Read this and the next section **before** you attend the first class.

1.1 Synopsis

The aim of this course is to teach the basics of scientific computer programming using Java in the Linux environment. This course is taught “on-line” in a series of **six** 3 hour sessions using the School’s Computational Physics Laboratories. This course is taken in weeks 2 to 7 of Semester 1, starting on Monday 15 September 2014.

1.2 Why learn a computing language

All computer programs, whether they perform simple calculations or run an entire network of systems, are written in a *programming language* of some type. Anybody wanting to use computers beyond simple Web browsing, word processing and e-mail has to learn some type of programming language. Most scientists, and especially physicists, need to make more than this basic use of computers and so need programming skills; now is a good time to start!

1.3 Uses of Java

Say Java and what springs to mind is animation and user interactions on Web pages using either,

1. JavaScript—a small section of Java like code that is inserted into Web pages to perform functions that normal XHTML does not support or,
2. Applets—small self contained Java programs that are run by a web browser to perform animation, provide user interfaces and other complex operations.

Java however, is much more than just a Web-enabled graphical add-on, it is a fully functional general purpose computer language with excellent graphics and seamless integration with windows/menus and Web environments. More importantly it is truly machine independent, so Java programs will run, without modification, on any system on which the language has been implemented. These reasons makes it an ideal modern language in which to teach elementary computing.

Java is a relatively new language which is still developing. Numerical support is still patchy. In particular there are few numerical libraries at present. This is changing and we expect to see a rapid emergence of Java as a scientific computing language via such projects as JAVA GRANDE which aim to implement Java on massively parallel supercomputers such as HECToR (<http://www.hector.ac.uk>).

1.4 What you will learn

This short course will concentrate on the Java programming language and you will be writing short *applications* to perform basic calculations. We will introduce the concept of using external packages to add additional functionality to the vanilla Java language. In addition, we will cover how we get data into and out of your applications using the command line and files. Finally, we will introduce the object-oriented approach to programming that now underpins much of modern computing.

1.5 Other Languages

There are a vast range of computer languages, the most commonly seen in a scientific setting being:

- “C”—a powerful general purpose language used extensively throughout the software industry for the last 10 to 15 years. Used extensively in system codes and many current applications, but most new applications are being written in one of the newer alternatives. This language is the basis of the Linux operating system and the bedrock of the Open Source movement.
- C++—a super-set of “C” which adds object-oriented programming. This language is much favoured by the software industry. It lacks standardised graphical and network interfaces, is rather syntactically difficult and the resultant programs are frequently complex to understand and maintain.

- Fortran—the traditional numerical language with excellent numerical libraries and support for parallel computer systems. There is an optional course at Senior Honours level using Fortran 95 with parallel computing extensions for large scale computational modelling.
- Perl—A modern interpreted (it does not need to be compiled, see later in the course) language with a C-like syntax and very powerful character and text manipulation features. It is widely used for system control programs, data analysis and file conversion.
- Python—Has many similarities to Perl in that it is interpreted but has better support for numerical functions, graphical interfaces and object-orientation. This language is widely used for the post-processing and analysis of scientific data.

1.6 Flavours of Java

The main developer of Java is Sun Microsystems Inc which was recently acquired by Oracle Corporation. This course has been developed using Sun Microsystem JDK 6, which can be downloaded from <http://www.oracle.com/technetwork> with a free user licence¹.

Other Java compilers are available, for example from gnu who are developing a open source Java compiler as part of the gcc compiler. This does not currently support all JDK 6 features and does not have fully functional graphics.

1.7 Documentation

This course is available on-line at:

<http://www.ph.ed.ac.uk/~cpalansu/teaching/Scientific-Programming>

which also includes a set of complete program Examples many of which are useful starting points for the checkpoints. These examples are available as clickable links from the electronic version of these notes.

These pages also contains links to many other on-line documents, including the details of the local classes and the full Java on-line documentation set.

There are relatively few Java books that teach “basic programming”, most either assume that you are an experienced “C” programmer and want to convert to Java, or that you want to use Java primarily for Web applications. Neither are appropriate to you.

The three most useful book appear to be:

1. JAVA GENTLY FOR SCIENTISTS AND ENGINEERS Judith & Nigel Bishop, Addison-Wesley.
Good simple book with physicist in mind. Uses its own classes. ≈£37.00.
2. Small Java: How-to-program by Deitle and Deilte. Deitel & Associates Inc Pub.
short(er) version of the full book, see below, with little on graphics or user interfaces, but sill 600 pages. Starts as the very beginning and explains every step in detail. ≈£52.00.
3. Java: How-to-program by Deitle and Deilte. Deitel & Associates Inc Pub.
Very very long detailed book (1,500 pages). It starts at the very beginning and explains every step in detail, including graphics and user interface. ≈£45.00, but available second hand from about £30.00.

There are many books on Java being published every month, the best way to find the book you like is to browse the shelves of the major city bookshops.

¹Subject to the restrictions detailed in the Binary Code Licence Agreement

2 Background

Read this and the next section **before** you attend the first class.

2.1 Basic Structure of a Computer

Before embarking on computer programming it is useful to have some idea about what a computer is, what it can, and more importantly, cannot do. The basic structure of a digital computer is outlined in Figure 1.

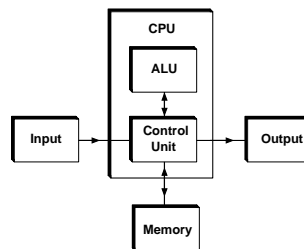


Figure 1: Basic structure of a digital computer

The *Central Processing Unit* (CPU) operates on a series of instructions called a *program* which is stored in *memory*. The CPU consists of an *Arithmetic Logic Unit* (ALU) which performs such functions as addition, subtraction, comparison etc. and a *Control Unit* which directs and monitors the operation of the computer. The *input* and *output* provide an interface to the outside world, for example *inputs* can be the keyboard and *output* the display screen. The *input* and *output* units also provide access to data storage devices such as hard discs, DVDs, USB devices etc. All these may be either connected directly to the computer or over the *network*.

The operation of the computer is based on the *Fetch - Execute* cycle as shown in figure 2 where the *Control Unit* continually *Fetches* instructions and data from memory and *Executes* them using the ALU. This cycle continues the whole time the computer is switched on.

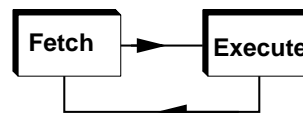


Figure 2: Fetch and Execute cycle

The computer holds all information (number, letter, instructions), in memory as patterns of binary bits, (ones and zeros). The instructions are decoded by the *Control Unit* and determine what operations the ALU performs on the data. The desired sequence of instructions (the *Program*) have to be placed in memory by the “programmer”, which is what we will be doing during this course. Remember that the computer is a machine that exactly follows instructions and it is the sole responsibility of the programmer to get these instructions right!

The binary instructions used by the computer are not convenient for humans to use. This is further complicated by the fact that different computers use different sets of instructions and even different format for numbers. It is therefore much more convenient to program in a *High Level Language* which is (more) humanly readable and, to a large extent, independent of the details of the computer being used. The high level language chosen for this course is Java.

2.2 The Computers being Used

This course will be run on *Linux* systems from within the *Computational Physics Laboratory*. All user files and data are held on a fileserver so you can work from any of the computers. It is also possible to connect to the Physics server from other University computers, for example in the Main Library.

2.3 The Linux operating system

The *operating system* controls the computer hardware, allows programs to run and controls the users interface. The Linux operating system with an X-windows interface is widely used in the scientific and computer science areas. It is a *multi-user* system where more than one person can use the machine at once without “seeing” each other. Linux is used on medium sized workstations and servers right up to massive super computers.

The user interface is via the X-windows system which is “windows and mouse” based, but most interaction with the system is via a *Terminal Window* into which you type commands. In addition the different utilities are less well integrated than on the PC, although things are slowly getting better.

2.4 Components of a program

A computer program written in a high level language has to be converted into binary instructions that are executed by the processor. Using Java this is a two stage process, the components of which are show in Figure 3

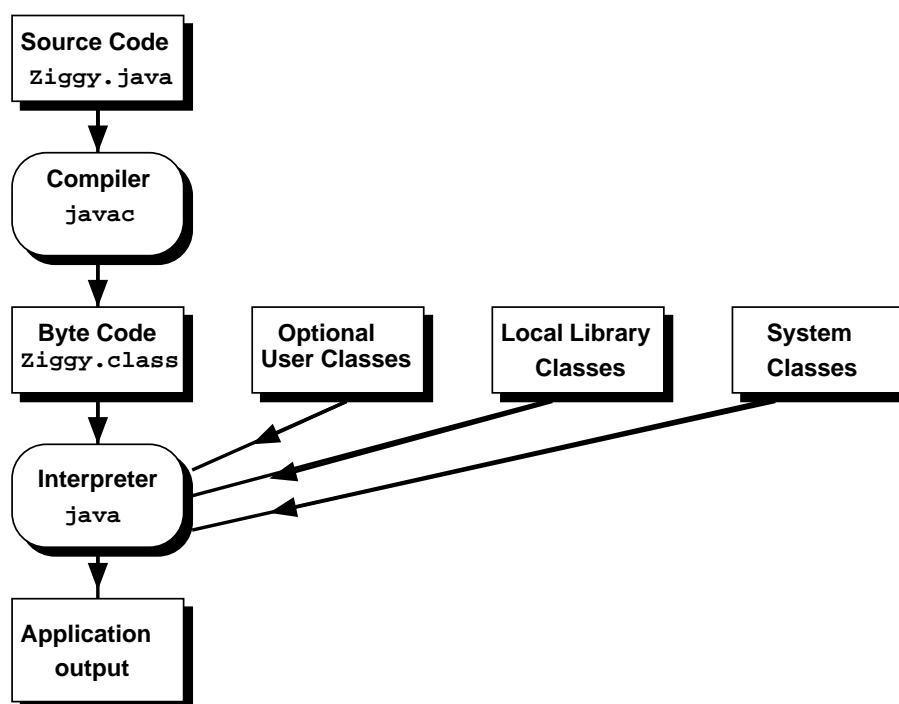


Figure 3: Components of a Java computer program

1. The actual Java program is held in a file on disc which is called the *source code*. You create the file with an *editor*. This file can also be viewed on the screen or printed. It has the .java extension.
2. This *source code* is *compiled* to produce *byte code*. This is done with a *Compiler* (javac in this case). The resulting file has the .class extension.
3. The *byte code* ² is then combined, (or *linked*) with other *byte code* files and library files and executed by the *Interpreter* (java in this case), hence running the program.

Utilities Used in Program Writing and Preparation.

The three utilities you will use in this course are:

²Java differs from most other languages in that the output form compiler is independent of the machine type.

1. **Editor:** This allows you to write, edit and store on disc the source code of the program. Most large computer systems offer you a choice of editors, but they all basically do the same thing. The editor being used in this course is `emacs`. See section on the use of `emacs`.
2. **Compiler:** This compiles the source code into byte code, this is called `javac`.
3. **Interpreter:** This executes the byte code on the particular machine after linking in the relevant additional *classes*. This is called `java`.

These are the three basic *tools* used to write and execute programs and the only ones that will be needed in this course.

3 Checkpoints

3.1 Introduction

*This section contains details of what you are required to complete during this course. Read this **before** you start the course.*

This programming course is defined and assessed via a set of *Checkpoints*, each of which involves at least one programming task. In order to complete these Checkpoints you will need to read and study the course notes. The laboratory is also supported by demonstrators who are there, and paid, to help *you* through it.

3.2 The Checkpoints

There are **6** compulsory Checkpoints and one *optional* Checkpoint. You work through the Checkpoints at your *own pace*. If you have had significant previous experience of programming, for example from *Computer Science 1A*, then you will find first three Checkpoints very simple. In this case you may attempt the optional Checkpoint 7 instead (examine the Checkpoint 7 before making a decision).

The Checkpoints are:

1. **Address Program**—Extension of “Hello World”. (5%)
2. **Variables and Arithmetic**—Two item checkpoint on basic input/output, arithmetic and variable types. (10%)
3. **File Input and Output**—Two item checkpoint on reading and writing files. (15%)
4. **Root of Quadratic**—Calculation of the roots of a quadratic equations. Used to demonstrate conditional statements. (25%)
5. **Damped Simple Harmonic Oscillator**—Calculation and display of amplitude of a damped harmonic oscillator under various damping conditions. (25%)
6. **The Circle Object**—Writing of a simple circle object to practice basis object-oriented programming. (20%)
7. **Percolation**—Program to simulate percolation through an array of variable size and density. (*optional replacement for checkpoints 1 to 3 for experienced programmers only, 30%*)

3.3 Checkpoint Submission Dates

You should get checkpoints marked by the demonstrators *immediately* after completion. You will typically get useful feedback from each checkpoint that will help you with the next. The **absolute** final submission dates for the checkpoints are:

- **Checkpoint 1 and 2: 5.00pm Thursday 3 October**
- **Checkpoint 3 and 4: 5.00pm Thursday 17 October**
- **Checkpoint 5, 6 and (optionally) 7: 5.00pm Thursday 31 October**

checkpoints submitted after these dates will **not** be counted towards the assessment of this course.

3.4 Checkpoint Grading

The first three checkpoints will be graded on a **3** point scale and the second four on a **5** point scale. The grading will include,

1. Function of the code, does it do what you think it does.
2. Design and layout of the code, including the use of good structure, comments and use of sensible variable names.
3. Understanding of the problem and the ability to answer questions regarding the program.

3.5 Checkpoint Requirements

The programs submitted for *Checkpoints* **must** be your own work, and **must** not be copied, either in whole or in part from other students, directly from textbooks or from the web. Submitting or attempting to submit other peoples work as your own is a breach of *University Code of Student Discipline*. This does *not* mean you cannot seek or give assistance to your class-mates, but you must not give or copy programs.

On a less legalistic stand, remember *programming* is a very valuable skill, both as part of your degree and as a general, and very marketable, skill in future employment. (Look in any recruitment paper and see how many jobs require Java programming!) The only way to become a competent programmer is to sit in front of a terminal and “*write, test and debug programs*”. This course gives this opportunity with demonstrator cover to assist you. Make the most of it, you *will need* the skills you acquire here!

What Next?

You are now ready to start the course. The next few sections will tell you how to log-in to the computer systems and the basics of Linux. These sections are best read while sitting in-front of a terminal. If you want to “read ahead” and start the Java language, jump to section 7.

4 Getting Started

Work through this section at the start of your first computing laboratory day.

4.1 Introduction

This section contains *very basic* information to allow you to get started and in particular how to log-in to the systems and obtain the on-line course and system documentation.

4.2 Preliminaries

You will have been issued with a *Username* and *Password* for the Physics computer system. Note actual *Username* is same as your Microlab / SMS one, but the account and associated *Password* are different. You will keep the same *Username* throughout your undergraduate period. The *Password* is the equivalent to your security PIN number for your bank card and must be considered as “strictly private” to you.

The Computational Physics Laboratory is located in


JCMB 1.1028 (the large computer lab in the basement)

It consists of about 70 Linux systems. You can login at any Linux system with your supplied username and password.

4.3 Basic Control (Linux Systems)

The system is used via a combination of “tool-bar”, pop-up menus connected to the mouse buttons *and* commands typed into an *active* terminal window.

1. The pop-up menus are obtained by clicking on the buttons on left of the upper tool-bar.
2. To create a “Terminal” select “System Tools”, then “Terminal”.
3. You can place a start “Terminal” icon on the lower tool-bar by “Right Click” and hold on “Terminal”, and “drag” to lower tool-bar. Then clicking on this icon will create a terminal.
4. Any window is made *active* by moving the cursor with the mouse into the window. The frame and top-bar will “highlight”. This window remains *active* as long as the cursor is in the window.

Pop-up menu operations occur immediately, but typed commands are *not* executed by the system until you press the RETURN or  key.

Create a “Terminal Window” now.

4.4 Changing your password

The first thing you **must** do is to change your password. The password that you were issued with is not secure and also is usually very difficult to remember. When selecting a password you should:

1. select one that *you* find easy to remember, but other people will find difficult to *guess*.
2. they *must* be at least 6 characters, and *must* contain at least one of, a) CAPITAL letter, b) digit, c) punctuation character.
3. avoid common words or names. (name of boy/girl friend are the easiest to guess and should be avoided). Remember there are also a finite number of swear words!
4. not so complicated that you cannot type it correctly, forget it, or you have to write it down!

Good passwords are typically based on “words” with mis-placed letters, added digits, punctuation and capitals in odd places.

Once you have selected a new password you can set it from an *active* terminal window with the command:

```
yppasswd
```

You will then be prompted for:

1. your CURRENT password,
2. your NEW password,
3. repeat of your NEW password.

each time there will be no echo and you must type RETURN after each response.

If you either a) get the current password wrong, b) do not type the new password identically twice, c) your new password does not obey the system rules, you will get an error message and the password will not be changed. If so repeat the operation correctly.

4.5 Getting Course Documentation

In addition to this booklet, all the course information is obtained via the Web, which is accessed via the *firefox* browser. To start firefox click *once* on the firefox button on the tool-bar, or type

```
firefox &
```

in the terminal command window.

Note: firefox is a very large application and will take several 10s of seconds to start. Be patient, and do *not* “try again”, it *will* appear!

All of the course documentation can be found at:

```
http://www.ph.ed.ac.uk/~cpalansu/teaching/Scientific-Programming
```

All the course instructions are available from this page. Most documents are in Adobe Acrobat format which is automatically viewed by a *plug-in*, and can also be printed. See on-line instructions regarding printing of documents.

4.6 Logging-off

When you are finished for the day you **must** log-off the system. To do this select *Logout* from the *Control Menu*, and then confirm your selection. You will be logged off the system.

