INTRODUCTION

to

JAVA

for first year
Computer Science

Jane Meyerowitz
July 2002
Updated by D Moodley
July 2004

References

In compiling these lecture notes, much use was made of

Java Gently (2nd edition) by Judy Bishop.
Addison-Wesley 1998

Developing Java Software by Russel Winder and Graham Roberts.
John Wiley & Sons 1998

Java: how to program by Deitel & Deitel.
Prentice Hall 1997

Other books consulted were

Java, an object first approach by Fintan Culwin.
Prentice Hall 1998

Java: First Contact by Roger Garside and John Mariani
Course Technology 1998

programming.Java by Rick Decker and Stuart Hirshfield
PWS Publishing Company 1999

Java: A framework for programming and problem solving
by Kenneth Lambert and Martin Osborne
PWS Publishing Company 1999

Java: how to program by Deitel & Deitel
Prentice Hall 1997

These lecture notes are designed for use in the first year Computer Science modules at the University of KwaZulu-Natal. They provide an introduction to problem solving, programming, and the Java language. They are not intended to be complete in themselves but serve as a complement to the formal lectures, and students are urged to make use of the books referenced in addition to these notes.

Contents

10. Classes and Objects

10.1 An introduction to classes, objects, members and constructors
10.2 Java packages, classes and objects
10.3 Designing classes
10.4 Examples

11. Streams, Files and Exceptions

11.1 Input and Output streams
11.2 File input and output
11.3 Exceptions

12. Arrays

12.1 Simple arrays
12.2 Sorting
12.3 Tables
12.4 Searching

13. Strings

13.1 Strings
13.2 String Buffers
13.3 Tokenizers
13.4 class Keyboard and GenIO
13.5 toString methods

14. Recursion

14.1 Recursion

15. Useful Data Structures

15.1 Inner classes
15.2 Arrays of independent objects
15.3 Sorting arrays of objects
15.4 Merging data sets

16. Simple Graphics

16.1 Introduction to AWT and Swing
16.2 Event-driven programming
16.3 Components, containers and layout managers
16.4 Buttons
16.5 Panels
16.6 Text Areas, Text Fields and Labels

Appendix A – Errors and testing

A1 Coding for testing
A2 Debugging
A3 Testing

1. Introduction

1.1 Algorithms, machines and programs

A common misconception is that the hardest part of programming is writing the program language instructions that tell the computer what to do. This is in fact not the case - the most difficult part of solving a program on a computer is coming up with the method of solution. After you have developed a method of a solution, it is routine to translate your method into the required language. When solving a problem with a computer it is a good idea to first formulate the broad steps of the solution diagrammatically or in English or some form of pseudo-code, and then once all aspects have been considered, to translate your algorithm into the programming language.

A set of instructions that leads to a solution is called an **algorithm**. The term comes from the name of the ninth-century Arabian mathematician, Mohammed Al-Khwarizmi, who wrote an important book on the manipulation of numbers and equations, and described some routine processes for arithmetic and for solving equations. Hence the term algorithm came to mean a routine process for computation. Today, algorithm is taken to mean a finite ordered sequence of precise, step-by-step instructions for performing some task, usually a computation.

The idea of an algorithm is to describe a process so precisely and unambiguously that it becomes mechanical in the sense that it doesn't require much intelligence and can be performed by rote or by a machine. An algorithm has at least 3 necessary qualities:

- It must accomplish the task.
- It must be clear and unambiguous.
- It must define the sequence of steps needed for accomplishing the task - ie. define the steps *in order*.

In the early 1800's mathematicians began to dream of machines that could carry out boring mechanical computations. In addition, there was a need to improve the reliability of number-crunching and to find a way to automate the process in order to reduce errors. In 1840, a mathematician named Charles Babbage planned a steam-powered, gear-driven calculating machine to automate the calculation and printing of navigational tables. This "Analytical Engine" was to be controlled by three sets of punched cards: one for the data, one for the instructions for manipulating the data, and the third for controlling storage of intermediate results. The machine was never built because its gearwork construction lay beyond the capabilities of Victorian engineering.

But Babbage and his collaborator, Augusta Ada, Countess Lovelace, were responsible for one of the most important ideas of the Information Age - that of the Stored Program. A computing machine processes data in the form of symbols. In doing this, it is controlled by instructions that make up its algorithm. These instructions must also be provided to the machine as sequences of symbols. All that was needed was a language for symbolising the instructions and the computing machine could be provided with a *program* for its activities.

Today we distinguish between an algorithm and a program. A **program** is an expression of an algorithm in a precise language that can be made understandable to a computer. The language itself is called a **programming language**.

1.2 Programming Languages

A program is an algorithm written in a form that is understandable to a computer system. Each CPU is designed to execute a very simple collection of instructions called its machine language. Before the 1950s, there were no high-level programming languages. Programmers were forced to program in machine language (i.e. the language understood by the machine). Machine code is simply a sequence of bit patterns (1's and 0's) which are interpreted by the machine as commands and data.

In the early 1950's, a low-level language was developed called assembly language. This language is really machine code disguised in slightly more friendly clothing, where each machine-language instruction is symbolised by an abbreviation (eg. `ADD`, `LD`, `STO`). An assembly-language program would first be translated by another program called an assembler into machine code which could then be executed.

By the mid-1950's programmers were asking for high-level programming languages which would allow algorithms to be described more nearly as humans think about them. FORTRAN (FORmula TRANslation) was invented for scientific programming and COBOL (COmmon Business Oriented Language) was invented for business applications. Programs called compilers are used to translate programs written in these languages into machine-level code. It was found that high-level code in FORTRAN or COBOL could be written more quickly than machine-level or assembly code, and with fewer errors.

As it became easier to write large, complex programs, new problems were encountered. A large, complex program was hard to understand and therefore hard to write correctly. By the 1960's, program correctness was an issue of great concern and computer scientists sought new programming language features that would aid correct programming.

The Pascal language was designed about 1970 by Niklaus Wirth, a Swiss computer scientist. It was named after the French mathematician, Blaise Pascal, who designed a mechanical calculator about 1644. It was designed as a teaching language, a small, easily learned language, with features for data organisation and algorithmic design that would encourage clear, correct programming. Its features were constrained by Wirth's demands that the language should be simple and easily compiled, that the compiler should clearly indicate grammatical errors in the program it is translating, and that the compiled programs should run efficiently.

The language C was invented by Dennis Ritchie at Bell Labs in 1972. Its unromantic name evolved from earlier versions called A and B. C produces code that approaches assembly language in efficiency while still offering high-level language features such as structured programming. C compilers are simple and compact. Although C is simple and elegant, it is not simple to learn - the learning curve is very steep.

One of the major differences between C and Pascal is that of the language philosophy and is also the reason why Pascal rather than C is still favoured as a teaching language, in spite of C's growing popularity in the business environment. This philosophy is that Pascal assumes the user is a novice and traps any instructions that have unusual and presumably unintended side-effects, whereas C assumes that a user knows what he's doing, error messages are often brief and ambiguous, and strange side-effects are accepted and are in fact one of the features of the language.

C++, an extension of C, was developed by Bjarne Stroustrup in the early 1980s at Bell Labs. C++ provides a number of features that "spruce up" the C language, but more importantly, it provides capabilities for **object-oriented programming**. Building software quickly, correctly, and economically remained an elusive goal. Objects are an attempt to help solve this problem. Objects are essentially reusable software components that model items in the real world. Software developers discovered that using a modular, object-oriented design and implementation approach can make software development groups much more productive than was possible with previously popular programming techniques such as structured programming. Object-oriented programs are easier to understand, correct and modify. As a teaching language, however, C++ retains the disadvantages of C.

The last few years have seen a phenomenal rise in interest in the Internet and the World Wide Web (WWW). Tens of millions of users regularly access this network to carry out operations such a browsing through electronic newspapers, downloading bibliographies, participating in news groups and e-mailing friends and colleagues. The number of applications that are hosted within the Internet has also grown - however, there are major problems in developing such applications such a security, lack of specific programming languages, lack of interaction and non-portability.

- The security issue involves ensuring that unauthorised access is prevented. This is a major problems and one of the reasons why commercial applications, particularly those involving the direct transfer of funds across communication lines, were slower in developing as compared with academic applications.

- Until the development of Java, there was no specific Internet programming language for WWW applications. Applications were written in a wide variety of languages which frequently involved dealing with low level facilities such as protocol handlers.

- It was difficult to build interaction into a WWW application. Most of the applications that had been developed tend to give the impression of being interactive but they often just involve the user in following links and moving through a series of text and visual images. In many cases, the only interactivity was asking for a user name and password and checking these against some stored data.

- Many applications were non-portable; they tend to be anchored firmly within one computer architecture and operating system because they use run-time facilities provided by one specific operating system.

The Java programming language originated at Sun Microsystems in 1991 as part of a research project to develop software for consumer electronic devices - television sets, VCRs, cellphones, etc. At that stage it was called Oak (after a tree the leader of the development team, James Gosling, could see from his window). However, while it was being developed the Internet and the WWW exploded into widespread use. A decision was made to adapt the language to the needs of the Internet and provide an Internet programming language for WWW-based applications. This decision was made because Oak included many features that were relevant for the Internet environment, including the idea of being architecturally neutral, which means the same program can run on a wide variety of machines.

In January 1995 Oak was renamed Java (because the name Oak was already in use) and was developed into a robust programming language for building WWW-based applications. It has since developed into a full-scale development system, quite capable of developing large scale applications that exist outside of the WWW environment, including those that make extensive use of networking to allow a language to communicate. Why the name Java? Popular legend has it that it is named for a coffee, copious quantities of which are supposedly required for programmers to be able to function. Others allege that it is an acronym for Just Another Virtual Architecture. Whatever the origin of the name, the coffee-cup logo is now an integral part of the Java culture.

The designers of the language had a number of goals:

- The language should be familiar, it should have no strange syntax and, as much as possible, it should look like an existing language but problems associated with other languages should not be carried through to Java. The developers decided to make it resemble the C and C++ family of programming languages, and its similarity to these languages means that a wide variety of users are able to program in it.

- The language should be object-oriented. A programming language is object-oriented if it offers facilities to define and manipulate objects. Objects are self-contained entities which have a state and to which messages can be sent.

  An object-oriented programming language has two major advantages, First, by adhering to a small set of programming principles it is possible to write systems which are relatively easy to modify. This is important since all the surveys that have been carried out on the amount of resources expended on software development have come up with figures which suggest that companies who have a significant software development capability spend between 60% and 80% of their development in changing existing software.

  The other feature of an object-oriented programming language is that it allows a high degree of reuse. One of the features of the Java system is a large library containing many useful objects that can be used when required.

- The language should be robust. One of the problems with some of the more popular programming languages is the fact that it is quite easy to produce applications which collapse. Sometimes this collapse can manifest itself immediately; however it can also occur outside the application because, for example, the application has corrupted some

memory which is not used immediately. One of the design aims of the developers of Java was to eliminate such features, for example, there is no concept of a pointer in Java.

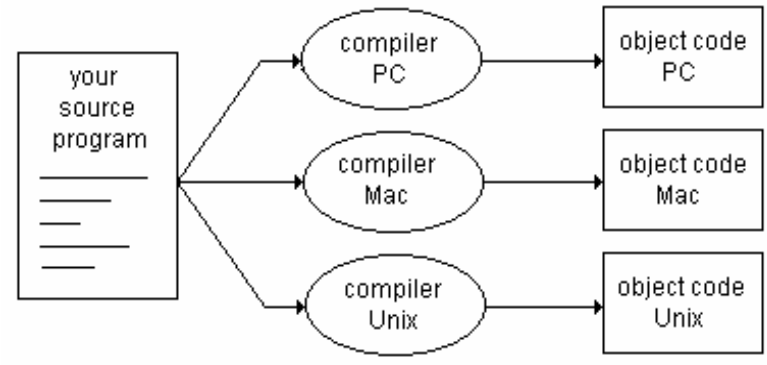
- The language should be portable, and a program developed in Java for eg, a Sun workstation running the Solaris operating system, should be capable of being executed directly on any other operating System, eg Windows NT or Windows 98.

- The language should be as simple as possible. Many languages have been overburdened with features. This has a number of effects: first, such programs are often expensive to compile and require a large amount of memory to run; second, the learning curve for such languages is long and hard; and third, compiling programs in such languages can take quite a long time. The designers of Java have tried to keep the base facilities of the language to a minimum and have provided the extra features within a number of libraries that may be included if the program requires them.

1.3 The compilation process

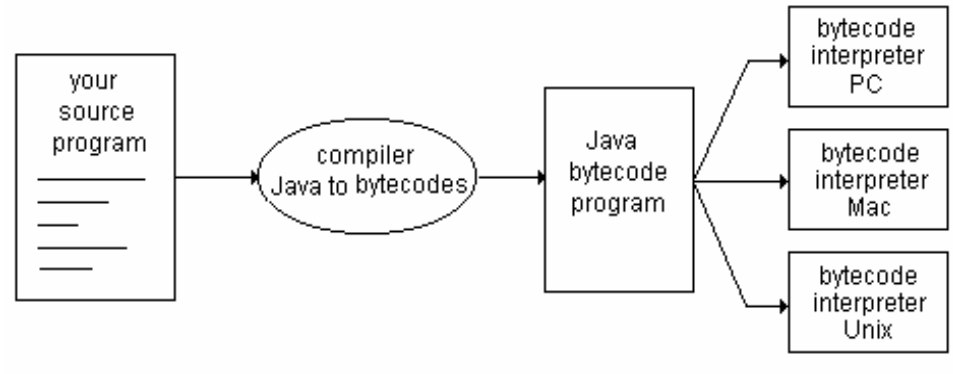
A program consists of text created by means of a simple text-processing program called an editor. The editor allows text to be entered into main memory from the keyboard or from a disk file, and then allows deletions, insertions and changes to be made. This text can be saved to a disk file - a Java file usually has the extension `.java`. These high-level program language instructions are known as the **source code**. A program in source code cannot be executed directly because a computer cannot execute high-level language instructions, only machine code. The program must first be translated by the compiler into machine code, known as the **object code**. The compiler reads the source code, checks it for errors, and if error free generates the object code. This is held in memory and can then be executed, or the object code itself can be saved to disk and then executed directly without having to be recompiled.