4.7 Rules on use of Computers

When you signed for your physics username, you were issued with a summary of the *University Computer Regulations* which you **must** read and obey. Breaches of these regulations constitutes a breach of the *Student Code of Discipline*.

Remember the use of the School Computer Facilities are a *privilege* made available to you for specified course work and to assist your academic studies at The University of Edinburgh.

What Next?

You now need to read the next **two** sections on **Basics of Linux** and **The emacs editor**.


5 Basics of Linux

You should read through this section, ideally sitting in front of a terminal so you can try out the various commands.

5.1 Introduction

The Linux operating system is accessed through a combination of the Gnome interface and a *terminal window* into which you type *commands*. This section covers the basic Linux commands that you will need to effectively use the terminal interface.

5.2 Entering Commands

Commands are typed into an terminal window, however the command is not executed by the system until you press the Return or  key. Note also that all commands are *case sensitive* (capital and lower case letters are different).

5.3 The Directory Structure

All information, be it a program source, letters, programs, or system applications, are all held in “files”. These files are grouped together in “directories”. The directories are then arranged as a “tree”, starting from the system *root* directory. This is shown schematically in Figure 4

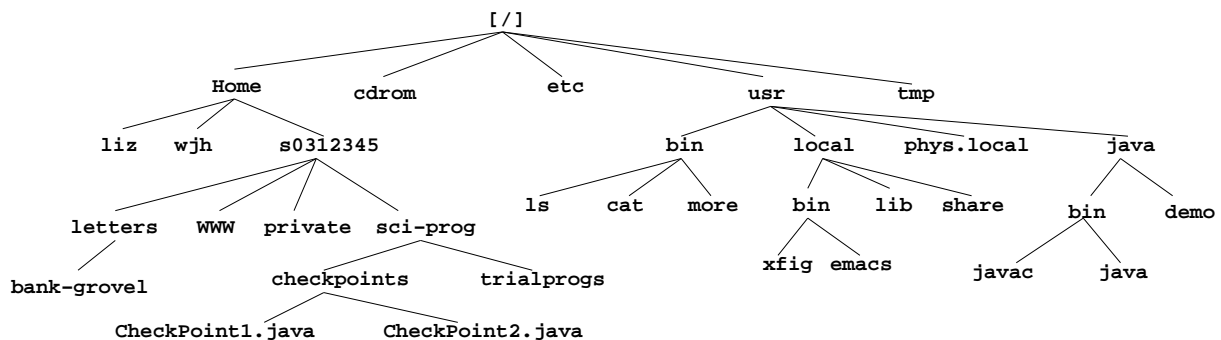


Figure 4: Schematic of part-of the Linux file system.

For example the file that contained the source code for the solution to your first checkpoint may be located in:

```
/Home/s0312345/sci-prog/checkpoints/CheckPoint1.java
```

while the Java compiler javac is located in:

```
/usr/java/bin/javac
```

The structure is divided into *user* and *system* files, being the files that belong to the users of the system and the files that are part of the general system. On this system the users file are under /Home, while the system one are mostly under /usr (the name is historical!).

Your files are located under /Home in a directory which has the same name as your *username*. So in the above example if your username is s0312345 then your files are located under

```
/Home/s0312345
```

This is known as your *home directory*. All files below this directory belong to *you*, and *you* can do anything to them, (read, write, delete etc.).

Path Names

Filenames can either be *absolute*, where the file names defines the whole *path* from the system *root* (/) directory. Such filenames start with (/) as above. Alternatively, filenames can be *relative* to where “you are” in the file structure. In this case the name does not have a leading (/).

For example, if your username is s0312345 then when you login the terminal screen is initially in your *home directory*, being /Home/s0312345. This is called your *present working directory*. So when in this directory the file checkpoint1.c can be referred to as **either**:

```
/Home/s0312345/sci-prog/checkpoints/CheckPoint1.java
```

or

```
sci-prog/checkpoints/CheckPoint1.java
```

where the first example is its *absolute* name and the second is its name *relative* to your *present working directory*. This all sounds rather complex, but is actually very intuitive once you have tried it out for yourself!

5.4 File and Directory Names

File and directory names can be up to 32 characters long and can be *any* combination of letters, digits and punctuation characters - + . , _ : ; . There are however some rules, and basic guidelines,

1. Directories or files names that start with a dot (“.”) are called *hidden*. These will not appear on normal directory listings, see below. Such files are *normally* used to hold user options to various applications.
2. Most filenames are of the form <basename>.<extn> where <extn> is used to specify the type of information in the file.
3. It is possible to use other punctuation characters but they can clash with wild-card characters and other special characters, so should be avoided.

5.5 Directory Manipulation

The following commands are used to check and set your *present working directory*:

```
pwd          show the present working directory (i.e. your current location in the directory tree).
cd           change directory to your home directory
cd newdir    change directory to newdir (newdir can be absolute or relative)
cd ..        change directory to parent directory (one level up)
```

Your prompt changes as you change directory to show the basename of the *present working directory*.

New directories can be created, and deleted with:

```
mkdir newdir make directory newdir (newdir can be absolute or relative)
rm -r olddir remove directory olddir.
```

5.6 List of Files and Directories

Generating lists of the contents of directories is obtained with the very flexible command `ls` which has many dozens of options. The most common are:

```
ls          list of file and directories in current directory (simple output)
ls -l       list of file and directories in current directory (long format: includes file permissions, sizes and modification dates)
ls -a       list of all files, including hidden files. (see section on filenames)
```

5.7 Viewing and printing files

The contents of a file can be “paged” to the screen with the utility `less`. For example, if you want to view the contents of file `Fred.java` then the command is,

```
less Fred.java
```

If the file `Fred.java` is longer than the current terminal screen the following keys come into play, these being.

SPACE	Forward one screen
D	Forward half screen
RETURN	Forward one line
U	Back one screen
Q	Quit

Plain text files can be printed using

```
lp <filename>
```

where `<filename>` is the name of the file to be printed. This printer is free and useful for short program listings.

5.8 Manipulating files

Files are normally created with an editor (*e.g.* `emacs`), or from the output of programs or applications. Other useful file manipulation commands are:

```
cp file1 file2  make a copy of file1, named file2
mv file1 file2  move (rename)file1 to file2
rm file1        remove (delete) file1
```

These utilities can also be used with directories in the filenames, in particular `mv` can be used to move files into a directory for example if `checkpoints` is the name of a directory, then

```
mv CheckPoint1.java checkpoints
mv CheckPoint2.java checkpoints
```

will move the two files, `CheckPoint1.java` and `CheckPoint2.java` into the directory `checkpoints`.

Warning: files that are deleted with `rm` are really gone. There is **no** Waste Basket in Linux.

5.9 Wild-cards in file names

Wild-cards characters allow matching of a range of filenames. The two commonly used wild-cards are:

- * matches any string of characters
- ? matches any single character

so for example

```
rm *.class      removes all files with extension .class
ls CheckPoint?.java  lists all files called CheckPoint<any>.java where <any> is
                    any character
```

5.10 e-Mail

You do not have a separate e-mail account on this system. You should continue to use your University SMS account which is available through the Firefox web browser.

man pages

All Linux commands, most applications and most functions are all documented via the on-line system manual pages. These are very complete, usually being written by the actual programmer who implemented the command or application. As a result they usually assume a level of knowledge about the system well above that possessed by the average (never mind the novice) user. The quality, readability and length of man pages is very variable, however they do *usually* contain the correct information!

The man pages are accessed by,

- Type `man <command>` in the terminal window. For example to get the man page for `ls` you simply type `man ls`. This will output the manual page to the terminal window one page at a time with the same page commands as `less` (see above).

In general, you will obtain more useful information by Googling for the command you are interested in.

5.11 The Shell

When you type commands they are interpreted by the program called the `shell`. Under Linux there are many different shells (*eight* I know of) all of which do the same basic task, but with subtle and annoying differences. All your accounts have been set-up to use `bash` (**b**ourne **a**gain **s**hell), which is possibly the most popular and useful variant.

5.12 More Information

This section contains the “absolute minimum to get by” with Linux. Extensive additional documentations are available:

1. Online via Google. The best first point of call.
2. Any and varied books on Linux in the library and for purchase in bookshops.

What Next?

Now you have some practice of using the basic Linux commands you should now read the next section on the `emacs` editor. You will need to use this to write your first Java program.

6 The emacs editor

You should read through this section, ideally sitting in front of a terminal. Try starting up the editor and practicing the various editing operations.

6.1 Introduction

emacs is a very powerful screen based editor available on almost all Linux machines. This editor has a huge range of features which take a complete book to document. Most people, however, only use a tiny fraction of these features and in particular all the common operations can be accessed by the menus.

emacs is a language sensitive editor having various features that assist you in writing programs, such as colour highlighting and automatic parenthesis checking. This feature is most useful when using programming languages like Java where a single missing “{” or “;” can result in serious problems.

6.2 Starting up emacs

To create a new file, or edit an existing file called, for example, `Hello.java` you type the following command in the terminal window.

```
emacs Hello.java &
```






the emacs window will then appear. If the file exists it will be read, else a blank screen will appear.

1. The “.java” extension to the filename tells emacs that you are editing a Java program. This will switch-on the correct syntax checking.
2. The “&” means “run as detached process” which creates the emacs window but then leaves the terminal window free to type other commands.


6.3 Simple Editing

Editing using emacs is very similar to using the notepad editor or Word under Windows.


Basics

1. Move the cursor into the emacs window to make the window *Active*..
2. Typing will insert text at the solid block cursor.
3. Return or  key will insert a newline.
4. Backspace key will delete character *to left* of the cursor.
5. Delete key will delete character that the cursor is positioned on.
6. Move the cursor using the arrow keys    .

Mouse Editing

Mouse based editing uses a combination of the two cursors, the *text* cursor (solid magenta block) and the *mouse* cursor (thin magenta ). You also use the menus at the top of the window.

1. To move *text* cursor with mouse: Move mouse cursor to required position in text and click the left button.
2. To *Select* a region of text place the mouse cursor at the start of the region to be selected and left-drag to the end.
3. To Cut or Copy a *selected* region select “Cut” or “Copy” from the “Edit” menu.

4. To Paste a *selected* region select “Paste” from the “Edit” menu.
5. To clear (or delete) a *selected* region select “Clear” from the “Edit” menu.
6. To search for a string in the text,
 - Select “Search...” from the “Search” menu.
 - Select Prompt “Search for string: ” appears at the bottom of the screen and the text cursor moves there.
 - Type in search string followed by “Return” or . Text cursor goes to start of the search string in the text window, (if it exists).
 - The search can be repeated with “Repeat Search” or “Repeat Backwards” from the same menu.

Saving Changes and Exiting

All saving is controlled by the “Files” menu as follows:

1. To Save the text,
 - Select “Save Buffer” from the “Files” menu.
 - This will write the current text to the current file. This will **overwrite** the current contents of the file.
 - If you want to save current text in a different file, select “Save Buffer as ...” and you will be prompted for a new filename with a “Write file:” prompt at the bottom of the screen. Type in the new filename followed by “Return” to save.
2. To Exit emacs, select “Exit Emacs” from the “Files” menu. If you have not saved the text buffer before doing this you will be warned and given a chance to save it before exiting.

This is all you really need to know about emacs.

6.4 Things that go wrong

There are a few things that can appear to “go wrong” when using emacs all of which are easily cured. Error messages appear on the bottom line, usually accompanied with a BEEP from the terminal.

- BEEP plus message about “*minibuffer in minibuffer*”: You clicked twice in a menu and confused things, click once in the text area with the left mouse button and try again.
- Window splits in two, both rather small, then either:
 1. You asked it to with the “Split Window” option.
 2. You typed <TAB> in the “Open File...” option and started the file browser.
 3. You were “playing” with the (so called) “Help” menu.

to get out of this:

1. Select the part of the window you want, (usually your program) by clicking in it once with the left mouse button.
 2. Select the “Unsplit Windows” option from the “Files” menu.
- You get some strange prompt at the bottom of the screen like C-x-, or M# and the text cursor moves to bottom of the screen. You have switched emacs into command mode.
 - Type Ctrl-G (hold down Ctrl and type G).
 - Repeat typing Ctrl-G until a Quit message appears at the bottom of the screen. Normality will be restored!
 - When typing you “overwrite” the text and the Delete key does not remove spaces: You have switched to *overwrite* mode by pressing the Insert key (beside the Delete key!).

1. Press the Insert key (again).
 2. The string `Ovwr` will be removed from the black information line at the bottom of the screen, and all will work again.
- The whole system “stops” and you think you have “*lost your whole editing session*”. No, emacs saves changes every 50 or so key-strokes so you will not have lost much.
 1. The “saved” file is called `#<filename>#` in the current directory.
 2. Rename this file to something more sensible using `mv`.
 3. Then edit this new file and you will find that almost everything is still there!
 - Other odd errors or odd behaviour: Call a demonstrator, or see Mrs McIvor in the laboratory.

What Next?

You are now ready to write your first Java program. Follow the instructions in the next section.

7 Your First Java program

You should work through this section sitting at a terminal and make sure you get it to work.

It is *traditional* in learning any programming language that your first program prints ``Hello World!`` to the screen. The code of such a simple program (in Java with explanatory comments) is shown below:

```
/*
 * Simple "Hello World" program
 *
 * These lines, between '/*' and '*/', are comments. They
 * have no effect on the program's function but are here
 * to aid and explain the code.
 */

// You can also have a single line comment. After the '//' is a comment.

/*
 * In java, all code must be contained within a 'class'. This class name
 * must begin with a capital letter and must match the file name (the
 * file containing this class must be called 'Hello.java'). All the
 * classes you will deal with must have the 'public' keyword - we
 * will discuss this in more detail towards the end of the course.
 * Everything within the class is contained within the braces ({ }).
 */
public class Hello {

    /*
     * The line below is declaring a 'class method' called 'main'.
     * Initially, all the actual code for your programs will be
     * contained within this method. The 'public' keyword specifies that
     * we can access this method; 'static' states that it is a class
     * method (more on this at the end of the course); and 'void' states
     * that the method does not 'return' any value (more about this
     * towards the end of the course). The '(String args[])' are known
     * as the 'arguments' of the method - they are values that come from
     * outside the method - we will discuss this in more detail later on
     * in the course. Again, braces enclose all of the code within this
     * method.
     */
    public static void main(String args[]) {

        // Print 'Hello World!' to the screen
        System.out.println("Hello World!");

        // Stop the program and return control to the terminal
        System.exit(0);

    }

}
```

This simple program introduces a few important concepts of Java programming.

1. **Comments**—Comments should be used liberally throughout your code. They allow other programmers to understand what you are doing and also serve to remind you what is going on when you come back to some code after a break.
2. **Blocks**—Any set of programming statements between two matching braces are known as a “block”. The

simple program above contains two blocks: one corresponding to the “Hello” class and one corresponding to the “main” method.

3. **Statements**—Programming statements are lines that instruct the computer on what to do. All statements in Java must end with the semicolon character (except statements that begin a block such as class and method specifiers).

7.1 Your First Program

You are strongly recommended to follow this task.

In an *active* terminal window,

1. Make a directory called `sciprog` to hold your computing examples by typing:

```
mkdir sciprog
```

2. Make this new directory your *present working directory* by typing:

```
cd sciprog
```

3. Now open a new empty file for editing by typing:

```
emacs Hello.java &
```

to start the editor. Note the “&” character means “run the editor in background” which will allow you to use the terminal window at the same time. Type in the Hello World program listed above (minus the large comments) and save it.

4. *Compile* the program with

```
javac Hello.java
```

which will create an *byte code* file called “`Hello.class`”. (check that this file exists with the `ls` command)

5. To run the program type

```
java Hello
```

If “Hello World!” appeared in the terminal, then ..., congratulations, you have just written your first Java program.

Examples

The following on-line example is available:

1. Source of `Hello.java`

What Next?

You are now ready to undertake Checkpoint 1.

8 Checkpoint 1

Aim of Checkpoint

This checkpoint demonstrates that you can write, compile and execute a basic program that is a simple modification of the minimal “Hello World” Java program in the previous section.

This checkpoint is worth **5%** of the course mark.

Submission Dates

It is expected that this checkpoint is completed during the **first** laboratory session.

Final submission date for this checkpoint is: **5.00 pm, Thursday 2 October.**

Computing Task

Modify the minimal “Hello World” program from the previous section to print out your name, home address and e-mail in a nice format using `System.out.println` statements.

When you have completed this:

1. Call a demonstrator.
2. Show them the code of your program, the compilation and demonstrate the working program.
3. Make sure they tick-off your name against Checkpoint 1.

Note: Make sure that you have tested your program and that it works before you call a demonstrator to mark it.

Material Needed

For this checkpoint you need to have covered:

1. “Getting Started”
2. “Basics of Linux”
3. “The emacs editor”
4. “Your First Java program”

What Next?

You will need to read and understand the next three sections (“Variables, Data Types and Operators”, “Formatted Input and Output” and “The Math Class”) before the next checkpoint. They are fairly long and probably best read away from the terminal with a strong cup of coffee.

9 Data Types, Variables and Operators

This fairly long section must be read and understood before you proceed. It also contains a considerable amount of syntax information that you will want to refer to throughout the course.

9.1 Introduction

Computer programs are based on the manipulation of different types of data, held in *variables*, by using various types of arithmetic and logical *operations*. This section defines the various types of data and the basic operations available to you in Java.

9.2 Data Types

The basic data types are:

Integer—Positive or negative whole numbers normally specified in *Decimal* format for example 10, -9, 854.

Boolean—being either *true* or *false*.