This description applies to most programming languages such as C and Pascal. The object code which the compiler produces is, of course, specific to the actual system on which the compilation takes place - ie. your object code will only execute on the system it was compiled on since the machine code is only recognisable by that system. If you want to use the same program on another system you have to recompile your original source code using a compiler for that system. The process is:



When you write source code in Java, the process is different. The Java development environment has 2 parts: a Java compiler and a Java interpreter. The Java compiler takes your Java source program and instead of generating machine code for a particular machine and

operating system it instead generates machine code for a hypothetical machine called the Java Virtual Machine (JVM). Java calls this machine code **bytecodes**. The bytecodes are not the native machine code of the computer, so they are executed by a bytecode interpreter running on your machine, which is a much simpler program than a compiler. A bytecode interpreter is supplied with the Java language, and there is also an automatic bytecode interpreter built into most Internet browsers such as Netscape and Internet Explorer.



Java programs come in two forms, **applications** and **applets**. Java applications are just like any ordinary program, while applets are mini-applications that are embedded in WWW pages. When a page containing an applet is displayed, the applet (in bytecodes) is downloaded to the users computer and executed locally. This is only possible because Java is architecturally neutral and the bytecode programs can run on any machine with an interpreter. Because applets are downloaded from a remote site to run on a local computer they are subject to a number of security restrictions, for example, they cannot access local files or print. Applications, on the other hand, are not restricted in what they can do, and any kind of program can be written.

1.4 Errors

There are three areas where errors can occur:

- Compiler errors - during compilation of your program.
- Runtime errors - while your program is running.
- Logical Errors - during the development and implementation of your program as a result of an error in your solution to the programming problem.

Compiler Errors:
Compiler errors, or *syntax* errors, occur when your code violates a rule of the language syntax. Syntax errors involve invalid word order, missing punctuation (punctuation delimiters are often obligatory in programming languages), spelling mistakes and undeclared variables. Usually, when a compiler comes to a statement it can't understand it stops compiling and displays the offending statements with an error message.

Runtime Errors:
Runtime errors, or *semantic* errors, occur while executing your program. The program contains legal program language statements (or syntax errors would have been reported) but does

something illegal when you execute it. Some common examples are: divide by zero, reading in the wrong type of value, integer too large, trying to open a file that doesn't exist.

Logical Errors:
These errors will never be directly reported to the programmer and may therefore be the most difficult to detect of all error types. They are errors in design and implementation and occur when the programmer has written syntactically and semantically correct code which does not achieve the desired result. ie. the programmer's logic is incorrect. To find and solve these types of errors:

- Test your program extensively by predicting the results for various inputs of the program before you run it (called dry running).
- Once the program is running, compare your predicted results with the results given by the program.
- If the two sets of results do not match then you probably have made a logical error.

Solving Errors
Remember that your program always has a potential error. You should (must!!!) spend time testing your program to see that it is free of errors. Good error testing will improve your programming and the final product of your program. Never say that you have written a perfect program. Even the greatest programmers admit that their programs are prone to errors (bugs).

Some general procedures for testing programs are:

- Always test the boundary conditions eg. If your program finds all the squares of integers between -10 and 10 (inclusive), then check the program works for -10 and 10.
- Always (if applicable) check for the case where a variable is zero. This test often turns up potential division by zero among other interesting errors.
- If possible, "watch" the execution of your program using a Debug facility. This will usually show you how the values of variables change and show up possible errors.
- Predict (manually work out) the solution to your program before running it. But do not to force your program to produce those results. Understand the discrepancies between the predicted and determined results.

2. Problem Solving

In this course you are going to learn how to program. By *"program"* what is meant is:

1. Problem Solve
   - analyse a problem
   - decide what steps need to be taken to solve it
   - take into consideration any special circumstances
   - plan a sequence of actions that must take place in a specified order
   - check your plan

2. Code
   - translate the plan into a programming language
   - enter it to the computer

3. Test and Debug
   - compile the program and correct any syntax errors
   - execute the program with the relevant data
   - check the results and if not as expected correct any logic errors

4. Documentation
   - write a report describing your program

Many people think of programming as merely step 2 but this is in fact just coding - writing the program in a form which can be understood by a computer. The crux to programming is problem solving because if the problem has been incorrectly or incompletely solved, no amount of coding will generate a correct solution. Inventing algorithms is the central task of computer science. The goal is to take a problem, decompose it into simpler parts, and then to imagine a sequence of steps by means of which a machine can generate a solution.

2.1 Algorithms

In order to solve a problem an algorithm or list of steps must be devised to describe a possible solution. There isn't necessarily just one correct algorithm or way of doing things, there may be alternative approaches that achieve the same end result in different ways. However if there is more than one way of doing things then some consideration should be given to whether one method is better than another (faster, more efficient, clearer etc).

Consider the problem of a mother planning her afternoon. There are two possible approaches:

- Drop Anne at ballet at 2.00
- Drop Steve at swimming at 2.30
- Fetch Anne from ballet at 3.00
- Fetch Steve from swimming at 3.30
- Take them both to do the shopping with her
- Go home and prepare dinner

or

- Drop Anne at ballet at 2.00

- Drop Steve at swimming to wait for his 2.30 lesson
- Do the shopping
- Fetch Anne from ballet a bit late
- Fetch Steve from swimming
- Go home and prepare dinner

You should ask yourselves the following questions:

- *Which plan is to be preferred ?*
- *Is there perhaps another plan which is better ?*

There is not necessarily one correct algorithm. All possible alternates should be considered and a choice made.

Consider another type of algorithm: a recipe. This very often has the form

1. Preheat the oven to 180C
2. Grease two cake tins
3. Cream the butter and sugar
4. Sift the dry ingredients
5. Add the eggs to the creamed mixture and beat well
6. Fold in the dry ingredients
7. Pour into cake tins and bake

Now ask yourselves:

- *Are there alternate algorithms ?*
- *How important is the order ?*

In many algorithms the order in which the events are done is crucial, in others some events can be resequenced without any effect.

By now you should have realised that the key structure in composing an algorithm is sequence. Actions follow one after the other:

do this *and then* do that *and then* do the next etc.

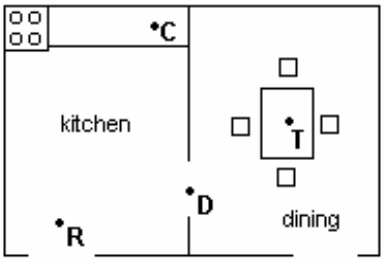
Even if the order of certain actions is immaterial, we must choose a particular sequence of actions and specify it in our algorithm.

Algorithms define a sequence of events which must take place one after the other.

One of the fundamental methods of problem solving is to break a large problem into several smaller subproblems. In this way we can solve a large problem one step at a time, rather than attempt to provide the entire solution at once. This technique is often referred to as **stepwise refinement** or divide and conquer.

For example, let us look forward to the year 2000. Our household robot, Robbie, helps us with some simple chores. Each morning we would like Robbie to serve us breakfast. Unfortunately Robbie is an early production model and to get him to perform even the simplest task we must provide him with a detailed list of instructions. In this case, the problem to be solved is getting Robbie to serve breakfast.

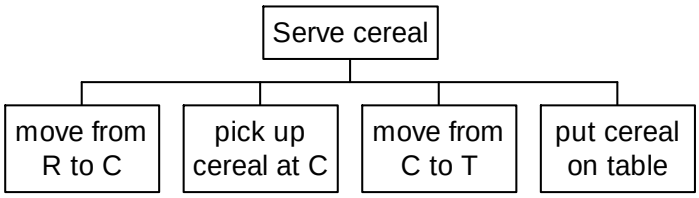
In the diagram Robbie is at point R (for Robbie). We want Robbie to fetch our favourite box of cereal (at point C) from the kitchen and bring it to the table (at point T) in the dining room.