Floating Point—Positive or negative rational numbers held in mantissa plus exponent form, ($0.nnnnnnnn \times 10^{mm}$). These are specified with a decimal point or optional exponent, for example,

10.0	=	10.0
-3.14	=	-3.14
4.26e8	=	4.26×10^8
-10.335e-15	=	-1.0335×10^{-14}

Character—Upper and lower case alphabet, the ten digits, symbols such as + - % & etc and a range of non printable control characters such as TAB, LINEFEED, BELL etc. These are specified by the character in single quotes, for example:

Character	Meaning
'0'	The letter '0'
'1'	The letter '1'
'A'	The letter 'A'
'b'	The letter 'b'

In addition there are a range of characters that are either not directly available from the keyboard or are used for other things. These are represented by pairs of symbols but are actually *single* characters. These are:

Character	Action
\0	Null
\b	Backspace
\t	TAB (Horizontal Tab)
\n	Linefeed (New line)
\v	VT (Vertical Tab)
\f	Formfeed (New page)
\r	Carriage Return (Beginning of line)
\"	Double quote
\'	Single quote

Characters can also be specified by their (Octal) numerical value, detail in advanced books.

Strings—Which are lists of *characters* surrounded by “double quotes”, for example

"Hello World!"

is a *String* containing 12 characters. Note the " " are not part of the *String*.

9.3 Variables

To manipulate the above data types and be able to perform arithmetic or logical operations they must be stored as *variables* in computer memory. A *variable* is a location in computer memory that your program can *read* and *write* to. Each *variable* has **three** properties, these being:

1. its *name*, which you choose,
2. its *value*, which you set,
3. its *address* (or location) in the computer memory, which is chosen for you.

In Java the most basic *variable* types are:

`boolean` which can take the values `true` and `false` only.

`char` 16-bit Unicode character. (not used in this course).

`int` Used to hold an integer value. In this implementation the `int` is 32-bits long and can hold integer values from -2147483648 to 2147483647 .

`double` Used to hold a *floating point* number in mantissa plus exponent form, $(0.nnnnnnnnn \times 10^{mmm})$. The `double` is 64-bits long. The mantissa is 14 decimal places, and maximum and minimum is $\pm 1.79769 \times 10^{308}$.

In addition to normal numerical values, `double` has three special non-numerical values to represent $\pm\infty$ and “Not-a-Number” normally used for error conditions. See later in this section.

`String` Used to hold a list of characters. Technically a `String` is a list of `chars` and is not really a separate variable type but is used so extensively it is easier, at this point, to consider it as a separate variable type.

There are additional basic variable types, being `byte`, `short`, `long` and `float` which shall not be used in this course.

Clearly the use of a single *variable* is a bit limiting; we will see how to group these into *arrays* later.

9.4 Variable Names and Declaration

Before we can use a *variable* it must be *declared*. This both defines the *name* **and** allocates a memory location in which its *value* can be stored. Java has a very flexible set of rules for the variable names, these being:

1. Any combination of letters³ and numbers including `_` (underscore).
2. Must start with a letter. (By convention a lower case letter).
3. Upper and lower case letters *are* different.
4. Must not clash with any of the Java *keywords* in Table 1 which have special pre-defined meanings.

When you are deciding on the *name* of a variable make the name describe its purpose. This makes the program much easier to read. Typical declarations may be:

```
int counter;           // Declare an integer
double errorValue;    // Declare a double
String myName;        // Declares a string
```

You will learn more about declaration later.

³Due to the use of the Unicode character the definition of ‘letters’ and ‘numbers’ is very wide. Best stick to normal keyboard characters, the use of Japanese, Tibetan, Hebrew etc. can result in confusion and problems with printing and viewing !

abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	
continue	goto	package	synchronized	

Table 1: Table of keywords

9.5 Assignment

Once we have declared a *variable* we can set its *value* which is done with the very misleading “=” sign. For example if we have:

```
double xValue;
xValue = 5.34e-6;
```

This means “xValue” is the *name* of a double variable, and we have set its *value* to be 5.34×10^{-6} .

Note: As mentioned in the previous section, each statement in the program must be terminated with a “;”.

The assignment can contain arithmetic expressions, for example

```
double xValue;
xValue = 10.0 + 13.8 + 56.6;
```

which sets the *value* of xValue to 80.4. See more details of this below.

Multiple variables can be defined as for example;

```
double xValue, yValue;
xValue = 10.0;
yValue = xValue + 25.0;
```

declares two double variables. The *value* of xValue is first set to 10.0, then the *value* of yValue is set to the *value* of xValue plus 25.0, being 35.0.

Note: The order of the statements is significant. Also subsequent changes to the value of “xValue” do **not** change the value of “yValue”.

Or multiple occurrences of the same variable, for example:

```
double xValue;
xValue = 10.0;
xValue = xValue + 25.0;
```

which declares one double variable xValue. The *value* of xValue is first set to 10.0. The second statement then says,

1. Take the current *value* of xValue. (10.0)
2. Add 25.0, (to get 35.0)
3. Set the new *value* of xValue to be 35.0,

so the “old” *value* of xvalue has been lost.

This idea is somewhat confusing on first encounter, but remember the “=” sign means “*set value to*” and is **not** the algebraic equals.

9.6 Basic Arithmetic

The basic arithmetic operators are,

Symbol	Operator
*	Multiply
/	Divide
%	Modulus (integer only)
+	Plus
-	Minus
++	Increment (integer only)
--	Decrement (integer only)

which are evaluated in “left-to-right” and in the normal order of precedence (Multiply/divide (and modulus) before addition/subtraction), with parenthesis () being used to alter this precedence if required.

Floating Point Arithmetic

When all the variables (or constants) are *floating point*, typically double, then the arithmetic obeys the normal arithmetic rules. For example;

$$y = \frac{2(3x + 6)}{(4.3x^2 + 6.9)}$$

is coded as:

```
double x,y;
x = 10.0;
y = 2.0*(3.0*x + 6.0)/(4.3*x*x + 6.9);
```

Note: there is no “power” operator. It is implemented as the function `Math.pow` detailed in a later section.

In arithmetic expressions SPACES, TAB(S) and NEWLINEs are ignored but the whole expression *must* be terminated by “;”.

It is good practice to space out arithmetic expressions with SPACE(s) and NEWLINE(s). Also *extra* brackets () are permitted. This makes them *much* easier to read, and consequently *much* easier to spot a mistake. A few extra keystrokes while writing a complex arithmetic expression takes about 3 or 4 seconds. If this results in you getting it *right* it can save *hours* of frustration and mental anguish!

Integer Arithmetic

The basic rules are as for *floating point* except for three additional operators and divide “/” being different.

Divide—Integer divide gives the “integer part” of the division only, so that :

$$4/3 = 1 \quad \text{and} \quad 4/5 = 0$$

Modulus—(%). This gives the “remainder” following integer division, so that

$$4\%3 = 1 \quad \text{and} \quad 4\%5 = 4$$

Increment/Decrement—(++ and --) add and subtract 1 (one) from an integer. Further details of these will be discussed later in the course. These operators are mainly used in loops to make programs more efficient but can (unfortunately) result in unreadable code.

Mixed Integer/Floating Point Arithmetic and Casting

In Java conversion from one type of data type to another requires thought, and the idea of “casting”. At this stage we will only consider conversion between ints and doubles.

The “cast” operators are

```
(double) ⇒ convert to double
(int)    ⇒ convert to int
```

where the () are required.

Integer to Double Conversion—is the simplest with,

```
int iValue = 12;
double xValue;
xValue = (double)iValue;
```

which will result in xValue being set to value 12.0 exactly as expected.

This casting can also be used in arithmetic expressions so that,

```
double xValue;
int iValue = 10, jValue = 3;
xValue = (double)iValue/(double)jValue;
```

this will result in a *floating point* divide and so xValue = 3.333.. as expected since it will convert value of iValue and jValue to double *before* the division.

However if you wrote the above example with,

```
double xValue;
int iValue = 10, jValue = 3;
xValue = (double)(iValue/jValue);
```

then the value of xValue would be 3.0, since we have an integer division, the result of which will be converted to a double.

Aside: In many occasions the (double) cast is optional and will occur automatically, however it is “good programming practice” to include it, it also help to make the code easier to read and understand.

Double to Integer Conversion—Clearly since int can only represent whole number we have to loose the decimal fractions. Thus

```
double xValue = 5.674;
int iValue;
iValue = (int)xValue;
```

will result iValue being set to 5. Note: the value is truncated and **not** rounded. See the section on Mathematical Function if you want to round.

Since we “loosing something” in this conversion the (int) cast is *never* optional. You will get a compile time error if you miss it out.

9.7 Arithmetic Assignment Operators

There are many instances when we want to perform an arithmetic operation on a *variable* and overwrite its *value* with the new calculated *value*, for example to add 10 to an int called val we can use,

```
val = val + 10;
```

However this construction occurs so frequently there are assignment versions of the arithmetic operators allowing this to be written as

```
val += 10;
```

which means:

1. take the *value* stored in val
2. add 10 to it to obtain a new *value*,
3. set val to this new *value*.

There are also assignment versions of the other operators, these being

Symbol	Operator
<code>*=</code>	Multiply by
<code>/=</code>	Divide by
<code>%=</code>	Modulus of (integer only)
<code>+=</code>	Add to
<code>-=</code>	Subtract from

These assignment operators should be used with care since they can easily result in your code being difficult to understand.

9.8 Strings

This short section on Strings is sufficient for the initial part of the course. There is a more detailed discussion later. Strings are lists of characters that can be declared, set to a value and “added” to. Note: unlike most other computer languages Strings in Java are **not** of fixed length.

To declare a String simply use

```
String aName = "Fred Smith";
```

which creates a String of 10 characters including the Space, but excluding the " which mark the beginning and end.

The “addition” of Strings “adds” or concatenates one on to the end to the end of the other exactly as expected, so for example we can re-write the basic “Hello World!” program as:

```
//
//      Hello World by adding strings.
//
public class StringExample{
    public static void main(String args[])
    {
        String firstWord = "Hello";
        String secondWord = "World";
        String sentence;

        sentence = firstWord + " " + secondWord + "!";

        System.out.println(sentence);
        System.exit(0);
    }
}
```

Here sentence is a String made up from the “addition” of **three** Strings.

Conversion of other data types to Strings

All of the other basic data types, boolean, int, double and char can also be converted to Strings by simple “adding” them to an existing String. For the data types used in this course we get.

Type	String format
int	Decimal value
double	Decimal value in fixed point format
boolean	Either “true” or “false”
char	The printable character

This gives us our first method for output of a value since the following code fragment

```
int iValue = 256;
String answer = "The value of iValue is : " + iValue;
System.out.println(answer);
```

will output The value of iValue is : 256.

This method of “adding” variable to Strings is the main method of formatted output in Java. It works well for all but double where we tend to get huge numbers of unwanted significant figures. We will see how to control that in the next section.

Conversion Strings to other data types

A task often encountered is the need to convert a number in a String variable (e.g. the string “3.142”) to a numerical value that we can use in calculations (in the example provided this would be a double). This operation most frequently occurs when we have read in some data from the command line or from a file. Almost always, this data will be in String format but we want the numerical equivalent.

The term for this operation is *parsing* and Java provides a set of standard tools to do this.

Converting a String to an int

This is done using the Integer.parseInt function. For example, this code fragment:

```
String intString = "512";
int iValue = Integer.parseInt(intString);
```

would set the value of iValue to 512.

Converting a String to a double

This is done using the Double.parseDouble function. For example, this code fragment:

```
String doubleString = "3.142";
double dValue = Double.parseDouble(doubleString);
```

would set the value of dValue to 3.142.

Converting a String to a boolean

This is done using the Boolean.parseBoolean function. For example, this code fragment:

```
String boolString = "True";
boolean bValue = Boolean.parseBoolean(boolString);
boolString = "tRUE";
bValue = Boolean.parseBoolean(boolString);
boolString = "true";
bValue = Boolean.parseBoolean(boolString);
```

would set the value of bValue to true in all cases. If the String contains the word “true” irrespective of case then the value will be parsed as true.

Unfortunately the parse methods fail if there are stray leading or trailing *whitespace* characters in the String⁴. To prevent this problem it is better to use the String method trim() which returns a String with any leading or trailing whitespace characters removed. So a better, and more robust, use of the parse methods is

```
String intString = " -45";
String doubleString = "45.7643e-3 ";

int iValue = Integer.parseInt(intString.trim());
double dValue = Double.parseDouble(doubleString.trim());
```

which, correctly, ignores leading or trailing spaces

Examples

Source code for the following on-line examples are available,

- Floating point arithmetic in DoubleExample.java.

⁴This is the only place I know in Java where extra spaces do really matter.

- Use of brackets in floating point expressions in BracketExample.java.
- Use of integer arithmetic in IntegerExample.java.
- Use of casting between data types in CastingExample.java.
- Simple use of Strings in StringExample.java.

What Next?

you will need to read the next chapter about basic input/output before you are ready to write a real program.

10 Basic Input and Output

10.1 Introduction

Before we start any real programming it is essential that we can read-in and write-out information, this is usually known in the computing world as *IO*. In particular, we want to read and write the values of *variables*. In Java input and output is very flexible, and as a consequence, it can be rather complex.

Generally, you will need to import external classes to perform any sort of useful input or output. In this section we will learn how to perform basic text *input* and *output* to the command line using the standard *Console* class. Further on in the course we will meet ways to perform IO using files and the *ptplot* package which will allow you to display your output graphically.

10.2 The Console Class

The *Console* class provides standard way to read data from and write data to the terminal window. Although we have already met a way to write to the terminal window in the previous section (`System.out.println`) the *Console* class provides more flexibility.

In order to use the *Console* class in your program you must first *import* it. This is done by adding the following line to the top of your Java code (before the class specifier).

```
import java.io.Console;
```

(Importing external classes in this way is used extensively within Java programming and we will meet this syntax throughout the course.)

Once the *Console* class has been imported you can use it in your own code. You will usually want to create a *Console* that is associated with the terminal that is running the program. You do this with the following line:

```
Console myConsole = System.console();
```

This creates a new *Console* object called `myConsole` that is associated with the *System* console (*i.e.* the current terminal).

10.2.1 Writing data using the *Console* class

You can write simple data to the *Console* using the `printf()` method (note that the `println()` method does not work with the *Console* class), *e.g.*

```
myConsole.printf("Some simple output.");
```

For more complex output we can use the `printf()` method which is covered in more detail below.

10.2.2 Reading data using the *Console* class

We read data from the *Console* using the `readLine()` method. This method takes a *String* argument that is the prompt to print and returns a *String* variable, *e.g.*

```
String myString = myConsole.readLine("Please enter a value: ");
```

will print "Please enter a value:" to the terminal and wait for input (which is entered by typing and finishing by pressing Return). The input that was typed at the terminal will be stored in a *String* variable called `myString`.

A slight complication in using the *Console* class to read data is that all entries are read as *Strings*. In order to use any numbers that we read in, we must convert the *String* we have read to either an *int* or *double* variable (we learned how to do this in the *Variables* chapter). Similarly, to use a value we have read in as a *boolean* variable we must first convert it.

The use is best explained by an example that simply reads in an *int* and prints it out again:

```
// Import the Console class
import java.io.Console;

// The class (note: this must be in a file called 'ReadInteger.java')
public class ReadInteger {
    public static void main(String args[]) {

        /*
         * Create a Console instance to read-from/write-to
         * using 'System.console()' ensures that we are accessing
         * the terminal we are running the program in
         */
        Console myConsole = System.console();

        /*
         * Read a string from the console with the prompt:
         * 'Enter an integer number: '
         * Store the string in 'myString'
         */
        String myString = myConsole.readLine("Enter an integer number: ");

        // Convert the String to an integer value and store it in 'myInt'
        int myInt = Integer.parseInt(myString.trim());

        /*
         * Print the value of the integer back to the console. We use the
         * 'printf()' method for this as Console does not support
         * 'println()'. 'printf()' will be covered in more detail in the
         * next section. The only difference from 'println()' in this
         * case is that we need to add a newline character to the end of
         * the String.
         */
        myConsole.printf("The integer is " + myInt + "\n");

        // Exit gracefully
        System.exit(0);
    }
}
```

You should type-in this program, calling it `ReadInteger.java` and make sure it works.

10.3 Nicier Output Formatting using `printf()`

We have seen in previous sections and in the above example, that we can display the value of a `int` or `double` by simply “adding” it to a `String`, and then printing the `String`. This works well for `ints`, but with `double` it gives all available digits, typically 14, which is often excessive.

Java (and many other languages such as C and Perl) provide a simple but powerful way to format the output of data using the `printf()` method.

`printf()` has many different options and ways of formatting so we will cover the most basic and common uses here. We will cover more functionality in later sections as we need it and further documentation is widely available on the internet.

The general syntax of the `printf()` method, which can take a variable number of arguments, is:

```
myConsole.printf(String template, Object, Object, .... )
```

where the `String` template specifies how the `Object` arguments are to be formatted. This rather complex looking syntax is best explained by a couple of examples. For example:

```
int intValue = 15;
double doubleValue = 12.546786534;
Console myConsole = System.console();
myConsole.printf("Values are %d and %f\n", intValue, doubleValue);
```

will print to the terminal:

```
Values are 15 and 12.546787
```

with the `%d` being the location of the integer formatted in decimal, and `%f` being the location of the double being formatted in fixed point format which, by default, gives 6 decimal places.

Note the `"\n"` at the end of the template string, this means *newline character* and makes sure that we get a carriage return at the end of the line.

There are two other important keys for doubles, these being,

`%e` formats a double in exponential notation with a default of 6 decimal places with correct rounding.

`%g` formats a double in general notation with 6 significant figures using exponential notation or fixed point notation depending upon the size of the number.

You can also control the number of significant figures, *or* number of decimal places as follows,

`%.3f` will format a double in fixed point format with 3 decimal places;

`%.5e` will format a double in exponential format with 5 decimal places (*i.e.* 6 significant figures);

`%.7g` will format a double in general format with 7 significant figures.