As a design overview, we can accomplish this goal by instructing Robbie to perform the following steps:

1. Move from point R to point C.
2. Pick up the cereal box at point C.
3. Move from point C to point T.
4. Place the cereal box on the table at position T.

or in JSP notation (Jackson Structured Programming):



Solving these 4 subproblems will give us the solution to the original problem.

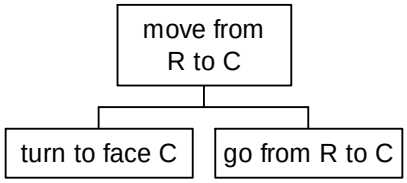
Assume that the basic operations Robbie can perform are

- rotate or turn to face any direction
- move straight ahead
- grasp and release specified objects

Returning to the design overview we see that steps 2 and 4 are basic operations provided Robbie is in the correct position.

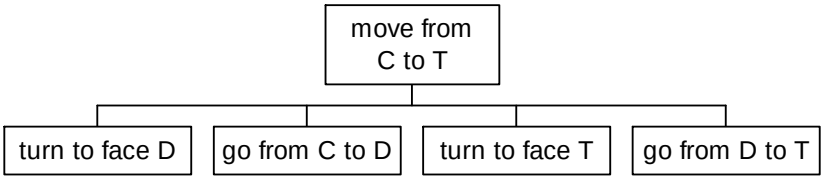
In solving step 1 we must allow for the fact that Robbie can only move 1 direction at a time, and that direction is straight ahead. Consequently the steps required are:

1.1 Turn to face point C.
1.2 Go from point R to point C.



Step 3 can be solved in a similar way. However, since Robbie cannot walk through walls the steps required are:

3.1 Turn to face the doorway (point D).
3.2 Go from point C to point D.
3.3 Turn to face point T.
3.4 Go from point D to point T.

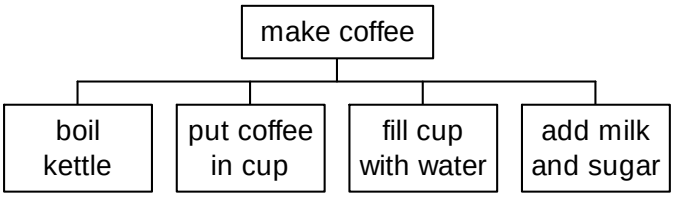


What we have done is to divide the original problem into 4 subproblems, all of which can be solved independently; then we broke up two of these subproblems into even smaller subproblems. So the complete algorithm is:

1. Move from point R to point C.
   1.1 Turn to face point C.
   1.2 Go from point R to point C.
2. Pick up the cereal box at point C.
3. Move from point C to point T.
   3.1 Turn to face the doorway (point D).
   3.2 Go from point C to point D.
   3.3 Turn to face point T.
   3.4 Go from point D to point T.
4. Place the cereal box on the table at position T.

Consider another example, that of making a cup of coffee.

1. Boil the kettle.
2. Put a spoon of coffee in the cup.
3. Fill the cup with boiling water.
4. Add milk and sugar.



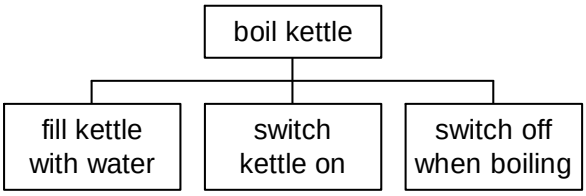
These are relatively independent tasks and can be broken down into smaller subtasks.

First consider step 1. Let's assume we are boiling the water in an electric kettle. Our refinement could look something like

1.1 Switch the kettle on.
1.2 When the water is boiling switch the kettle off.

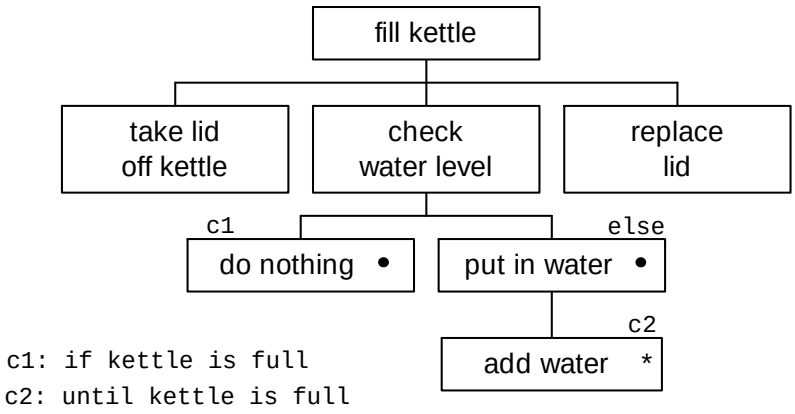
But wait - surely we should check first to see if there is enough water in the kettle!

1.1 Fill the kettle with water.
1.2 Switch the kettle on.
1.3 When the water is boiling switch the kettle off.



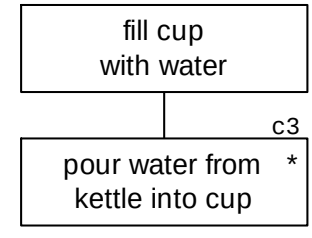
Steps 1.2 and 1.3 are OK, but we probably need to expand step 1.1 a little.

1.1.1 Take the lid off the kettle.
1.1.2 If the kettle is full do nothing otherwise repeatedly add water until it is full.
1.1.3 Replace the lid.

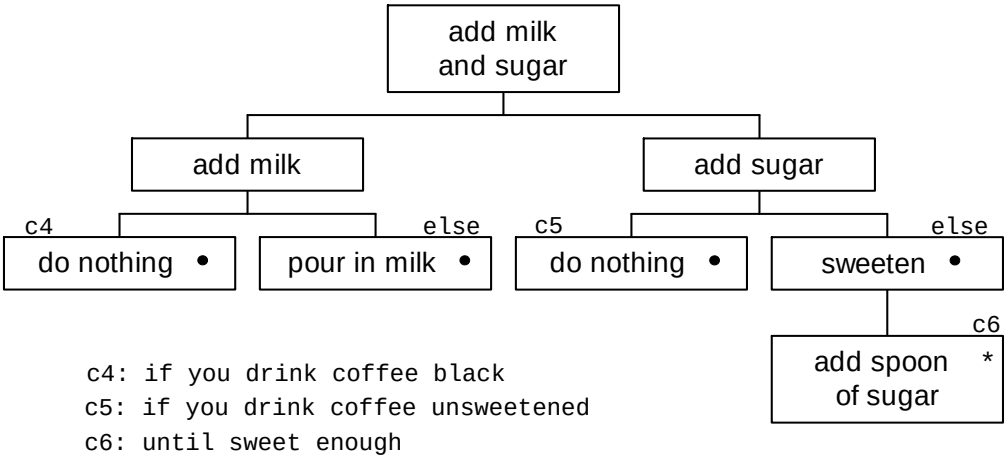


Step 2 could stand as it is, although we could consider opening and closing the coffee tin.

Step 3 ("fill the cup with boiling water") needs expanding.