So, for example:

```
double plank = 6.67E-34; // Plank's Constant
double lightSpeed = 2.999867E8; // Speed of light.
Console myConsole = System.console(); // Set up the console
myConsole.printf("Plank's constant is %.3g\n", plank);
myConsole.printf("Speed of light is %.4e\n", lightSpeed);
```

will give as output:

```
Plank's constant is 6.67e-34
Speed of light is 2.9987e8
```

Here is an example that reads in a wavelength in nm and converts it to metres.

```
// Import the Console class
import java.io.Console;

public class Convert {

    public static void main(String args[]) {

        // Set up the console for terminal input/output
        Console myConsole = System.console();

        // Get the wavelength in nm
        String myString = myConsole.readLine("Enter the wavelenght (nm): ");
        double lambda = Double.parseDouble(myString.trim());

        // Convert to metres
        lambda = lambda / 1.0e9;
```

```
// Write the value to the terminal
myConsole.printf("Wavelength in metres to 3 places %.3e\n",lambda);
myConsole.printf("Wavelength in metres to 5 places %.5e\n",lambda);

// Exit gracefully
System.exit(0);
}
}
```

You are strongly encouraged to type this program into a file called `Convert.java` and make sure that it works and you understand it. You should then vary the format keys and see what happens, things to try are,

1. Try `%.4f` what happens and why?
2. Try an illegal key, for example `%d` which will try and format a double as an integer, again see what happens.

Note: This example only works once, you will see in the section on loops how to make this program “loop-round” continually asking for input.

Examples

The following on-line source examples are available

- Read an single integer `ReadInteger.java`
- Read multiple variables `ReadVariable.java`
- Format a double with specified template `FormatExample.java`

What Next

You should read the next section (Math Class and Constants) before attempting Checkpoint 2.

11 Math Class and Constants

11.1 Introduction

We saw in the last section that we can have *classes* that contain *methods* that read and write data from the terminal. In this section we look at a *class* that contains *methods* to calculate mathematical functions, for example \sqrt{x} , $\cos()$, etc. and return their numerical value.

11.2 The Java Math class

To use the mathematical methods defined in the external Math class you need to import the class into your program by adding the following line to the top your Java source code (above the class definition):

```
import java.lang.Math;
```

All the mathematical functions are defined to take arguments of type double and return a double *value*. They are simply used by `<Class>.method(<variable>)>`. For example to use the `sqrt()` method you write,

```
double x = 56.90;
double y = Math.sqrt(x);
```

which will set `y` to be the square root of `x`.

11.2.1 Trigonometric Functions

Method	Action
<code>cos(x)</code>	$\cos(x)$
<code>sin(x)</code>	$\sin(x)$
<code>tan(x)</code>	$\tan(x)$
<code>acos(x)</code>	$\cos^{-1}(x)$
<code>asin(x)</code>	$\sin^{-1}(x)$
<code>atan(x)</code>	$\tan^{-1}(x)$
<code>atan2(y,x)</code>	$\tan^{-1}(y/x)$
<code>toRadians(d)</code>	Converts degrees <code>d</code> to radians
<code>toDegrees(r)</code>	Converts radians <code>r</code> to degrees

All angles are in radians.

The only complication is the two $\tan^{-1}()$ functions. `atan(x)` returns the normal $\tan^{-1}(x)$ in the range $-\pi/2 \rightarrow \pi/2$ while `atan2(y,x)` returns the $\tan^{-1}(y/x)$ in the range $-\pi \rightarrow \pi$ where the quadrant is given by the signs of *both* `x` and `y`.

11.2.2 Exponential, Power and Hyperbolic Functions

Method	Action
<code>exp(x)</code>	$\exp(x)$
<code>expm1(x)</code>	$\exp(x) - 1$ valid for <i>small</i> <code>x</code>
<code>log(x)</code>	$\log_e(x)$
<code>logp1(x)</code>	$\log_e(x+1)$ valid for <i>small</i> <code>x</code>
<code>log10(x)</code>	$\log_{10}(x)$
<code>pow(x,y)</code>	x^y
<code>sqrt(x)</code>	\sqrt{x}
<code>cbrt(x)</code>	$\sqrt[3]{x}$
<code>cosh(x)</code>	$\cosh(x)$
<code>sinh(x)</code>	$\sinh(x)$
<code>tanh(x)</code>	$\tanh(x)$
<code>hypot(x,y)</code>	$\sqrt{x^2 + y^2}$

All as expected.

11.2.3 Absolute values, Nearest Integers, Max/Mins

Method	Action
<code>abs(x)</code>	$ x $ (any numeric data type)
<code>ceil(x)</code>	Smallest integer value not less than x
<code>floor(x)</code>	Largest integer value not greater than x
<code>rint(x)</code>	x rounded to nearest integer (returns double)
<code>round(x)</code>	x rounded to nearest integer (returns long)
<code>max(x,y)</code>	Larger of x and y (any numeric data type)
<code>min(x,y)</code>	Smaller of x and y (any numeric data type)
<code>signum(x)</code>	Sign of x , being ± 1.0 or 0.0 if $x = 0.0$.

There are a couple of complications here:

1. `abs`, `min` and `max` work for all numeric data types, `int` or `double` in this course, and return the same data type that they were passed.
2. `round` return a long which is a 64 bit integer which you need to convert to normal `int`. *i.e.*

```
double x = 2.7;
int i = (int)Math.round(x);
```

11.2.4 Random Numbers

`random()` | Random double value between 0 and 1.

Simple random number generator, see class `java.util.Random` for a better version.

11.3 Constants

In addition to numeric variables, each data type can have *Constants*. These can be used in arithmetic and logical expressions exactly like *variable* but their value **cannot** be altered. Such constants are extremely useful for numerical calculation, for example involving π , checking for errors, or ranges of loops etc. without having to “hard code” explicit numbers into your program. The use of such constants make code much more readable, maintainable and also more likely to be correct. There are two types of constants, these being **predefined** and **user-defined**.

Predefined Constants

The useful predefined constants for novice users are detailed below. In addition there are many additional constants used to control operation of various class types especially if you want to use the full Java graphics or menu libraries. There are detailed in the relevant books.

Defined in Math Class:

The following double constants are defined in the `Math` class.

<code>Math.E</code>	$e = 2.718\dots$
<code>Math.PI</code>	π

This rather limited range of constants *will* be expanded in future versions of Java as it becomes a more established scientific language.

Maximum and Minimum Constants:

Each primitive data type has a Maximum and Minimum storable value, for the `int` and `double` these are:

<code>Integer.MIN_VALUE</code>	Minimum int value	-2147483648
<code>Integer.MAX_VALUE</code>	Maximum int value	2147483647
<code>Double.MIN_VALUE</code>	Minimum double value	4.9E-324
<code>Double.MAX_VALUE</code>	Maximum double value	1.7976931348623157E308

Double Error Constants:

In addition, for the `double` type there are three additional constants that are normally used to signify an error condition, these being:

<code>Double.POSITIVE_INFINITY</code>	$+\infty$
<code>Double.NEGATIVE_INFINITY</code>	$-\infty$
<code>Double.NaN</code>	Not-a-Number

The $\pm\infty$ results if the \pm range is exceeded, this is normally the result of dividing by zero. The “Not-a-Number” constant is used to represent an “illegal value”, for example if you call `Math.sqrt` with a negative number.

Printing variables set to these values will result in the character strings `Infinity`, `-Infinity` and `NaN` respectively.

Note: Trying to do any other arithmetic on a `double` variable set to any of these constants will result in your program crashing, a good thing as it signifies that something has gone horribly wrong!

User Defined Constants

Any data type can be defined as being *Constant* by use of the `final` qualifier, for example to declare a constant `double` or `int` you simple use.

```
final double LAMBDA = 550e-9;
final int MAXIMUMVALUE = 1000;
```

The value **must** be set when the constant is declared and this value **cannot** be altered. If you try and overwrite this constant your program will crash with an error.

Where and why should you use *constants*,

1. When you have a fixed physical constant, for example “charge on electron”, ϵ_0 , h etc. It saves you typing long strings of digits multiple times, (with associated typing errors!).
2. When you have a fixed variable, for example in an optical modeling program you may have a fixed λ . Using a *Constant* prevents fixed numbers appearing in the code. This makes the code **much** easier to modify later on. For example to change to model to a different λ you only need to change one line.
3. It makes the code much more readable, especially if you call your constants by *sensible* names, like `MAXIMUMVALUE`. This results in pieces of code like,

```
if (currentValue < MAXIMUMVALUE)
```

which is easier to understand and debug.

By “convention” *Constants* in Java are in “UPPERCASE”. This makes them easy to find in the code and also ensures that the colour coding in editors (such as `emacs`) are correct.

Examples

The following on-line source examples are available

- Calculate Plank radiation distribution

$$\rho(\lambda) = \frac{8\pi hc}{\lambda^5} \left[\frac{1}{\left(\exp\left(\frac{hc}{\lambda T}\right) - 1\right)} \right]$$

PlankRadiation.java

What Next?

You should now attempt Checkpoint 2.

12 Checkpoint 2

Aim of Checkpoint

This Checkpoint consists of **two** short programs to demonstrate basic input/output of variables and basic arithmetic.

This checkpoint is worth **10%** of the course mark.

Submission Dates

It is expected that this checkpoint is completed during the **second** laboratory session.

Final submission date for this checkpoint is: **5.00 pm, Thursday 2 October**.

Computing Task A

Write a Java program to read a double value from the terminal and print out the value:

1. when it is converted to an int,
2. of the difference between the int value and the original double value.

Test your program for both positive and negative numbers.

Computing Task B

Write a Java program to calculate the surface area and volume of a sphere. Your program should:

1. Prompt you for the radius of the sphere in **mm**.
2. Calculate and display the surface area and volume in **m²** and **m³** respectively.

Check that your program works for radii in the range **0.01 mm** → **10 m** and that the output is formatted in a readable form with **5 significant** figures.

(**Note:** Ensure you keep this code in a working format as you will reuse it in the next checkpoint.)

End of Checkpoint

When you have completed **both** programs, call a demonstrator and show them the code, compile it and demonstrate the working programs. This is the end of **checkpoint 2**. Ensure that the demonstrator checks off your name.

Note: Make sure you have tested your program and that it works before calling a demonstrator to mark your work.

Material Needed

In addition to the material for Checkpoint 1 you will need material from the following sections:

1. Variables, Data Types and Operators
2. Basic Input and Output
3. The Math Class

What Next?

Before attempting the next checkpoint you should read the section on reading and writing files: “File Input and Output”.

13 File Output and Input

13.1 Introduction

To make programs really useful we have to be able to input and output data in large machine-readable amounts, in particular we have to be able to read and write to files. Since Java has extensive network and *Web* support, the built in input and output system can be very complex, but for simple applications this can be simplified to a few recipes.

13.2 The Basics

There are two types of input and output schemes in Java these being (a) text based and (b) binary data based⁵. In this section we will only consider the text based files where output files typically hold human readable output, and input files are created by the editor.

The Java file model contains three parts, these being

1. The file on disc which is characterised by its name and other attributes such as read/write permission, creation date etc.
2. The high-level Reader/Writer classes that implement simple to use methods to read/write Strings and Lines.
3. The Stream classes that connect the high level Reader/Writer classes to the actual files.

In this section we will give a recipe for connecting these three together, so that you will only have to deal with the high-level Reader/Writer classes.

13.3 Output with the `PrintWriter` class

The `PrintWriter` class implements simple character based file output as shown in the example below.

```
// Import the java.io package to access the file classes and methods
import java.io.*;

public class PrinterTest {

    // main must "throw" an IOException as we are reading/writing
    // files
    public static void main (String args[]) throws IOException {

        // Set a file name. If this file does not exist it is created;
        // if it already exists then it is overwritten
        String fileName = "mydata.data";

        // Set up the PrintWriter to point to the file
        PrintWriter output = new PrintWriter(new FileWriter(fileName));

        // Output some data to the file
        output.printf("Here are some numbers\n");
        output.printf("Double = %.5g; Integer = %d\n", 3.14, 1852);

        // Close the output file. If you do not do this you may not see
        // any data in the file.
        output.close();

        // Exit gracefully
        System.exit(0);
    }
}
```

⁵There also the distinction between sequential and random access which will not be discussed here.

```
}  
}
```

So once the `PrintWriter` is created and attached to the output file the actual writing of the data is very simple, you basically send `Strings` that are appended into the current position in the file.

For a fuller example, see the `CosPrinter.java` example which outputs to file the x,y coordinate pairs of a $\cos()$ graph with input parameters via the terminal.

13.4 Input with the `BufferedReader` class

Reading in data from a file is slightly more complex, the simplest interface being offered by `BufferedReader`⁶ which is again best explained by an example that reads a line from the file and outputs it to the screen.

```
// Import the IO package  
import java.io.*;  
  
public class ReaderTest {  
  
    // Remember to throw IOException  
    public static void main (String args[]) throws IOException {  
  
        // A Console to write the data to  
        Console myConsole = System.console();  
  
        // Input file name. If the file does not exist then the program  
        // will exit with an IOException  
        String fileName = "mydata.data";  
  
        // Attach the BufferedReader to the file  
        BufferedReader input = new BufferedReader(new FileReader(fileName));  
  
        // Create a String to hold the data being read  
        String line;  
  
        // Read a line from the file  
        line = input.readLine();  
  
        // Print the line back to the console  
        myConsole.printf("%s\n", line);  
  
        // Close the file  
        input.close();  
  
        // Exit gracefully  
        System.exit(0);  
    }  
}
```

So again, as with `PrintWriter`, if the syntax for opening the `BufferedReader` is used as a recipe, the reading of lines from a input file is very simple. The real difficulty is in interpreting the input `String`, for example if you want to read ints or doubles into your program. Here the best strategy is to read the input file “line-at-a-time” and then break up the input line extracting ints or doubles inside your program. See the next section for how to do this.

⁶You would logically think that there should be a `PrintReader` class, but there is not one.

13.5 Breaking up input lines with Scanner

Most scientific programmers want to read numerical data into their programs, for example to read in columns of numbers into arrays which the program then processes. This is always a tricky problem especially if the exact format of the input lines is not known exactly so you have to “hunt” for the start of the numbers which may be surrounded with stray *white space* characters, such as *space*, *tab* etc. The problem is called *Tokenizing* a string and there is a very useful class within the Java `util` package that does almost all the work for you. The following example does exactly what we want,

```
// Import the Scanner class
import java.util.Scanner;

< ---      usual start of program      ----- >

// Set up a String
String line = "Hello World and Welcome";

// Create a new Scanner based on the line above
Scanner tokens = new Scanner(line);

// Loop over the tokens in the line printing them out
while(tokens.hasNext()) {
    String s = tokens.next();
    System.out.println("Token is : " + s);
}
```

This piece of code will print out the *four* tokens, being “Hello”, “World”, “and”, and “Welcome”.

Run this program with various input lines and see what it does. By default `Scanner` assumes that there is a space between each token, this can however be altered to a wide range of characters or logical combination of patterns, see Java documentation for details.

The `Scanner` can also be attached directly to a `BufferedReader` rather than having to read each line in turn. This is best illustrated by an example. If we have a file called `sometext.txt` that contains the following text:

```
This is some
text in a
file
```

we could read and tokenise it within Java with the following code fragment:

```
String myFile = "sometext.txt";
BufferedReader input = new BufferedReader(new FileReader(myFile));

Scanner tokens = new Scanner(input);

String s1 = tokens.next();
String s2 = tokens.next();
String s3 = tokens.next();
String s4 = tokens.next();
String s5 = tokens.next();
String s6 = tokens.next();
String s7 = tokens.next();
```

This may look overly complex, but illustrates one of the main concepts in object oriented programming. That is the use of pre-defined, and hopefully, well tested objects to do the “work”. You do not need to know what is inside `Scanner` you just need to know how to construct it and access the tokens it generates. The more advanced your programming gets the more your programs consist of objects and methods to access them.

13.5.1 Reading ints and doubles

We have seen above how to break a long line up into tokens, but when you read in data what you will almost always want to do is to set int or double values. Scanner provides methods to return token as int or double using the `nextInt()` and `nextDouble()` methods⁷, so if a String called `line` contains two doubles, we can read them with,

```
Scanner scan = new Scanner(line);
double x = scan.nextDouble();
double y = scan.nextDouble();
```

There are also very useful boolean *companion* methods which allow you to *look ahead*, to see if the next token can be read as specified. For int and double these are,

```
boolean hasNextInt() and boolean hasNextDouble()
```

which will return true if the next token is a valid int or double respectively.

So putting this all together leads to the more complex example of `SumDataFromFile` (see examples below) which uses the console to ask for an input file containing two columns of numbers. The file is read in line at a time, tokenized and parsed into doubles and then the columns summed. This example also ignores blank lines and lines that start with the # character which, by convention, is used to denote comment lines in a data file. You can experiment with either data produced by the `CosPrinter` program or data typed into using the emacs editor.

Note: This program will fail in a most un-graceful way if the data format is wrong, a data value is missing or the file contains stray, unexpected, characters. Catching and dealing with such errors is difficult and well beyond a simple programming course. However, using the above strategy of reading lines, then analysing them is the correct start. For example, it would be easy to put a check that we get two tokens from each line, and if not, print out a useful message saying on which line of the input data file the error occurred. Testing and checking input is vital to getting a program to work correctly and robustly.

13.6 Summary and Additional Features

Java has possibly the most complete and thus complex input/output system of any computer language allowing almost unlimited tailoring and optimisation. Here we have seen the very basics of file input/output and how to read, tokenize and parse simple int and double inputs. For many simple programs this is really all you will need. The two useful advanced areas worth looking at are:

1. Binary data input/output is where we read and write direct binary representations of numbers, strings, etc. This is *much* more efficient than character input/output and is used where there is large amounts of data, for example graphics files, images, sound files, movies, etc. In most cases there are high-level classes to handle various file types either as part of Java or add-on packages, the most useful being `java.imageio` which handles images in jpeg, png or bmp formats.
2. Pipe input/output is where the reads/writes from/to the writes/reads of other programs. This allows data to be transferred between programs, one program to control another or data to be sent direct to system utilities; for example being able to print data directly to system printer by “piping” your output directly to `lpr`, the standard system print command.

In addition there is a list of less common input/output areas, including “random access files”, mainly used in data-base system, binary reading and writing complex objects where efficiency is essential and network files access where, for example, accessing a WEB page on a remote machine.

Examples

Source code for the following on-line examples are available,

- Write $x, \cos(x)$ pairs out to a specified file with one pair on each line `CosPrinter.java`
- Program to read in data pairs from a file and sum the columns `SumDataFromFile.java` and some test data [Here](#).

⁷There are also methods to read `Float`, `Byte` etc.

What Next?

You are now ready to attempt Checkpoint 3.

14 Checkpoint 3

Aim of Checkpoint

In this checkpoint you will write **two** short Java programs. Both programs are based around input/output (IO) to files rather than the screen (as was used in the previous checkpoint).

This checkpoint is worth **15%** of the course mark.

Submission Dates

Final submission date for this checkpoint is: **5.00 pm, Thursday 16 October.**

Computing Task A

Modify your Java program from Checkpoint 2B (calculate the properties of a sphere) so that instead of writing the results to the console it:

1. Prompts you for the name of an output file.
2. Writes the results to this file.

Computing Task B

Write a Java program that reads two Cartesian points from a file (whose name is entered in the terminal) which looks like:

```
1.5 5.6  
10.0 13.4
```

Your program should then compute the gradient and the intercept of the straight line connecting these two points and display the equation of the line in a nice format in the terminal window.

(**Note:** The read-in data must be parsed into a `double` value before you can perform any calculations.)

End of Checkpoint

When you have completed **and** tested your programs, call a demonstrator and show them the code, compile it and demonstrate your working program.

This is the end of **checkpoint 3**. Ensure that the demonstrator checks off your name.

Material Needed

In addition to the material for Checkpoint 2 you will need material from the following document:

1. File Output and Input

What Next?

Read through the next section, “Conditional Statements”, before attempting the next checkpoint.

15 Conditional Statements

Read and study this section with care. It is fundamental to programming and contains new ideas and some complex syntax.

15.1 Introduction

Up to now we have been dealing with programs that read numbers, do fixed calculations in a pre-specified order and output results. One of the main powers of computing is conditional control over which statements (parts of the program) are executed, in which order, and how many times. This is generally called *Flow Control* and will be considered in two sections, the first dealing with *Conditional Statements* and the second dealing with *Loops*, which you will deal with *after* the next checkpoint.

15.2 The boolean Data Type

Fundamental to flow control is the boolean data type which can take on the two possible values:

true or false

This can be set by the “comparator” operators,

>	Greater Than
<	Less Than
==	Is equal to (Note: double == sign)
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to

so for example we can write;

```
boolean ask;
int iValue = <some expression>;

ask = iValue < 5;
```

which will set ask to true or false depending on the value of iValue.

These comparator operators can be used to compare any basic data types, ints or doubles in this course.

15.3 Conditional Statements

In Java conditional execution is mainly accomplished by the `if` statement which in its simplest form is:

```
if (boolean) {
    <-- First line of optional code block -->;
    <-- Second line of optional code block -->;
    <-- .... -->;
    <-- nth line of optional code block -->;
}
```

When the “boolean” is true then the *optional code block* is executed, else it is skipped over.

Note the following points about the syntax,

1. There is **no** “;” after the `if ()` statement. *This is a very common source of programming bugs!*
2. The optional code block is enclosed in `{ }` brackets that *must* match. The emacs editor will help here.
3. The `<-- text -->` line means “replace with valid Java statement”.
4. Each line of optional code ends with a “;”.

so for a simple example we have:

```
double xValue;
<-- code to set the value of xValue -->
if (xValue > 5.0) {
    System.out.println("xValue is greater than 5.0'");
}
```

which will print out the message if `xValue > 5`.

A few points to note are:

1. The “indentation” is *not* part of the program, but laying the code out with “indented” `if()` will make it much easier to read. The emacs editor will do most of this for you if you insert a `<TAB>` before each line.
2. The “equal to” (`=`) operator and the “not equal to” (`!=`) operators should not be used with double variables since they compare every *bit* so are highly dependent on how the number is stored and are very sensitive to rounding errors. (See more on this later.)

More on Logical Statements

To form more complex comparison statements the comparator operators can be combined with the three *logical operators*

<code> </code>	OR
<code>&&</code>	AND
<code>!</code>	NOT

Note: The logical AND and OR operators are **double**⁸ characters.

These operators are evaluated after the comparison operators, but it is good programming practice to put in brackets to make the order of evaluation clear (to you). For example a condition of `xValue < 5 OR xValue > 10` can be written as:

```
if((xValue < 5) || (xValue > 10)) {
    <-- First conditional statement -->;
    <-- Second conditional statement -->;
    .....
}
```

These operators can be combined to form very complex statements.

15.4 Double Conditionals

The extended syntax of the `if()` statement is the very useful `if () {} else {}` construct:

```
if (boolean) {
    <-- First line of optional true code block -->;
    <-- Second line of optional true code block -->;
    <-- ....-->;
    <-- nth line of optional true code block -->;
} else {
    <-- First line of optional false code block -->;
    <-- Second line of optional false code block -->;
    <-- .... -->;
    <-- nth line of optional false code block -->;
}
```

If the boolean value is true then the first code block is executed; *else* the second code block is executed.

The use of the `if() {} else {}` gives good “block” structured code that is easy to read, and thus is *more* likely to be correct.

⁸There *are* single character operators `|` and `&` which are the “bitwise operators” will not be used in this course.

15.5 Multiple Conditional

The full syntax of the `if()` includes the `else if()` giving the rather complex structure of:

```
if (boolean_1) {
    <-- optional code if boolean_1 is "true" -->;
} else if (boolean_2) {
    <-- optional code if boolean_2 is "true" -->;
} else if ...
.
.
} else if (boolean_n) {
    <-- optional code if boolean_n is "true" -->;
} else {
    <-- optional code if all booleans are "false" -->
}
```

which allows a whole “chain” of logical statements to be “tried-out” with the correct code executed. The logic of such `else if` “chains” is very difficult to get right and even more difficult to Debug. If you do need to use this structure then you should “draw” the structure out on paper first before you try and code it.

15.6 The `System.exit()` Method

The `System.exit()` method basically “stops” execution of the program exactly as if the program had completed. Up to now you have been using this at the end of your program but it can also be used to conditionally exit the program.

The syntax is simple being

```
System.exit(int status);
```

where `status` is an integer value that is returned to the operating system.

This is useful for complex programs than interact with the actual system.

The typical use of `exit()` is to stop your program if an error has occurred, for example:

```
double xValue, yValue;
<-- code to calculate value of xValue -->;
if ( xValue < 0){
    System.out.println("Value of xValue < 0. Fatal Error");
    // An exit value of 1 usually indicates an error to the system
    System.exit(1);
}
else {
    yValue = Math.sqrt(xValue);
}
```

which will stop the program (with a sensible message) if `xValue` is negative *before* it tries to take the square root of it.

15.7 The `switch` Construct

`switch` is a complex and very useful dispatch construct that can be used to replace complex `if else()` chains. You are *strongly* advised to learn about this, see textbooks, but *after* you have mastered the `if()` structure.

Examples

The following on-line source examples are available:

- Simple square root calculation trapping negative numbers `SquareRoot.java`
- Determining if a double is $+/-/0.0$ with a `if`, `else if`, `else` chain in `NumberSign.java`
- More complex mark processor program with multiple conditionals `PassFail.java`

What Next?

You have now completed sufficient Java to attempt *Checkpoint 4*.

16 Checkpoint 4

Aim of Checkpoint

This checkpoint consists of a more complex program to calculate the roots of a quadratic equation. This program demonstrates the use of conditional statements, mainly the `if(){}else{}` construct, to deal with the various input conditions.

This checkpoint is worth **25%** of the course mark.

A quadratic equation of the form

$$ax^2 + bx + c = 0$$

has roots given by the well known formula of

$$x_0 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

when $a \neq 0$. There are three possible conditions depending on the value of the formula under the $\sqrt{}$. In particular, if $b^2 - 4ac$

- > 0 Two real roots.
- $= 0$ A single root.
- < 0 Two complex roots.

Obviously when $a = 0$ we have a linear equation with a single root and when $a = 0$ and $b = 0$ we have a constant expression with a trivial solution.

Submission Dates

Final submission date for this checkpoint is: **5.00 pm, Thursday 16 October.**

Checkpoint Task

Write a Java program that:

1. reads the values for the three coefficients a , b and c as doubles from the terminal;
2. calculates and print the roots of the quadratic to the terminal in a nice format.

Test your program with the following values of a , b and c .

a	b	c
1	-6	5
2	8	8
1	2	5
0	4	8
0	0	6

Make sure you get what you expect, including dealing correctly with complex roots.

(Hint: The logic of this program is more difficult than you first think. Write out all the possible conditions that can occur on paper before you try and write your program.)

End of Checkpoint

When you have completed **and** tested your program, call a demonstrator and show them the code, compile it and demonstrate your working program with at least the first three items of the set of test data. (The final two are a bit trickier).

This is the end of **checkpoint 4**. Ensure that the demonstrator checks off your name.

Material Needed

In addition to the material for Checkpoint 3 you will need material from the “Conditional Statements” section.

What Next?

Read through the next three sections (“Loops”, “Arrays and Strings” and “Plotting Graphs Using the `matplotlib` Package”) before attempting the next checkpoint.

17 Loops

Read and study this section carefully. You may want to write yourself some simple test programs to check you understand these new constructs.

17.1 Introduction

The simple *Conditional Statements* used in the previous checkpoint allow optional pieces of code to be executed. This is the start of a program but real tasks usually require a piece of code to be executed a “number of times” in a *loop*. *Loops* are fundamental to scientific programming and much of scientific program design is “how to write your task in terms of loops”.

Java has *three* looping methods, these being the while loop, the do-while loop and the for loop. This section covers these three types.

17.2 The while Loop

The simplest loop in Java is the while loop which has the syntax:

```
while (boolean) {
    <-- First line of loop code block -->;
    <-- Second line of loop code block -->;
    <-- .... -->;
    <-- nth line of loop code block -->;
}
```

The boolean is evaluated and if true the code in the {} braces is repeatedly executed with the boolean checked *before* each execution. A simple example is:

```
int iValue = 1;
while( iValue < 10000 ) {
    System.out.println("Value of x is : " + iValue);
    iValue = iValue*(iValue + 1);
}
```

which will print out the series “1,2,6,42,1806,...” up to 10000.

A few points to note about this type of loop:

1. The boolean is almost always an expression that is evaluated to give a boolean.
2. The loop code block *must* change the variable(s) used to form the boolean, or the loop will run forever.
3. If the boolean is initially false the body of the loop will *never* be executed.
4. There is **no** “;” after the while() statement. This is a very common mistake.

This type of loop is used extensively in iterative numerical solutions, for example to refine a value “until” it reaches a certain accuracy.

17.3 The do/while Loop

The do-while loop is very similar to the while loop *except* that the boolean is checked at the *end* of the loop. The syntax of the loop is:

```
do {
    <-- First line of loop code block -->;
    <-- Second line of loop code block -->;
    <-- .... -->;
}
```

```

    <-- nth line of loop code block -->;
}
while( boolean );

```

Again the code block within the `{}` braces is executed repeatedly until the boolean becomes false.

1. The contents of the loop are *always* executed *before* the boolean is tested.
2. As with the while loop, the body of the loop must contain code to update the variables used to evaluate the boolean or the loop may run forever.
3. The placement of “`;`” are critical. There is **no** “`;`” after `do` but there is a “`;`” after the `while()`.

This type of loop is generally less useful than the while loop, it being mainly used in iterative numerical calculations. It also tends to make your programs more difficult to understand and hence to Debug.

17.4 The for Loop

The for loop has two different forms in Java but we will concentrate on the form that is most commonly used in scientific programming.

The form we will use most often *iterates* over a known range of integer values with a set step size. Initially it looks intimidating, but once you understand the syntax with a few examples it will be the main loop you will use in your programs. The syntax is:

```

for ( <-- start statement -->; boolean ; <-- increment --> ) {
    <-- First line of loop code block -->;
    <-- Second line of loop code block -->;
    <-- .... -->;
    <-- nth line of loop code block -->;
}

```

The operation is then

1. The *start statement* is executed first. This is usually used to setup the initial condition.
2. If the boolean evaluates to “true” the body of the loop is executed.
3. The *increment statement* is executed at the *end* of the loop.
4. If the boolean is still evaluates to “true” the body of the loop is executed again.
5. This is repeated until the boolean evaluates to “false”.

This all sounds complicated, but if the for loop is used with a integer variable that *counts* the number of times the loop code block it is executed it becomes much simpler, for example:

```

for (int iValue = 0; iValue < 10; iValue++) {
    System.out.println("The square of " + iValue + " is " + iValue*iValue);
}

```

prints out the numbers from $0 \rightarrow 9$ and their squares.

Note two additional points here:

1. The variable `iValue` is declared in the *start statement*, and so “inside the loop”. You therefore **cannot** access `iValue` outside the loop. It *is* usually what you want, if not you must declare `iValue` outside the loop.
2. The *increment statement* used the `++` operator which means “add 1 to”.

You should work through this loop by hand using the description of the `for` loop and make sure you understand when each statement and the logical test are executed.

This is by far the mostly widely used loop in scientific programming and is ideal when the number of times a loop must be executed is *known* before the loop starts. This loop will be the main construct of your program for *Checkpoint 5*, and for accessing *arrays* (see the next section).

Note: the `for` loop can be used with very complex start, logical and increment conditions. This however usually leads to difficult to understand code and should be avoided. If a loop needs complicated control conditions the simple `while` or `do/while` loop should be used. You are far more likely to get it right!

Aside: some books use the `for` loop to implement an infinite loop using the very odd syntax of `for(; ;)` which I find very confusing, and suggest should be avoided.

17.4.1 The for-each Loop

The alternative form of the `for` loop is often known as the `for-each` loop. This loop allows us to simply loop over all of the items in an array (or, more generally, a collection of objects).

17.5 Nesting of Loops

Any of the above loops can be *nested* one inside the other, to an arbitrary depth.

In scientific computing it is most common to nest `for` loops, especially in two-dimensional problems. For example, the following piece of code calculates and prints out the value of $\cos(x) \sin(y)$ on a 20×20 point grid for x/y in the range $\pm 2\pi$,

```
double x, y, value;

Console myConsole = System.console();

// Outer loop (over i)
for(int i = -10; i < 10; i++) {

    y = 2.0 * Math.PI * (double)i/10.0;

    // Inner loop (over j)
    for(int j = -10; j < 10; j++) {

        x = 2.0 * Math.PI * (double)j/10.0;

        value = Math.cos(x)*Math.sin(y);

        myConsole.printf("%g,%g,%g\n",x,y,value);
    }
}
```

here, for *each* iteration of the loop with respect to the i variable, the j variable loop is executed $20\times$. Note: this piece of code will produce 400 lines of output on the terminal, not very useful, better to plot a graph, see the section on “Plotting Graphs Using the `ptplot` Package”.

When nesting loops it is *essential* to lay out the code in a well structured way with clear indentation. This makes the loop structure much easier to follow and to debug if there is an error. Again the `emacs` editor will assist you here. If you start each line with a `<TAB>` then the indentation will automatically follow the structure of the loops.

When using nested loops, usually with additional `if-else` blocks the logic of the program can get very complex. Under these conditions it is *essential* that you write out the structure of the program *before* you start trying to type the program in. This way it is much easier to ensure that the code you write actually performs the task you intend.

17.6 The break statement

The `break;` statement allows a second route “out” of a loop. For example, if we have `while` loop, and inside this loop there is an `if()` with a *second logical statement*:

```
while (boolean) {  
    <-- First line of loop code block-->;  
    <-- Second line of loop code block-->;  
    if (<-- second logical statement -->) {  
        break;  
    }  
    <-- .... -->;  
    <-- nth line of loop code block-->;  
}
```

then if the `break;` statement is executed the loop is exited *without* further execution of code and the program continues by executing the statements *beyond* the loop.

Note: If the loop is nested, then only the loop in which the `break;` statement occurs is exited.

Using this scheme to exit a loop is *not* good practice, but is very useful for trapping errors or finishing a loop “early” when some type of convergence criteria is reached, or the end of a file of data occurs before it is expected.

The `break;` statement works in all three types of loops. It is also central to the `switch` construct which was noted in the section on Conditional Statements.

Examples

The following on-line source examples are available:

- Number guessing game with `break` of infinite loop `IfGuess.java`
- Print out series of numbers and their squares with a `for` loop `SquaresLoop.java`
- See examples in the next two sections which all use loops in various forms.

What next?

You should read the next two sections (“Arrays and Strings” and “Plotting Graphs Using the `ptplot` Package”) before attempting Checkpoint 5.

18 Arrays and Strings

Read through this section carefully, it contains a significant number of new concepts and difficult syntax. You will almost certainly have to refer back to this section many times while undertaking the final checkpoint(s).

It is also essential that you understand loops before attempting this section.

18.1 Introduction

Up to now we have been declaring and using *single* variables mostly of type `int` or `double`. This has restricted you to problems that involve little data. This limitation is addressed by the use of *arrays* which are fundamental to useful scientific programming.