And step 4 ("add milk and sugar") needs expanding.



The algorithm for making coffee has three categories of action: ***sequential execution*****,** ***conditional execution*****,** and ***repetition*****.** These three categories of action occur in computer programs as well.

2.2 Mathematical Algorithms

When we attempt to solve problems that are presented either verbally or in writing, one of the biggest problems is that we do not pay close enough attention to the problem statement to determine what is being asked. As a result the solution is often incorrect because it solves the wrong problem. To successfully solve a problem you must analyse the problem statement carefully before you try to solve it. You may need to read each problem statement two or three times. The first time, get a general idea of what is being asked. The second time, try and answer the questions

- *What information should the solution provide?*
- W*hat data do I have to work with?*

The answer to the first question will tell you the answers required, or the problem outputs. The answer to the second question will tell you the data provided, or the problem inputs. Then the problem resolves to one of how to get the desired results from the given input.

Consider now a more mathematical problem - that of finding the average of a list of numbers.

Problem: Given the numbers 21, 13, 20, calculate their average.

- *What information should the solution provide?*
- *What data do I have to work with?*

Obviously we need to calculate the total of the three numbers, and then divide the total by 3 to get the average. Using a computer we could allocate 3 memory cells to hold the three numbers, say

- get 21, store in *no1*
- get 13, store in *no2*
- get 20, store in *no3*

Now 2 more cells are needed to hold the total and the average:

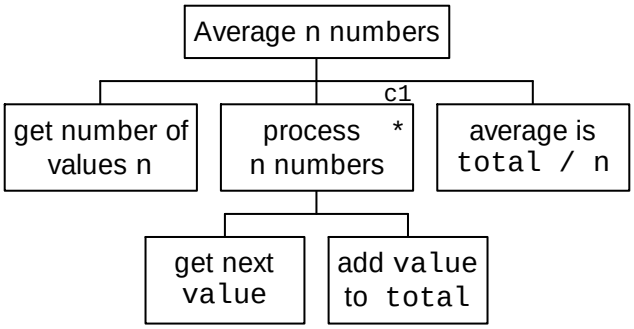
- *total* is set to *no1 + no2 + no3*
- *average* is set to *total / 3*

If however the problem involves averaging 4 values a different program must be created with an extra memory cell to hold the fourth value. And what if we have to average one thousand values - will we then require 1000 memory cells?

There is a way we can write a general algorithm to cater for any number of numbers. It uses only 4 memory cells, the control structure *repetition*, and is a general averaging algorithm that can be used with any number of values.

The 4 memory calls needed are

- *n* to store the number of values to be averaged,
- *value* to store each of the *n* values in turn,
- *total* and *average*

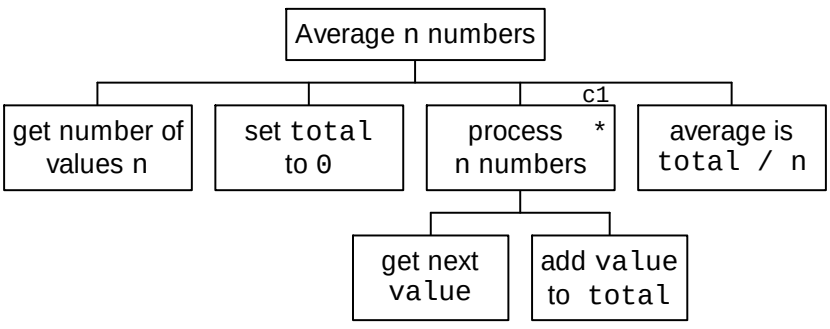


c1: until n numbers read

Will this algorithm give the correct answer?

Remember that you are working with memory cells and there is no guarantee that any particular memory cell is set to zero initially. Sometimes this is so, but often a memory cell may be holding left-over data from earlier use. If the memory cell `total` happens to be holding 0 the algorithm will work correctly because `0+x=x`, but if `total` is holding some other value the result of `total+x` is not equal to `x`. To ensure we always get the right answer irrespective of whether or not anything was stored in the memory cell beforehand, a memory cell that is going to be used in a calculation should always be initialised.

The correct algorithm would therefore be:

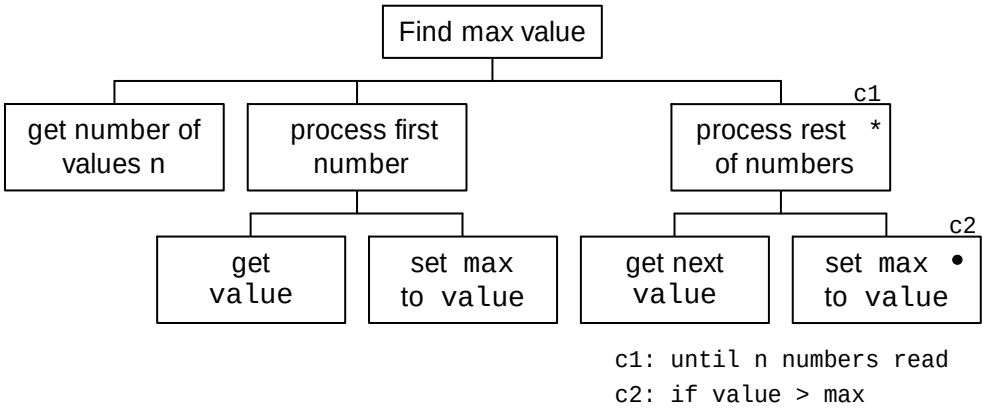


c1: until n numbers read

Problem: Which is the largest of the numbers 10, 15, 3, 2, 14, 18, 27, 19, 7, 23 ?

- *What information should the solution provide?*
- *What data do I have to work with?*

The solution is to scan the list of numbers and compare each number to the "largest so far" until all the numbers have been considered at which stage the "largest so far" is the largest in the list. The algorithm includes the control structure *comparison and conditional execution*, and can once again be generalised to deal with any number of values along the lines of the averaging algorithm. Then we need 3 memory cells only, *n, value* and *max*.



Exercises

2.1 Write a algorithm that describes how to get on a bus and sit down. (Remember to check that there are empty seats).

2.2 Write an algorithm that describes how to get from your bedroom to the bathroom.

2.3 Write an algorithm that explains how to cross the road at a pedestrian crossing with a traffic light.

2.4 Design an algorithm to successively subtract all the numbers in a list from 1000 and display the result.

2.5 Design an algorithm to find the smallest value in a list of numbers.

2.6 Design an algorithm to find the largest and smallest value in a list of numbers.

2.7 Design an algorithm to compare the average of a list of numbers with its range mid-point. The range midpoint is calculated by determining the largest and smallest number in the list and averaging them. Both the average and the range mid-point must be displayed, together with a message saying which is larger.

2.8 Design an algorithm to convert a Fahrenheit temperature to Celsius. Display an error message if the Fahrenheit temperature is below absolute zero (-459.7°F).

2.9 Design an algorithm to arrange any 3 given numbers n_1, n_2, and n_3, so that $n_1 \leq n_2 \leq n_3$

3. Simple Programs

When learning a programming language, the easiest approach is to look at some examples of simple programs. They serve to introduce the general structure of a program, and can be used to explain how to enter, compile and execute (run) programs.