New programmers often find *arrays* a difficult concept but since they are the basis of all numerical programming, it is essential that you work through this section very carefully and try the given examples.

18.2 One Dimensional Arrays

The one-dimensional array is an object that contains a set of variables with the same *base name* but with a number of elements, similar to a mathematical *vector*⁹.

To declare and allocate an `int` array of length 10 we use,

```
int iValues[] = new int[10];
```

which is actually a two stage process that:

1. Declares an object, `iValues`, that is an `int` array.
2. Allocates storage for 10 `int` in this object, and set their initial value to 0.

The **two** stage aspect of this process may appear irrelevant at the moment, but will become clear later on.

The 10 `int` variables have the *names*:

```
iValues[0], iValues[1], iValues[2], ..., iValues[8], iValues[9]
```

Note: The index starts at “0”, so the last element is “9”!!!

In addition the Object `iValues` has accessible properties, this most important one at this stage being its length, which is accessed by `iValues.length`.

In a program we can either treat the elements of `iValues` as 10 separate integers, or more usefully we can access them in a loop, for example to set all 10 elements to 100 we can use:

```
int iValues[] = new int[10];
for(int i = 0; i < iValues.length; i++){
    iValues[i] = 100;
}
```

where the indexing variable `i` *must* be an `int`.

Note the use of `iValues.length` in the termination condition of the for loop, this is good programming practice as it removes the use of arbitrary numbers in the code.

Arrays of other data types are identical, for example `double` arrays are declared as:

```
double xValues[] = new double[10];
```

which will declare an object which is a `double` array, allocate space for 10 elements, and set their initial values to 0.0.

Again the individual elements of the array can be accessed via a `int` variable index exactly like the `int` array discussed above.

⁹The numbering of elements is different.

Unlike many other computing languages in Java you can declare and/or allocate objects at any point in your program and more importantly for arrays, the size can be an `int` variable. So for example you can read in the length of the array from the terminal or a file and use this value to allocate an array of the correct length, a typical code fragment would be,

```
// Open the file
BufferedReader inFile = new BufferedReader(new FileReader(fileName));

// Read the array size
String line = inFile.readLine();
Scanner tokens = new Scanner(line);
int lengthOfArray = tokens.nextInt();

// Declare and allocate the array
int iValues[] = new int[lengthOfArray];

<- Rest of code using int array iValues of length iValue.length ->
```

which reads the length of the array, then declares and allocates it.

The obvious question is “what happens the second time round the loop” when you ask for a different size array. Again the power of Java comes to your aid and the “old array” will automatically be deallocated¹⁰ and the new one created of the new length. This feature of Java is wonderful for programmers but is also one of the main reasons why Java code is always much slower than other programming languages.

18.3 Initialisation

In addition to the allocation scheme detailed above, arrays can be initialised at definition, for example `int` and `double` arrays

```
int iValues[] = {1,9,5,3,4};
double xValues[] = {12.78,23,9.25262,1.2e6,9.45e-3};
```

declares and allocates objects with **5** elements that will have the values of their elements set to the given list.

18.4 Addressing Arrays

The processing of arrays is almost always done with some type of loop, the most common in scientific computing being the `for` loop we have seen earlier. For example if we want to fill a `x[100]` and `y[100]` arrays with the values of $y = \sin(x)$ for x in the range $0 \rightarrow 5\pi$ we would use:

```
double x[] = new double[100];
double y[] = new double[100];

for(int i = 0; i < x.length; i++) {
    x[i] = 5.0*Math.PI*(double)i/(double)x.length;
    y[i] = Math.sin(x[i]);
}
```

which goes round the `for` loop $\times 100$, calculates the value for `x[i]` then uses its value to calculate `y[i]`.

This will be the most common loop and array construct in your programs, and in particular in the next *checkpoint*.

Note: the element of an array can only be accessed via an `int`.

¹⁰This is in stark contrast to C and “C++” where the programmer must deal with all the allocation and deallocation themselves and the complications and potential for error this involves!

18.5 Multi-Dimension Arrays

In many scientific applications we will want to use multi-dimensional arrays, for example a matrix is best represented by a two-dimensional array.

A two dimensional double array of size 8×10 elements is defined by

```
double matrix[][] = new double[8][10];
```

which is actually **8** one-dimensional arrays each of length **10**.

This would be accessed by *two* indexing variables, typically in a nested for loop, for example to set the whole array matrix to 100 we would use:

```
for(int j = 0; j < matrix.length; j++) {
    for(int i = 0; i < matrix[j].length; i++) {
        matrix[j][i] = 100;
    }
}
```

Note that `matrix.length` will return the **first** dimension, (8 in this case), while `matrix[j].length` gives the **second** dimension (10 in this case).

I have avoided denoting the indices of the array with the misleading terms *row* and *columns* since there is an unfortunate, and irreconcilable difference, between mathematics and computing regarding the meaning of these terms, so when using two-dimensional arrays to perform matrix operations, great care must be used to get the indices the *right way round*; see the example below.

As with one-dimensional arrays the allocation size can be int variable set as run-time. It is also “possible” to allocate two-dimensional array which is an array of one-dimensional arrays with *different* lengths. This is a recipe for vast confusion and should be avoided!

Higher dimensional arrays are obvious, for example a three-dimensional double array can be declared and set to a constant with:

```
double tensor[][][] = new double[3][5][6];

for( int k = 0 ; k < tensor.length ; k++) {
    for( int j = 0 ; j < tensor[k].length ; j++) {
        for( int i = 0 ; i < tensor[k][j].length ; i++) {
            tensor[k][j][i] = 100;
        }
    }
}
```

If you have understood the section above, then you will guess that what you have *really* created is **3** two-dimensional arrays, each of which consists of **5** one-dimensional arrays, each of which has **6** elements. If not do not worry about it at this stage, you will come back and visit this problem in next year’s course!

You can continue this “dimension expansion” to any order you like, but arrays of dimension greater than three are unlikely to be very useful!

18.6 Warning

When you declare an array to have “100” elements it is up to **you** to make sure your program, or any functions your program calls, does not try to access elements outwith the range $0 \rightarrow 99$. Java has strict array bounds checking, and if you do try and access elements outside the array your program will crash with an *exception* (or error) of the type:

```
java.lang.ArrayIndexOutOfBoundsException
```

and a line number where this exception occurred.

18.7 Strings

Strings are the main object in Java used for handling non-numeric data. In scientific (numeric) applications this usually means for formatted output, filename, input parameters and other relatively simple tasks. At this stage of your programming career you need to know relatively little about the details of Strings, other than how to use them.

We have been using Strings right from the first program mainly for data output and passing title and other fixed non-numeric data to objects. This is all you actually need at this stage, but the String is a nice example of an object and how to access some of the methods that act on strings will help you with your future programming.

18.8 Declaration and Initialisation

As we have seen we can declare and initialise a String with,

```
String outputString = "Hello World !";
```

which declares String called outputString and sets its value to Hello World !, being 13 characters long. A few points to note:

1. the " " are **not** part of the String, they mark the beginning and the end.
2. Spaces, punctuation, operators etc. inside the String are just characters and have no special meaning.
3. To put a " inside a String you must use \", so the declaration,

```
String sentence = "She said \"My name is Jane\"."
```

sets the string value to,

```
She said "My name is Jane".
```

which has 27 characters. Note the \ " is a single character. This procedure is often known as “escaping” the special character.

Constant Strings

If we have a String in our program that **must** not be altered or added to, this can be declared as final. Any attempt to accidentally alter this String will result in a program crash.

18.9 Extending Strings

As we have seen in previous sections we have simple “add” Strings together to form a single String, so for example we can write.

```
String outPut = ""; // Declare a null string
String firstWord = "Hello";
String secondWord = "World";

outPut += firstWord + " " + secondWord + " !";
```

A few details points are worth noting here:

1. The declaration of outPut sets its value to a blank String that contains no characters. This may appear odd, but it does allow easy use of the += operator to “add” Strings.
2. The expression forming outPut consists of the “addition” of **five** Strings.

18.9.1 Useful String Methods

The `String` object has large range of methods which “act” on the `String`. This is nice example of how Objects and Methods interact. This list is not complete, but it contains the simplest and most frequently used methods, see the full documentation for a complete list:

1. **Length of String.** `int` value returned by

```
stringName.length()
```

Note the `()` are essential since `length` is a Method.

2. **Contents Equality**¹¹. `boolean` value returned by

```
string1.equals(string2)
```

is true if the contents of `string1` and `string2` are identical. There is also a case insensitive version, being

```
string1.equalsIgnoreCase(string2)
```

which ignores the difference between upper and lower case letters.

There are many more complex comparator Methods, a few of which are, `compareTo()`, `regionMatches()`, `startsWith()` and `endsWith()` which allow ordering of strings, comparisons of sections, and comparison of starts/ends. See full documentation if you need these.

3. **Location of Characters:** Finds the location of a specified char in a `String`, returning its location so

```
stringName.indexOf('a');
```

returns the location of char `'a'` in the `String`, (`-1` if it does not exist). There are also variants to start the search at a specified location in the `String`, and to look for the *last occurrence* of a specified char, this being `lastIndexOf(char)`.

4. **Extracting Substrings:** Returns a `String` that is part of a longer `String`. There are two methods with either one or two `int` arguments, these being:

```
stringName.substring(start)
```

which returns a `String` starting at the `start` character and extending to the *end* of the `String`, and

```
stringName.substring(start , end)
```

which returns a `String` starting at the `start` character and ending at the `end - 1` character. Both generate an error if the initial string is not long enough.

5. **Concatenation** As well as the `+` operator the Method

```
string1.concat(string2)
```

returns a **new** `String` that consists of `string1 + string2`.

6. **Case Conversions:** There are two useful case conversion methods, these being,

```
stringName.toLowerCase() and stringName.toUpperCase()
```

which return a new `String` with all characters in the `String` converted to Lower Case and Upper Case respectively.

7. **printf style format:** is provided by the static `String` method

¹¹Do NOT use `==` it does not do what you expect!.

```
String.format(String template, Object, Object ...);
```

which has *exactly* the same syntax as the `printf()` format scheme covered in the Basic-IO section, so for example we can write

```
double x = Math.PI;           // Set x to PI
int i = 1024;
String oString = String.format("Value of i is %d and PI is %.4g",i,x);
```

which will format `i` and `x` into a `String` using the supplied template so setting `oString` to,

```
"Value of i is 1024 and PI is 3.143"
```

In addition there are a range of more complex `String` methods which are mainly used in non-numeric applications.

Examples

The following on-line source examples are available:

- Fill one-dimensional array with squared numbers and sum, `SquareAdd.java`
- Read in two three element vectors and form the scalar product, `ReadVector.java`
- Multiply two matrices together with for loops (tricky!!!), `MatrixMult.java`
- read in a file name in ‘name.suffix’ format and split into ‘name’ and ‘suffix’, `FileSplit.java`

What Next?

You should read the “Plotting Using `ptplot`” Section before attempting Checkpoint 5.

19 Plotting Graphs Using the `ptplot` Package

This section introduces the `ptplot` package that is required for the next checkpoint.

19.1 Introduction

Much of Scientific Computing involves the analysis and display of data which is best achieved graphically. In Java we have the ability to produce high quality graphical output directly from your application. In this course we shall use an interface called `ptplot`.

The basic use of `ptplot` is best explained by a simple example to plot out a *cosine* graph.

```
import java.lang.Math.*;
import ptolemy.plot.*;      // Import the ptplot package

public class CosPlot {

    public static void main (String args[]) {

        // Set up a Plot object for the plot
        Plot myPlot = new Plot();

        // Set the title of the graph
        myPlot.setTitle("A cosine graph");

        // Set the axis labels
        myPlot.setXLabel("Angle");
        myPlot.setYLabel("cos(Angle)");

        // Add the data to the plot
        for(int i = 0; i < 100; i++){
            double angle = 4.0*Math.PI*(double)i/100.0;

            /*
             * Add a point:
             * The first argument specifies the dataset to add the point to;
             * second argument is the x-value; third argument the y-value and
             * the fourth argument is a boolean that specifies whether or not
             * to connect the points
             */
            myPlot.addPoint(0, angle, Math.cos(angle), true);
        }

        // A frame to display the plot
        PlotFrame myFrame = new PlotFrame("Example", myPlot);

        // Set the frame size
        myFrame.setSize(800, 600);

        // Display the frame and the graph
        myFrame.setVisible(true)

    }
}
```


19.2 Plotting Multiple Datasets

This is done just by specifying a different value for the first argument to the `addPoint` method. The code snippet below could be used in the `CosPlot` example above to plot both the cosine and sine graphs on the same plot.

```
myPlot.addPoint(0, angle, Math.cos(angle), true);  
myPlot.addPoint(1, angle, Math.sin(angle), true);
```

19.3 Setting the Plot Range

By default, `ptplot` will try to show all the data added to the plot but you can set the plotting ranges using the following methods

```
myPlot.setXRange(<lower limit>, <upperlimit>, myPlot.setYRange(<lower limit>, <upperlimit>
```

These set the x and y ranges of `myPlot`. `<lowerlimit>` and `<upperlimit>` are numerical values.

Examples

Source code for the following on-line example is available,

- Code to plot the sine and cosine functions using `ptplot`, `CosSinPlot.java`

What Next?

You are now ready to try *Checkpoint 5* which involves graphs of simple harmonic motion.

20 Checkpoint 5

Aim of Checkpoint

This checkpoint explores the behaviour of a damped simple harmonic oscillator for a range of damping coefficients. It involves writing a Java program to trace out the amplitude against time for a damped simple harmonic oscillator under a range on conditions.

From a computing viewpoint this checkpoint demonstrates the use of loops, arrays, and the `ptplot` package for simple graph plotting.

This checkpoint is worth **25%** of the course mark.

Submission Dates

Final submission date for this checkpoint is: **5.00 pm, Thursday 30 October**

Damped Simple Harmonic Oscillator

The damped simple harmonic oscillator satisfies the second order differential equation,

$$m\ddot{x} + b\dot{x} + kx = 0$$

where m is the mass of the oscillator, b is the coefficient of damping, and k is the harmonic force constant.

Defining new constants $\gamma = b/m$, and $\omega_0^2 = k/m$, we can re-write this as:

$$\ddot{x} + \gamma\dot{x} + \omega_0^2 x = 0$$

where ω_0 is the **natural frequency** of the undamped oscillator.

The solutions to this equation take the following forms:

$$\begin{aligned} x &= \exp(-\gamma t/2) (a \cosh(pt) + b \sinh(pt)) && \text{when } \gamma > 2\omega_0 \text{ with } p^2 = (\gamma^2/4) - \omega_0^2 \\ x &= \exp(-\gamma t/2)(a + bt) && \text{when } \gamma = 2\omega_0 \\ x &= \exp(-\gamma t/2) (a \cos(\omega t) + b \sin(\omega t)) && \text{when } \gamma < 2\omega_0 \text{ with } \omega^2 = \omega_0^2 - (\gamma^2/4) \end{aligned}$$

where these three conditions are known as “over damped”, “critically damped” and “under damped” respectively. (**Note:** Read these equations very carefully and note the locations of the parentheses.)

With the initial conditions that $x = 1$ and $\dot{x} = 0$ at $t = 0$ the above constants, after some manipulation, become,

$$\begin{aligned} a &= 1 & b &= \gamma/2p && \text{when } \gamma > 2\omega_0 \\ a &= 1 & b &= \gamma/2 && \text{when } \gamma = 2\omega_0 \\ a &= 1 & b &= \gamma/2\omega && \text{when } \gamma < 2\omega_0 \end{aligned}$$

Checkpoint Task

Write an interactive Java program to compute and display, via the `ptplot` package, the solution for x against t for t in the range $0 \rightarrow 5\pi/\omega_0$. Your program should:

1. Read in the values of ω_0 , γ and the number of points to plot from the terminal.
2. Calculate and plot the amplitude vs. time using with the `ptplot` package.
3. Re-prompt for new values of ω_0 , γ and number of points.
4. Add the results of this new calculation to the existing graph as a new series.

Test your program using the following inputs. To obtain a smooth plot you need to calculate the above expression for at least 200 evenly spaced values of t . For an under damped condition use:

$$\gamma = 0.5 \quad \omega_0 = 1.0$$

For a critically damped condition use:

$$\gamma = 2.0 \quad \omega_0 = 1.0$$

Finally, for an over damped condition use:

$$\gamma = 4.0 \quad \omega_0 = 1.0$$

(**Note:** The structure of this program is reasonably complex. Think very carefully how you are going to structure it before you start. Also be **very careful** when coding the expression for the amplitude of the oscillation, in particular make sure it is performing the correct arithmetic!.)

End of Checkpoint

When you have completed your program, call a demonstrator and show them the code, compile it and demonstrate the working program for all three damping conditions.

This is the end of **checkpoint 5**. Ensure that the demonstrator checks off your name.

Material Needed

In addition to the material for Checkpoint 4 you will need material from the following sections:

1. “Loops”;
2. “Arrays and Strings”;
3. “Graph Plotting using the `ptplot` Package”.

What Next?

You should now read the next two sections (“Introduction to Methods” and “Introduction to Objects”) before attempting the next Checkpoint.

21 Introduction to Methods

The concepts of methods are essential to good programming and you will make use of them in the next checkpoint.

21.1 Introduction

You have been making extensive use of *Objects* and *Methods* in your programs, for example `Console` objects for input/output; manipulating strings; calculating mathematical operations; and creating graphs with the `ptplot` package. So far all the objects and their associated methods have been supplied and you have just invoked them. The real power of programming is when you create your *own* objects and methods where the methods perform certain tasks on the objects. This short section concentrates on static methods associated with the main program and leaves the creation and manipulation on new objects to the next section of the course and future computing courses¹².

A static method is self contained piece of Java code that takes parameters, does a calculation and returns a value. You have already been using such methods, for example `Math.sin` or `Math.sinh`. Methods are frequently used in scientific and numerical calculations and are the first step towards modularisation of your program and structured programming in general.

21.2 An Example static Method

We want to write a method to evaluate a quadratic of the form

$$ax^2 + bx + c$$

so we need a method that

1. Takes the *values* of 4 doubles as arguments (corresponding to a , b , c and x).
2. Returns the *value* of the equation, as a double.

We can write this as the *static* method `quadratic()` which can then be accessed from the `main()` program. The code for the static method and the main program looks like:

```
/*
 * Define the main class (object) which contains two methods: quadratic
 * and main
 */
public class QuadCalc {

    /* Define our static method: quadratic
     * static - Means the method can be called without needing an object
     *           to be created
     * double - Means that this method returns a double value
     * (double a, ...) - Define four double arguments to the method. These
     *                   variables are local to this method
     *
     * note the lack of the "public" keyword - this method can only be
     * called from within the "QuadCalc" class.
     */
    static double quadratic(double a, double b, double c, double x) {

        // Compute the value of the equation into a local variable
        double value = a*x*x + b*x + c;

        // Exit the method and return the value of the equation
        return value;
    }
}
```

¹²Computer Simulation or Computational Methods in Physics Junior Honours.

```
}

// The main method (MUST be "public static void")
public static void main (String args[]) {

    // Set a variable value
    double x = 10;
    // Variable for the result
    double calculatedValue;

    // Call the method
    calculatedValue = quadratic(2.3, 4.6 , 3.1 , x);

    // Print out the calculated value
    System.out.printf("Value is : %.5g\n", calculatedValue);

}
}
```

The arguments can be any combination of objects, at this stage in your programming they will usually be int, double or Strings.

21.3 Arrays and Methods

The use of arrays and methods together takes a little more care. At its simplest a method can have an array as a parameter *or* can return an array as shown in the example below.

```
public class ArrayTest {

    // Method to declare and fill an array
    static double[] makeAndFillArray (int length) {

        // Create the new array
        double array[] = new double[length];

        // Fill it with values
        for(int i = 0; i < array.length; i++) {
            array[i] = (double)i;
        }

        // Return the array
        return array;
    }

    // Method to sum elements of an array
    static double arraySum (double array[]) {

        // Zero the sum variable
        double sum = 0;

        // Loop over array elements accumulating sum
        for(int i = 0; i < array.length; i++) {
            sum = sum + array[i];
        }

        // Return the sum
        return sum;
    }
}
```

```

    }

    // The main method
    public static void main (String args[]){

        // Define new array using method
        double arrayOfDoubles[] = makeAndFillArray(10);

        // Sum elements of array
        double sum = arraySum(arrayOfDoubles);

        // Print out result
        System.out.printf("Vector sum is : %.5g\n", sum);
    }
}

```

The `makeAndFillArray` method takes an `int` as its parameter and, being declared as `double[]`, returns a double array, in this case filled with 1.0, 2.0, ... etc.

The `arraySum` method takes a `double[]` as its parameter and returns the sum of the elements as a double. Note that `.length` holds the length of the array which can be used inside the method.

The two methods can then be used by the main program exactly as expected.

The conceptual problem comes when an array is passed to a method as a parameter and the method changes the values of the elements. In this case the elements of the array also change in the main program. Lets modify the above method `makeAndFillArray` to a `fillArray` method as detailed below.

```

public class ArrayTest {

    static void fillArray(double array[]) {
        for(int i = 0; i < array.length; i++) {
            array[i] = (double)i;
        }
    }

    static double arraySum(double array[]) {
        double sum = 0;
        for(int i = 0; i < array.length; i++) {
            sum += array[i];
        }
        return sum;
    }

    public static void main(String args[]){

        double arrayOfDoubles[] = new double[10]; // Declare array
        fillArray(arrayOfDoubles);                // Set it value
        double sum = arraySum(arrayOfDoubles);     // Check its value
        System.out.printf("Sum is : %.5g\n", sum);

    }
}

```

The method `fillArray` takes a `double[]` as a parameter, but, unlike a single variable, the elements of the array passed become *common* with the main program so that changes to the elements inside the method **will** change the element values in the main program. Type in the above program, it prints `Sum is : 45.000` exactly as the previous version did.

Use this as a *feature* at this stage of your programming career, it is actually perfectly logical but you need to understand a bit more about the language before we can cover this.

Overloading of Methods

This is now a most convenient time to introduce the idea of *method overload* where two, or more, methods have the same *name* but *different* parameter lists. The reason that this works is because in Java unlike other languages, the method signature is a combination of its *name*, *parameter list* and *return type*. Again this is best illustrated by an example or a better `fillArray` method, begin:

```
static double[] fillArray(double array[]) {
    for(int i = 0; i < array.length; i++) {
        array[i] = (double)i;
    }
    return array;
}

static double[] fillArray(int length) {
    double array[] = new double[length];
    return fillArray(array);
}
```

where we have now two methods

1. takes a `double[]` as argument, fills its elements, and returns the same array.
2. takes an `int`, allocates a `double[]` of the correct length, calls the *first method* to fill its elements, and then returns the `double[]`.

These two methods of the same name can now be used for the *same task*, that of filling an array with numbers as follows

```
double firstArray[] = new double[20];    // Declare and allocate
firstArray = fillArray(firstArray);      // Fill using method 1

double secondArray[] = fillArray(30);    // Declare, allocate/fill
                                         // using method 2
```

This is an extremely powerful and useful technique where methods that do the same task can be given the same *name* so making the final code much easier to read.

If you did not understand this last section, don't worry, you will come back to it later.

21.4 `main()` as a Method

The observant will notice that the

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

that is required at the start of your main program looks just like a method declaration, correct, it is a method executed when the program is run. The declaration is now clearer with:

1. `public` meaning that it can be accessed from outside the *class* that contains it.
2. `static` means it is not associated with actions on an object but is a stand-alone method.
3. `void` means that it does not return anything.
4. `main` the name of the method automatically invoked.
5. `String args[]` the parameter list, which is an array of `Strings` which are passed from the terminal command line when the program is executed. This is an alternative method of inputting data into to program.

The command line options are simply accessible as an array of `Strings`. Considering the simple program below.

```
public class ArgumentTest {
    public static void main (String args[]) {

        for(int i = 0; i < args.length; i++) {
            System.out.println("Argument " + i + " is : " + args[i]);
        }
    }
}
```

If you execute this with:

```
java ArgumentTest Hello World
```

you will get:

```
Argument 0 is : Hello
Argument 1 is : World
```

showing that `args[0]` has value "Hello" and `args[1]` has value "World", which have been passed in from the command line.

This is the standard scheme by which command line options are passed to programs on execution. You will see this again in future courses.

Examples

Source code for the following on-line examples are available,

- Program with static (class) method `quadratic QuadCalc.java`.
- Examples of using arrays with static (class) methods `ArrayMethods.java`.
- Reading arguments from the command line, (simple test) `ArgumentTest.java`.

What Next?

You should now read the “Introduction to Objects” section before you attempt the final checkpoint.

22 Introduction to Objects

This section is required for the final checkpoint (6).

22.1 The Basics

An *object* is a programming building block that is defined to have a set of *properties* and a series of *methods* that modify these properties, return information from the object or instruct the object to perform some task. This somewhat abstract concept is best looked at by taking one of the *objects* we have been working with, the `ptplot` Plot object. This object has

1. **Properties:** for example *colour* of the graph, *title* of the graph, list of points to be plotted etc.
2. **Methods:** to change these properties (for example `setTitle()`); and to add data points (`addPoint()`); and to instruct the object to perform tasks.

The internal workings of the object are hidden from the user who only creates instances of the object (using a *Constructor*), and manipulates it using its own methods. This allows the object to be a self contained and isolated from the programs that uses it. This has the advantage that the object can be used in many different programs, but more important is that the object can be tested, or replaced with an improved version (for example, a more sophisticated one), and then provided that the new object has the same methods then the the main program will work with the new version.

This concept of having methods associated with the object and the internals of the object *hidden* and only accessible via methods is central to the “object-oriented” programming approach. The problem of programming then becomes the design of objects to hold the data in a useful way complete with methods to manipulate and interact with the objects.

22.2 A simple Point object

Let us consider a simple example of a `Point` object which describes a three-dimensional point in space. A *point* can be described by three coordinates, typically its x, y, z location, this defines how we wish to create a `Point`. Look at this first with the following partial definition.

```
/*
 * Begin the definition of a new object (class) called "Point".
 * Note the "public" keyword that allows the class to be seen
 * by external methods.
 */
public class Point {

    /*
     * Define the properties of the object.
     * Three internal variables that specify the x-, y- and
     * z-coordinates.
     * The "private" keyword specifies that the variables
     * are only visible within the class.
     */
    private double xLoc, yLoc, zLoc;

    /*
     * Declare a constructor to be used when creating a new Point
     * object. This constructor takes three variables that define
     * coordinates.
     * Note that the constructor is "public" and has the same name
     * as the class. A constructor never has a return type.
     */
    public Point(double x, double y, double z) {
```

```

        // Set the internal variables to the supplied values
        xLoc = x;
        yLoc = y;
        zLoc = z;
    }

    /*
    * An alternative constructor that takes no arguments and creates
    * a Point object at 0, 0, 0 (the origin).
    * Note: You can have as many different (overloaded) constructors
    * as you need but * they must all have different argument
    * signatures.
    */
    public Point() {
        // Set the internal variables
        xLoc = 0.0;
        yLoc = 0.0;
        zLoc = 0.0;
    }
<more to follow>

```

So far this allows us to create a `Point` object, for example in our main program we can now write:

```

Point location = new Point(2.5 , 4.5 , 7.0);
Point origin = new Point();

```

which will create a `Point` object with coordinates 2.5,4.5,7.0 called `location` and a second `Point` called `origin` at location 0,0,0. However, this object does not yet allow us to do anything.

We now need *instance* methods to work with the object. Let's first consider methods to return the *x,y,z* location which are written as:

Note: An *instance* method is a method that is associated with a particular instance of an object. In this case, these methods require an actual, defined `Point` object to have any meaning. Contrast this with *class* (static) methods from the "Introduction to Methods" section which work even if we have not defined a particular instance of an object.

<definition from above>

```

    /*
    * Define a public instance method "getX" that returns the value of
    * the x-coordinate.
    */
    public double getX() {
        return xLoc;
    }

    /*
    * Define a public instance method "getY" that returns the value of
    * the y-coordinate.
    */
    public double getY() {
        return yLoc;
    }

    /*
    * Define a public instance method "getZ" that returns the value of
    * the z-coordinate.
    */
    public double getZ() {

```

```

        return zLoc;
    }
<more to follow>

```

These methods allow, in our main program to write,

```

double xPosition = location.getX();
double yPosition = location.getY();
double zPosition = location.getZ();

```

where location is a Point as defined above. This allows us to read back the values of a Point (still not very useful!).

Note the syntax. The () are essential since .getX() is a method.

Aside: By convention methods that return the values of internal variables all start with get this is not actually required, but is considered good Java programming practice. Such methods are commonly referred to as “getters”.

Now let's look at an instance method to calculate the distance to the point from the origin which can be written as,

<definition from above>

```

/*
 * Public instance method to get the distance of this point from
 * the origin.
 */
public double fromOrigin(){
    return Math.sqrt(xLoc*xLoc + yLoc*yLoc + zLoc*zLoc);
}

```

<more to follow>

where, since the method is defined within the class *Point*, it has access to the private variables xLoc, yLoc and zLoc.

This method allows us to write (in our main program):

```

double distanceFromOrigin = location.fromOrigin();

```

again note that the final () are essential.

Things become more useful when we are able to pass parameters to the methods, for example we wish to calculate the distance between two Points. This starts to introduce the real useful power of this technique. We want a method that will allow us to write

```

Point first = new Point(1.0,1.0,1.0);
Point second = new Point(2.0,2.0,2.0);
double distanceBetweenPoints = Point.separation(first, second);

```

Notice that this method does not depend on a particular instance of a Point object, it just operates on two point objects. We are therefore going to be writing a *class* (static) method.

Calling static methods is achieved by using the class name (Point in this case) and appending the method name. (You have seen this before when calling the Math class methods.)

The code for such a method is:

<definition from above>

```

/*
 * A class method to calculate the separation of two points
 * Note the "static" keyword that denotes a class method.

```

```

    */
    public static double separation (Point p1, Point p2) {

        // Get the coordinate differences using the "getter" methods.
        double xDelta = p1.getX() - p2.getX();
        double yDelta = p1.getY() - p2.getY();
        double zDelta = p1.getZ() - p2.getZ();

        // Compute the separation and return it
        return Math.sqrt(xDelta*xDelta+yDelta*yDelta +zDelta*zDelta);

    }

```

<more to follow>

Now lets declare our final instance method to *add* a Point to a defined Point and return a new Point. This method has to take a second Point as a parameter and then return a new Point.

The code for this is:

<definition from above>

```

    /*
    * Define a public instance method to add a Point to this one
    * and return a new Point
    */
    public Point add (Point p) {

        // Get the summed coordinates
        double xNew = xLoc + p.getX();
        double yNew = yLoc + p.getY();
        double zNew = zLoc + p.getZ();

        // Return a new Point object
        return new Point(xNew, yNew, zNew);

    }

```

<more to follow>

This now allows us to write (in our main program),

```

    Point first = new Point(1.0,1.0, 1.0);
    Point second = new Point(2.0, 1.0, -1.0);

    Point third = first.add(second);

```

which is a much nicer, and more obvious, piece of code than having many *x,y,z* locations in your main program.

Finally, lets add a *toString* instance method to return a formatted String containing the three coordinates:

<definition from above>

```

    /*
    * Public instance method to return a Point as a String
    */
    public String toString() {
        // Set the String format here
        return String.format("(%g,%g,%g)", xLoc, yLoc, zLoc);
    }

```

Now the `toString` method is one of the standard methods defined for *all* objects and is automatically called when we try to convert any *Object* to a *String*. We have replaced (overloaded) it for our *Point* object, so in our main program we can now write

```
.....
Point third = first.add(second);
System.out.println("Value of third point is : " + third);
```

When the *Point* is *added* to the *String*, it is automatically converted to `toString()` using the method we have just written, and then concatenated, just as you would hope.

This simple example illustrates the basics of creating and using a new object and writing simple methods to manipulate and operate on it. This is only the start of *object-oriented programming*, which will be expanded on in future courses, in particular the optional *Computational Simulation* next term, or *Computational Methods* next year.

22.3 Putting it together

Due to the filename constraints in Java we have to be careful in putting this together. Remember that the filename **must** match the *class* it contains, so this means that the *Point* class must be contained in a file called *Point.java*, while the class containing the main program **must** be in a different file.

In this case we have **two** files *Point.java* that contains,

```
public class Point {

    private double xLoc, yLoc, zLoc;

    public Point(double x, double y, double z) {
        xLoc = x;
        yLoc = y;
        zLoc = z;
    }
}
```

<definition of the methods>

```
}
```

and the main program that uses *Point*, for example *PointTest.java* containing:

```
public class PointTest {
    public static void main(String args[]){

        Point first = new Point(2.0,4.0,6.0);
        Point second = new Point(-1.0,3.0,-5.0);

        System.out.println("Distance between first and second is : " +
                           first.distance(second));

        <---- rest of code ---->
    }
}
```

When you compile *PointTest.java* using `javac` it will automatically look for a file called *Point.java* in your current directory which it will also compile to give a *Point.class* file. Then when you run *PointTest* with `java PointTest` it will automatically pick-up the code for the *Point* object, which is contained in the *Point.class* file.

This is the **most** basic method of using objects, you will learn much more about managing objects and packages in future courses.

22.4 Why bother with objects and classes

The first thing you notice when starting with objects is that you appear to write *more code* and it is *more complex* than if you did it all in the main program. Yes, to start with this is true, but the main program becomes massively simplified and if you define your objects carefully they can be re-used and *extended* for use in other programs. This saves you time and effort in the long run. More importantly, the use of objects forces you to think in a structured way and start breaking the task down into manageable chunks. You can then define, write and (more importantly) test each object (and its associated methods) *before* you use them in your final main program¹³. Most programmers spend in excess of 90% of their time testing and debugging, so creating sensible objects in an object oriented programming language **is** actually a very efficient method of working.

There are articles, books and complete courses on structured programming, and *object oriented programming*, all of which are valid, however you only really *learn* and *understand* the power of this technique by trying it out, seeing what works, making mistakes and finally starting to think in the *correct way*; there is no substitute for sitting at the computer trying things out and making it work!

Examples

Source code for the following on-line examples are available,

- Code for the `Point` object `Point.java`
- Code for the `PointTest` test program `PointTest.java`

What Next?

You are now ready to try the final checkpoint 6.

¹³With large programs the different classes are often written by different programmers in a development team, this scheme allows each programmer to concentrate on their piece of the code allowing them to fit it all together at the end.

23 Checkpoint 6

Aim of Checkpoint

This checkpoint will provides a simple introduction to the object-oriented programming technique. You will create an object that represents a circle, add instance methods for accessing the properties of the circle, and add class methods for general use with circle objects.

This checkpoint is worth **20%** of the course mark.

Submission Dates

Final submission date for this checkpoint is: **5.00 pm, Thursday 30 October**

The Circle Object

A circle can be completely defined, in two-dimensional Cartesian space, by its radius and the coordinates of its centre. These properties provide the basis for defining an object that represents a circle and directly correspond to the *class variables* in a Java class.

A circle also has other well defined properties that we could imagine that it would be useful to include in a circle object. These include, amongst others, the circumference and area of the circle and correspond to *instance methods* in a Java class. That is, the values are associated with a particular circle object rather than circle objects in general.