3.1 A first program

This is an example of a simple Java application program. When it is run, it displays

```
****************
*              *
* Hello World! *
*              *
****************
```

in the output window on your screen.

```
/**
* My first program
* written by Jane Meyerowitz 23/07/98
* -----------------------------------
*/

public class Hello
{
  public static void main(String[] args)
  {
    System.out.println("****************");
    System.out.println("*              *");
    System.out.println("* Hello World! *");
    System.out.println("*              *");
    System.out.println("****************");
  } // end of method main

} // end of class Hello
```

Hello.java

The first thing to be aware of is that Java is **case-sensitive**. This means that the use of upper-case (capital) and lower-case letters is significant. For example, `PUBLIC`, `public` and `Public` are not the same, and the use of the incorrect case will cause errors.

The first 5 lines are a comment - arbitrary text between the characters **`/*`** and **`*/`**. Everything between these two character pairs is ignored by the Java system, and is used by the programmer to add commentary to programs to help the human reader understand the structure and purpose of the program. As a general rule, you should always include a header comment such as this at the beginning of your programs, giving your name and a brief statement of the function of the program.

There is another form of comment permitted in Java :

```
// end of method main
```

These comments begin with the double slash `//` and extend to the end of the current line. They are useful for adding in-line commentary within the body of a program.

Everything in a Java program must be part of a class, and the statement

```
public class Hello
```

defines a class called `Hello`. The entire class itself is contained between the curly brackets `{` and `}`. The comment after the closing curly bracket is useful in matching up the curly brackets.

In addition, in order to try and make it easier to match up pairs of brackets it is usual to line up the matching pairs with each other, and to indent the program after an opening curly bracket, as shown in the program. These layout conventions are used purely to help the human reader - Java itself ignores all indentation, spaces between words, and blank lines. However careful program layout can help clarify the structure of a program to the human reader, particularly a large, complex one.

The word `public` is an access modifier - it specifies the circumstances under which the class can be accessed - and this class can be publicly accessed. Initially, all the programs you write will be public.

In Java, it is a rule that the name of the file in which the program is saved must be the same as the public class name - hence this class (program) is saved in a file called `Hello.java`. (There may only ever be one public class in a file.)

The class `Hello` contains only one thing - a piece of code called a **method** that carries out a task. Each Java application program needs a main method which is always the first method executed when a Java application is run. The header that defines the main method is

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

Later we'll consider the different components of this header - for now just accept that this is the way to define the main method and copy this statement when required.

Once again, the curly brackets `{` and `}` are used to define the beginning and end of the main method, whose contents are indented inside the braces, with a comment after the closing brace to make things quite clear.

The body of a method consists of one or more statements, each of which is terminated by a semi-colon (`;`). In this example, the body of the main method consists of 5 statements, each of which function to output a line of text.

The statement used to output data to the user's output window is

```
System.out.println(…);
```

where the … denotes whatever is to be output. When this statement is executed the values specified in the round brackets (parentheses) are displayed and the cursor is moved to the next line, so that any further output does not appear on the same line.

In this example there are 5 `println` statements, so the output will appear on 5 lines. In each case the value to be output is some text enclosed in double quotes ("). This denotes a character string, which is then displayed in the output window. Other types of values can also appear in the parentheses, as we shall see later.

To leave a blank line, don't put any values in the parentheses

```
System.out.println();
```

3.2 Running a Java application program

We are using the Kawa IDE (Integrated Development Environment) to enter, edit, compile and run the Java programs.

Your Java program should be entered into an edit window. As you type, you'll notice that different components of the program appear in different colours - comments in green, strings in red, other statements in black etc. This is merely an aid to the human user and helps differentiate between the different types of words and statements in the program.

The program should then be saved to a file with the same name as the public class name - Eg, our example above is saved in a file called `Hello.java`. I use folders (directories) to contain groups of program files - for example all the programs in this chapter are saved in a folder called `3SimpleProgs`. With more complex programs containing many classes there may be a number of files and I would save each program in its own folder. The folders can have any name - I like to choose one that is descriptive of the programs functionality.

Once the program has been entered it needs to be **compiled**. This is the process of checking the program for compilation or **syntax errors**. These are errors such as spelling mistakes, incorrect case, missing semi-colons, unmatched brackets etc etc. If the Java compiler (`javac`) finds any errors it will report them in the Build page of the Output windows.

For example, writing `public` with an initial capital `P` causes the following error:

```
C:\Java 1.2\bin\javac.exe   Hello.java
File Compiled...

--------------------------- Javac Output -------------------------
Hello.java:7: Class or interface declaration expected.
Public class Hello
^
1 error

------------------------------------------------------------------
```

In this case the error is in line 7 of `Hello.java` as indicated.

Sometimes a single mistake can cause a number of errors. For example, leaving out the semicolon at the end of the first `println` statement results in 2 errors, one in the statement with the missing punctuation and one in the next.

```
C:\Java 1.2\bin\javac.exe   Hello.java
```

```
File Compiled...

--------------------------- Javac Output -------------------------
Hello.java:11: Invalid type expression.
     System.out.println("****************")
                        ^
Hello.java:12: Invalid declaration.
     System.out.println("*              *");
                        ^
2 errors
------------------------------------------------------------------
```

If there are errors in your program they must be corrected, and the program recompiled until it is error free.

At this stage the `javac` compiler has created a new file in the same folder as `Hello.java` called `Hello.class`. This consists of Java bytecodes which can then interpreted when the program is executed. When the `Hello.java` file is in the current edit window and **Run** is selected, the Java bytecode interpreter `java` looks for a file called `Hello.class`. If it finds the file it looks for the method called `main` and executes it. The first statement is executed, then the second, and so on. When the last statement in `main` has been executed the program terminates. The output window will display the result of the program execution:

```
C:\Java 1.2\bin\java.exe   Hello
Working Directory - C:\javaprogs\3SimpleProgs\
****************
*               *
* Hello World! *
*               *
****************
Process Exit...
```

Even if your program compiles correctly it is possible it may not execute because of run-time errors - errors that are only detected when the Java interpreter attempts to execute the statements.
For example, if the word `static` is omitted from the header for method `main`,

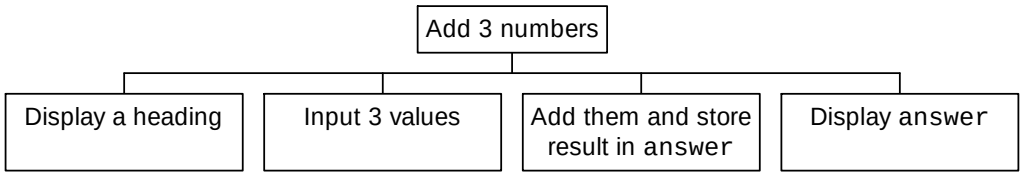
```
    public void main(String[] args)
```

`Hello.java` will not give any syntax errors, but when an attempt is made to run it, it does not execute:

```
C:\Java 1.2\bin\java.exe   Hello
Working Directory - C:\javaprogs\3SimpleProgs\
In class Hello: main must be public and static

Process Exit...
```

3.3 A second example program

For a second example, consider a slightly more complex program that reads in 3 numbers from the keyboard, adds them, and displays a total.




```
/**
* Reading 3 numbers and adding them
* written by Jane Meyerowitz 23/07/98
* -----------------------------------
*/
import Utilities.Keyboard;

public class ReadSumNos
{
  public static void main(String[] args)
  {
    double num1,num2,num3;   // the 3 numbers to be input
    double answer;           // the sum of the 3 numbers

// display a heading
    System.out.println();
    System.out.println("This program reads in 3 numbers"
      + " and adds them");
    System.out.println("--------------------------------"
      + " -------------");
    System.out.println();

// input 3 values
    System.out.print("  Enter the first number  > ");
    num1 = Keyboard.getDouble();
    System.out.print("  Enter the second number > ");
    num2 = Keyboard.getDouble();
    System.out.print("  Enter the third number  > ");
    num3 = Keyboard.getDouble();
    System.out.println();

// add them and store the result in answer
    answer = num1+num2+num3;

// display answer
    System.out.println("The sum of the three numbers is "
      + answer);
    System.out.println();
  }
}
```

ReadSumNos.java

This program requires 3 variables to store the values that are read in, and another variable to store the result. A **variable** is the name given to a memory location that has a name (identifier) associated with it and can store a value. The variable name is used to refer to the value currently stored at that memory location. The variables in this program are declared at the start of the main method.

```
double num1,num2,num3;   // the 3 numbers to be input
double answer;           // the sum of the 3 numbers
```

The first declaration defines `num1, num2` and `num3` to be the names associated with 3 memory locations that can store values of datatype `double` (ie. a floating point value of about 15 significant digits) *(More about variables and data types later)*. The second declaration defines another `double` variable with the name `answer`. The comments explain what these variable represent.

The program statements to input the numbers consist of pairs of statement such as

```
System.out.print("  Enter the first number  > ");
num1 = Keyboard.getDouble();
```

The first of these is merely an output statement displaying a message in the output window telling the user what to enter. It is called a prompt. The second statement is a call to a class `Keyboard` which contains a number of methods to read in different types of values from the keyboard. The input facilities provided by Java are complicated and tedious to use, so these methods have been written for you. The call to `Keyboard.getDouble()` (note capitalisation) will wait for the user to enter a double value and then store it in the variable specified - in this case `num1`.

When the program is run, it will execute the statements up to the first input statement and then pause for the user to enter some input:

```
C:\Java 1.2\bin\java.exe   ReadSumNos
Working Directory - C:\javaprogs\3SimpleProgs\

This program reads in 3 numbers and adds them
---------------------------------------------

  Enter the first number  >
```

The user should type in the value required, and press the `ENTER` key:

```
C:\Java 1.2\bin\java.exe   ReadSumNos
Working Directory - C:\javaprogs\3SimpleProgs\

This program reads in 3 numbers and adds them
---------------------------------------------

  Enter the first number  > 25
  Enter the second number >
```

The next pair of statements have now been executed,

```
  System.out.print("  Enter the second number > ");
  num2 = Keyboard.getDouble();
```

and the program halts for the second number to be entered, and then displays the prompt for the third number and halts for it to be entered.

```
C:\Java 1.2\bin\java.exe   ReadSumNos
Working Directory - C:\javaprogs\3SimpleProgs\

This program reads in 3 numbers and adds them
---------------------------------------------

  Enter the first number  > 25
  Enter the second number > 10.2
  Enter the third number  >
```

When all three values have been entered, a simple arithmetic statement is used to add the three numbers and store the result in the variable named `answer`.

```
  answer = num1+num2+num3;
```

Notice that using the variable name (eg. `num1`) denotes *"the value currently stored in the memory location referenced by the name `num1`"*.

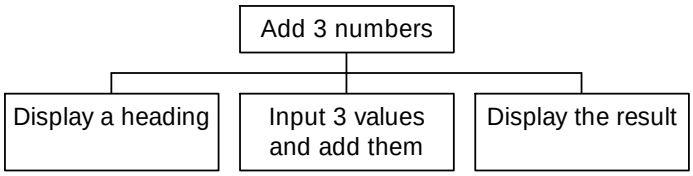
Finally a `println` statement displays a message and the value of `answer` ("*the value currently stored in the memory location referenced by `answer`*").

The overall effect of running this program and entering the values 25, 10.2 and -1 when requested is: (*user input in bold*)

```
C:\Java 1.2\bin\java.exe   ReadSumNos
Working Directory - C:\javaprogs\3SimpleProgs\

This program reads in 3 numbers and adds them
---------------------------------------------

  Enter the first number  > 25
  Enter the second number > 10.2
  Enter the third number  > -1

The sum of the three numbers is 34.2

Process Exit...
```

3.4 A final example

The last example has the same function - reading in and adding 3 numbers, but uses a modular style in order to achieve this, where modules (called procedures or methods) are used to perform different sub-goals. The main method serves as a top-level controller and calls the other methods that perform the actual work. This modular style of programming has a number of advantages, chief among them the structuring of complex problems and breaking them down into manageable components.




```
/**
* Using methods to read 3 numbers and add them
* written by Jane Meyerowitz 27/07/98
* -----------------------------------
*/
import Utilities.Keyboard;

public class ReadSumNos2
{
  public static void main(String[] args)
  /**-----------------------------------
  * the main method calls other methods to do the work
  */
  {
    double total;           // the sum calculated

    DisplayHeading();
    total = SumNos();
    DisplayAnswer(total);
  }

  static void DisplayHeading()
  /**-------------------------
  * Displays information about what this program does
  */
  {
    System.out.println();
    System.out.println("This program reads in 3 numbers"
      + " and sums them");
    System.out.println("-------------------------------"
      + " -------------");
    System.out.println();
  }
```

ReadSumNos2.java *part1*

```
  static double SumNos()
  /**-------------------
  * this method inputs the 3 numbers, adds them,
  * and returns the answer
  */
  {
    double num1,num2,num3;   // the 3 numbers to be input
    double answer;           // the sum of the 3 numbers

    System.out.print("  Enter the first number  > ");
    num1 = Keyboard.getDouble();
    System.out.print("  Enter the second number > ");
    num2 = Keyboard.getDouble();
    System.out.print("  Enter the third number  > ");
    num3 = Keyboard.getDouble();
    answer = num1+num2+num3;
    return answer;
  }

  static void DisplayAnswer(double answer)
  /**-----------------------------------
  * this method is passed the answer which it displays
  */
  {
    System.out.println();
    System.out.println("The sum of the three numbers is "
      + answer);
    System.out.println();
  }
}
```

ReadSumNos2.java *part2*

This program defines a total of 4 methods - the main method and three other. The main method calls or invokes each of the other methods in turn (this does not necessarily have to be the case - sub-methods can themselves invoke other methods - more later). A method is called in one of two ways - by merely stating its name as in

```
DisplayHeading();
DisplayAnswer(total);
```

or by assigning the value it returns to a suitable variable

```
total = SumNos();
```

Methods are defined with a method header statement, eg.

```
static void DisplayHeading()
```

This defines that the method is `static` (at this stage all methods you use should be static), that it does not return a value but is called merely for its effect (`void`), the name of the method (`DisplayHeading`) and, in brackets, any values it will use during its execution - none in this case hence the empty brackets.

The next method header returns a value to the calling method - it functions to compute some value which it passes back to the place it was called.

```
static double SumNos()
```

The word `double` in the definition indicates that this method returns a value whose data type is `double`. This method is called by assigning it to a `double` variable (`total`).

The last method header does not return a value, but uses a value in order to perform its task. The value it needs to be passed is specified in the header together with the datatype:

```
static void DisplayAnswer(double answer)
```

In this case the method will use and refer to a `double` variable `answer` for which a value is supplied when the method is called. The call to this procedure is

```
DisplayAnswer(total);
```

so the value of `total` is supplied to `DisplayAnswer`.

Each of the sub-methods functions as a self-contained module and performs its own task, with the main method serving as controller. Execution of a program starts with the first statement of the main method which is the call to method `DisplayHeading`. Control is thus transferred to the first statement of `DisplayHeading` which is executed, and proceeds through each of the statements of `DisplayHeading` in turn. When the last statement has been executed control is transferred back to the main method and the next statement of `main` is executed.

This is the call to `SumNos`, so control is transferred to the first statement of `SumNos`, and each of its statements are executed. The last statement of `SumNos` is

```
return answer
```

which specifies the value to be returned by `SumNos`, and which is then stored in the variable `total` in the main method.

Lastly the third statement of the main method is executed and control is transferred to `DisplayAnswer` with the value of `total` being passed to `DisplayAnswer`'s parameter, `answer`. The statements of `DisplayAnswer` are executed, the `println` statement outputs a message and the value of `answer`, then control is transferred back to the main method. There are no more statements in the main method so execution of the program ends.

The effect of executing this program is exactly the same as the second example - in this case different values have been entered:

```
C:\Java 1.2\bin\java.exe   ReadSumNos2
Working Directory - C:\javaprogs\3SimpleProgs\

This program reads in 3 numbers and sums them
----------------------------------------------

  Enter the first number  > 7.3
  Enter the second number > 19
  Enter the third number  > 105

The sum of the three numbers is 131.3
```

Exercises

3.1 Compile and run the three programs.
Experiment with them - try causing errors (delete a semicolon, misspell a word etc) and see the errors that result.

In the following exercises you are asked to modify existing programs. In order to keep both the original and the new version, it is easiest to make a copy of the original with a new filename (remember to change the name of the main method too!), and then make the modifications to this copy.

3.2 Modify `Hello.java` so that it displays the following message.

```
************
*  I love  *
* COMPUTER *
* SCIENCE. *
************
```

3.3 Modify `ReadSumNos` so that it inputs and adds 4 numbers.

3.4 Modify `ReadSumNos` so that it inputs, calculates and displays the average of the 3 numbers. `(num1+num2+num3)/3`

3.5 Modify `ReadSumNos2` so that it uses methods to input 2 numbers and display their difference.

4. Using data - types and items

All programs manipulate data - often they read in some values, perform some calculations, output the results. This data may be of different types - for example, numeric or textual - and in order to be able to use these values they must be stored (usually) as variables in the computer's memory.

4.1 Simple Data Types

Java provides a number of simple data types - integer, floating point, character and boolean.

Integer data types are used to store whole-number (integer) quantities

Eg. 23 -7 0 999 +999

They are often used to count objects (eg. The number of students in a class, the number of lines of data etc). They consist of an optional sign followed by a number, with no decimal point.

There are in fact 4 integer data types, each of which use different amounts of memory to represent the integer value and store different ranges of numbers:

TYPE	STORAGE	MAX VALUE
`byte`	8 bits	255
`short`	16 bits	32767
`int`	**32 bits**	**2147483647**
`long`	64 bits	over 10^{18}

`int` is the type usually used for declaring integers.

Integer data is stored directly as a binary string in the specified number of bits using two's complement representation. Because there are a fixed number of bits available each type has a set range of values that can be stored - for `int`, the range of values is -2147483648 to +2147483647. If you try to use a value outside this range you may get an error, or worst of all, the number may "wrap round" and leave you with the wrong value without any indication that this has happened.

Floating point data types are used to represent real numbers (in the mathematical sense). Numbers are stored with a decimal point (not comma!) and a fractional part which may be zero. There must be a digit on either side of the decimal point, and a optional sign may be specified.

Eg. 1.25 -0.001 0.0 999.9 +66.6667

There are 2 floating point data types in Java which use different amounts of memory and hence are accurate to different numbers of decimal places:

TYPE	STORAGE	MAX VALUE	PRECISION
`float`	32	over 10^{38}	7 decimal digits
`double`	**64**	**over 10^{308}**	**15 decimal digits**

double is the type usually used for declaring floating point numbers.

Floating point numbers are stored with both a mantissa and an exponent. The mantissa is a binary fraction between 0.5 and 1.0; the exponent is a power of 2. The mantissa and exponent are chosen so that

$$number = mantissa \times 2^{exponent}$$

Because of the finite size of a memory cell not all fractions can be represented precisely. Thus the decimal fraction 0.1 might actually be represented within the computer's memory as 0.9999999… even though it may appear as 0.1 when displayed. For this reason it is not advisable to use floating point numbers for purposes such as counting or indexing where exact values are required.

The **character data type** is used to store a single character. This **`char`** type includes all the letters, digits and symbols that are available on a keyboard. Characters in Java are enclosed in a single quote

Eg. 'A' 'a' '+' '3' ' '

The character type `char` uses 16 bits to represent the character with a binary numeric code. Each of the codes is associated with a particular character according to the UNICODE coding system. When the computer recognises that it is dealing with character data, it merely interprets the binary number as a character code from this set instead of the numeric value. The UNICODE values from 0 to 255 are identical to the ASCII codes - further characters are used for additional symbols and letters in Greek, Cyrillic, Hebrew … etc alphabets.

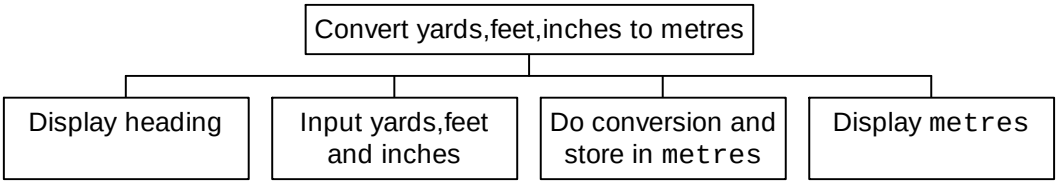
When dealing with character data types the actual code value is not of importance - what is significant is that the upper- and lower-case alphabetic letters and the digit characters are in continuous sequences - ie. a set of sequential codes is used for the letters 'A' … 'Z'; another set of sequential codes is used for the letters 'a' … 'z'; and another set for the digits '0' … '9'.

The **Boolean data type** has only two possible values - `false` or `true`. These are Java keywords and are used to reflect the result of a Boolean operation such as `num1<0`. Boolean expressions such as this govern the decisions made in programs as to which of alternative paths to follow, and the results of these Boolean expressions are of data type **`boolean`**.

4.2 Variables

When data items are used in a program, they are (usually) stored in variables. Variables are areas of memory that have a name or identifier associated with them, and can store values. The amount of memory allocated for a particular variable depends on the type of data that it will store. A variable to store a value of type `int` will be allocated 4 bytes, a variable to store a value of type `double` will be allocated 8 bytes.

For example, the following program inputs a distance in yards, feet and inches and outputs the corresponding distance in metres. Variables are used in this program to store the values for yards, feet and inches as they are input, to store the intermediate results of the total number of inches and the conversion to centimetres, and the final result in metres which is then displayed. All the variables are in bold in the text.




```
/*
* Reads a distance in yard, feet, inches and
* converts to metres
* ----------------------------------------
*/
import Utilities.Keyboard;

public class ConvertToMetres
{