We can also imagine a number of useful operations that we would like to perform on mutliple circle objects. For example, it is often useful to know if two cricles overlap. This type of operation best corresponds to a *class method* in a Java class.

Checkpoint Task

You should write a class that represents a circle object and includes the following:

1. Private class variables that store the radius and centre coordinates of the object.
2. Constructors to create circle objects with nothing supplied, with just a radius value supplied and with a radius and centre coordinates supplied.
3. Public instance methods that allow the radius and centre coordinates to be set and retrieved (often known as set/get methods).
4. Public instance methods that return the circumference and area of the circle.
5. A public class method that tests if two circle objects overlap or not.

Once you have written your class you should write a program that proves that the object and all of the the methods function as expected.

(**Note:** Although the class itself is realtively straightforward you will produce more code than you might expect and this will be the largest Java program you have written so far.)

End of Checkpoint

When you have completed both your programs call a demonstrator and show them the code, compile it and demonstrate the working program.

This is the end of **checkpoint 6**. Ensure that the demonstrator checks off your name.

Material Needed

In addition to the material for Checkpoint 5 you will need material from the following sections:

1. Introduction to Methods.

2. Introduction to Objects,

What Next?

If you started at Checkpoint 1 then congratulations you have completed the course. If you started at Checkpoint 4 then you should go on and attempt Checkpoint 7.

24 Checkpoint 7

Aim of Checkpoint

This checkpoint is an alternative to checkpoints 1 to 3 for advanced programmers.

We will create a simulation to test if a randomly filled array of a specified size and fill density can *percolate*. In other words, is there a unobstructed path from the top to the bottom of the array? To do this we use Java classes and methods, two-dimensional arrays and random numbers. You are required to code the algorithm described below; test that it works on single array instances; and then extend the program to perform a statistical simulation of the system.

Your final simulation code will allow you to perform experiments to find the *critical density* at which percolation occurs with a probability of 0.5.

This checkpoint is worth **30%** of the course mark.

Submission Dates

Final submission date for this checkpoint is: **5.00 pm, Thursday 30 October**

Percolation

We can imagine randomly filling a square array with filled cells such that the fill density fraction is between 0.0 and 1.0 (with 0.0 being a completely empty array and 1.0 being a completely full array). See Figure 5.

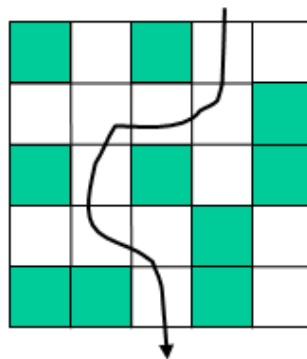


Figure 5: A 5-by-5 array that does allow percolation through the empty cells (white).

What we want to test is: Is there a link between the top and bottom of the array through linked empty cells (without moving diagonally)? If so, we say we can *percolate* through the array.

The algorithm that we are going to use to test for percolation is a simple iterative solver and works in the following way:

1. Randomly fill a square (integer) array of cells to the specified density (1 = empty, 0 = full).
2. Add a border of filled cells around the array so that when we look at neighbouring cells in the fourth step we do not have problems with looking outside of the array boundaries. This will increase the dimension of our array by two. See Figure 6.
3. Loop over all the cells in the array. If they are filled (= 0) leave them unchanged; if they are empty (= 1) assign its value to a unique positive integer.
4. Iteratively loop over all the cells in the array. If the cell is empty (> 0) then check if any of the neighbouring cells contain a larger value; if so, replace the value of the current cell by this larger value. In this way the largest number in a linked cluster of empty cells gradually expands to fill the cluster.
5. Did any values in the empty cells change in the last step? If so, go back and repeat the previous step; if not, test for percolation.

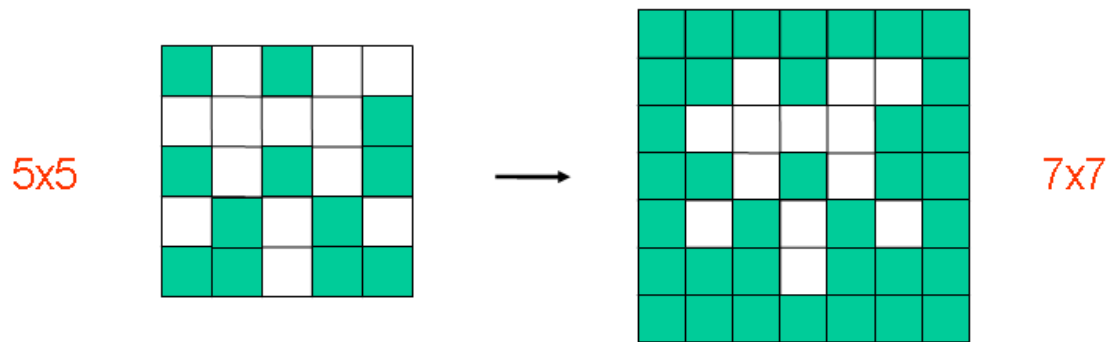


Figure 6: Addition of a filled border to facilitate looking outside the original array bounds.

6. Once the values in the empty cells are no longer changing check for percolation by seeing if the index of an empty cell cluster is present in both the top and bottom row of the array.

This algorithm is illustrated in Figure 7.

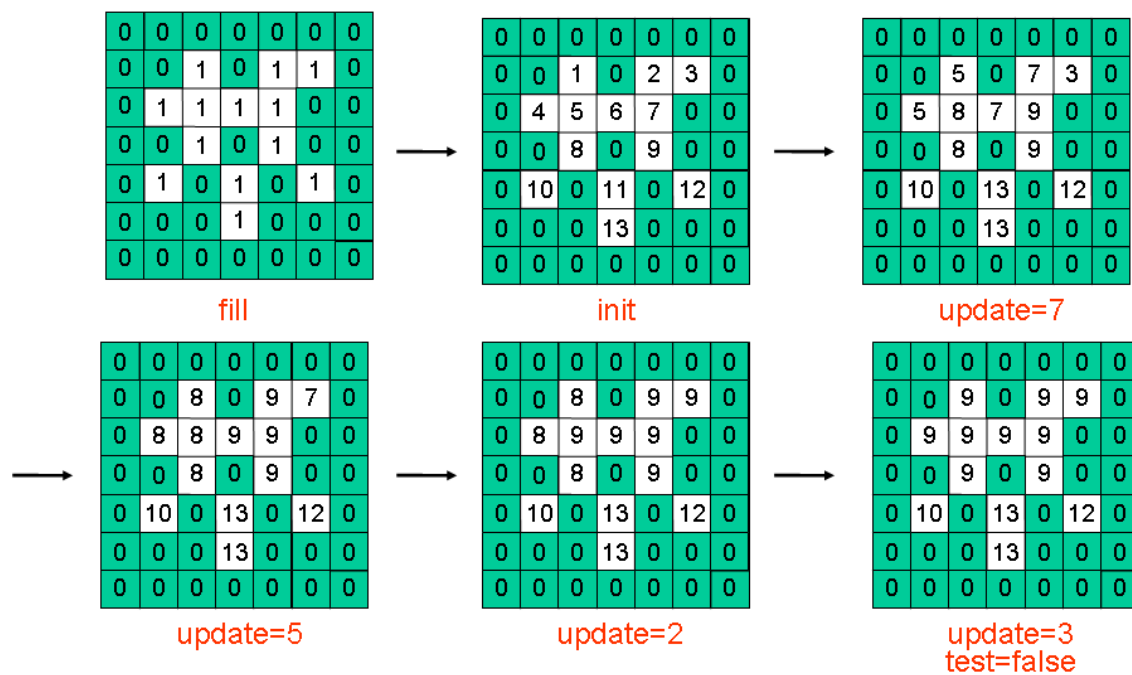


Figure 7: Schematic of the percolation algorithm.

Checkpoint Task

You should write a class that represents the square array object and includes the following:

1. Private class variables that store the size of the array and the array itself.
2. A Constructor to create a new, empty array of the specified size.
3. A public instance method that randomly fills the array to the specified density.
4. A public instance method that initialises the array by assigning a unique positive integer to each of the empty cells.

5. A public instance method that performs an update step on the array by replacing each empty cell value with the largest value of itself and its four neighbours. This method should return the total number of changes made.
6. A public instance method to check for percolation. Should return *true* if percolation is possible or *false* if it is not possible.

Once you have written your class you should write a program that tests it works for a single array of a specified size and density. *i.e.* it:

1. Reads the array size and density from the console.
2. Using the class written above creates, fills and initialises a new array.
3. Updates the array (using the class method) until no more changes are being made.
4. Checks for percolation (using the class method) and prints the result to the console.

Finally, you should write a program to compute the probability of percolation for a specified array size, density and number of simulations. Use this program to determine the critical point for arrays of size 20 and 200. You should compute the probability based on at least 100 simulations at each point (*i.e.*, array size and density).

End of Checkpoint

When you have completed both your programs call a demonstrator and show them the code, compile them and demonstrate the working programs.

This is the end of **checkpoint 7**. Ensure that the demonstrator checks off your name.

A Finding and Fixing Bugs

A.1 Introduction

As you will quickly find the *Bug* is the pain of all programmers existence. This section looks at the most common types of Bugs and some of the strategies for finding and fixing them.

Remember: Computer are inanimate pieces of electronics with no inherent intelligence or malice; they do “exactly” what *you* tell them to do, *wrong though it may be!* A non-working program means *you* have put a mistake into it.

A.2 Types of Bugs

Basically there are three types of BUGS, these being:

A.2.1 Syntax Errors

You have a mistake in the syntax of a statement in the Java code. Such errors usually mean that no class file is produced, or if it is produced it is very unlikely to work correctly. Then you compile your program, with `javac`, you will get error messages and the “line number” where the compiler first noticed the errors. The most common errors are:

1. Forgetting to declare a variable, or miss-spelling the name of the variable either in the declaration or when used in the code. You will get an error message of the form:

```
CheckpointFive.java:34: cannot find symbol
symbol   : variable nsetp
location: class CheckpointFive
    double timeInterval = timeMax / (double)nsetp;
                                   ^
1 error
```

This means at line 34 of file `CheckpointFive.java` the compiler found a variable called `nsetp` which it was not expecting to find. It also shows you the line where the error occurred.

2. Getting the name or calling parameter of a *method* wrong will result

```
CheckpointFive.java:81: cannot find symbol
symbol   : method addPiont(int,double,double,boolean)
location: class ptolemy.plot.Plot
    plot.addPiont(0,time[i], amplitude[i], true);
               ^
1 error
```

Remember correct spelling is **vital**, also if you supply the wrong parameters the compiler will tell you that it cannot resolve symbol since the parameter type list is part of the definition of the method.

3. Forgetting the “;” at the end of a statement. You will typically get as error message of the form:

```
CheckpointFive.java:38: ';' expected
    double amplitude[] = new double[nstep]
                                   ^
1 error
```

with the ^ “showing” you where the compiler *thinks* the ; *should* go. Remember it is the compilers “guess” and it may not be correct!

4. Forgetting to close the “ ” round a string, the “{ }” round a group of statements, or the “/* */” round comments. You will get a variety of odd errors, most of which make sensible suggestions as to what is wrong. Most of these problem are picked up by the colour highlighting in emacs.
5. Forgetting an import file at the top of the program. For example if you try and use Plot object without including `ptolemy.plot.*` you will get:

```
CheckpointFive.java:77: cannot find symbol
symbol   : class Plot
location: class CheckpointFive
    Plot plot = new Plot();
    ^

CheckpointFive.java:77: cannot find symbol
symbol   : class Plot
location: class CheckpointFive
    Plot plot = new Plot();
    ^
```

which means the compiler was not expecting Plot since you forgot to tell it to look in `ptolemy.plot.*`.

6. General syntax errors in statements, for example in arithmetic statements, assignments or loops. Again the compiler will give you the line number where the error occurs.

Fix syntax errors **one at a time**, starting with the *first* error detected. Remember that one *simple* error, for example a missed “}” can result in dozens, sometimes hundreds, of totally spurious, errors message from other parts of the program that are actually perfectly correct!

A.2.2 Runtime Errors

These occur when the program *compile* successfully but when you try and run it, it either *crashes* with an *Exception* or in the worst case does not appear to do anything at all!

There are a vast range of possible *Exceptions*, however at this level of programming the most likely are:

1. `NumberFormatException`, normally combined with

```
at java.lang.FloatingDecimal.readJavaFormatString...
```

means you have tried to read a number from a String that contained something other than a sensible number. When using the Console class, this means that you types *rubbish* into a prompt field that was expecting an int or a double.

2. `ArrayIndexOutOfBoundsException`: Your program has tried to access “off the beginning or end” or an array. Again the line number will tell you where this happened.
3. *Floating Point Errors*: Your program has tried to created a double number greater than 10^{308} . This usually means you have tried to divide by *zero* (or a very small number), *or* you have gone round a loop far more times than you wanted and some number has got “very big”. The double gets set the NaN or Infinity but the program **continues**, producing rubbish.
4. *Nothing Happens*: Your program could be charging aimlessly, and non-productively round a infinite loop without ever getting to its termination condition. To stop the program type Ctrl-C (both keys at once).

The debugging strategy depends on the complexity of the program. For the type of programs you are writing at the moment usually “reading through” the code is enough to spot the mistake. The most common ones are:

- (a) Forgetting to assign a value to a variable, which defaults to zero, then using it assuming it has a non-zero value. *Overflows* and *Infinity loops*.
- (b) Miss-placed “;” in loop, for example

```

int x = 1;
while ( x < 100) ;
{
    < rest of loop>;
}

```

will loop *infinitely* since the miss-placed “;” means the while loop has a single null instruction as the loop body, and *not* the section in “{ }” as you expect!

(c) Forgetting to update the loop variable.

(d) Using “==” test on float numbers which, due to rounding errors, is *always* false. Note:

```
boolean testValue = Math.sqrt(2.0)*Math.sqrt(2.0) == 2.0;
```

will set testValue to false since not all 64 bits of Math.sqrt(2.0)*Math.sqrt(2.0) will be *identical* to 2.0.

If “reading the code” fails, then there are two strategies, these being:

- (a) Insert `println()` statement to check the value of key variables in the program. This is the “old-way” but is simple and valuable for finding errors in simple programs.
- (b) Use an *interactive debugger* or profiler. At present line debuggers for Java are very cumbersome and complex, not a sensible strategy for programs at this level.

A.2.3 Working Program – Wrong Results

This is the real *fun* one! If the program produces:

1. **Utter Rubbish:** For example a *constant* no matter what input you supply. The most likely problem is a program bug as discussed above. For example forgetting to assign a value to a variable, mistake in a loop which means that it never get executed etc. Search for bugs as described in the *Runtime Errors* above.
2. **Almost Right Results:** For example the right results with some data, wrong for others. This is not likely to be simple coding error, much more likely to be an error in the logic of the code, for example some conditional statement is wrong. This is tough to find.

There is also the worse-case synario, you have the underlying mathematics and/or physics of the calculation wrong. Then no matter how much you “play” with the program it will *always* produce wrong results!

Finally when you do make changes to the source code with `emacs` remember to use *Save Buffer* to update the source file on disk *and compile* the modified code with `javac`. I would prefer not to think about how many times I have failed to do one or other of these!

A.3 Problems with the Systems

Big computer system do sometimes “go wrong”. This usual symptom is they become *very slow* to respond or *stop* responding all together. This can be caused by anything from too many people trying to run big programs to a hardware fault on the server. If this happens:

1. Stop trying to do anything. Trying to open a new window or “playing” with the mouse will only make things worse. If however you are running a program that may have “looped” it may be *you* causing the problem. Try stopping it with `Ctrl-C` as discussed above or using the *Kill* option to stop the terminal window.
2. If nothing improves within about 30 secs:
 - (a) Call a demonstrator or see Mrs McIvor in the computing office for help.
 - (b) If there is nobody else the laboratory try and *Exit* the Window Manager, (try and log-off). If this also fails, leave the system alone, **do not** switch-off but leave a note on the terminal and/or see Mrs McIvor as soon as possible.

A.4 Myths, General Miss-conceptions and Classic Excuses

Round computing there are a whole series of *myths*, *miss-conceptions* and *total rubbish*, a few of which are:

1. The computer does “*what you tell it*” it is a deterministic machine *without* intelligence. It does *not* have a personal vendetta against you, despite what you may feel when all the other students programs run perfectly and yours completely fails!
2. Repeating the same *compile* or *execute* many times “*in case the computer made a mistake*” just wastes your time and add to your frustration and stress levels. You *will* get the same answer each time. Your program will *not* magically start working.
3. The chance of the “*computer being wrong*”, that is *you* finding a mistake in the *compiler* or system is about the same as you been struck by lightening on the way home! All non-trivial pieces of software do have “bugs” but the bugs in the compiler are likely to be so obscure that none of the simple programs that you will write will show them up.
4. The *PC urban myth* of “*the program getting corrupted on disk*” would result on the whole computer system crashing with lots of “panic” messages. It would *not* just effect your program.
5. The “*the computer lost my program*” is just possible, but it would not loose *just* yours. Lost files mean a computer hard disc fault which typically looses many files with lots of “panic” messages. In this unlikely event we are able to “restore” your files to the state they were in “yesterday evening”. This has only happened once in 8 years of running the CP-Lab, so it is very unlikely. If a file vanishes from the system it almost always means that *you deleted it*.
6. The “*somebody hacked into my account and changed my files*” is very unlikely *unless* you gave them your password; in which case tough! There is a finite chance of a machine being “hacked into” but if somebody does succeed in doing this is is rather unlikely they would modify one persons file when there is a whole system to muck up!

Experimental observation of the probability of “*occurrences*” 4,5, & 6 suggest a relation of the form $\exp(-\Delta T^2)$ where ΔT is the time (in days) left before a programming project deadline is due!

Notes: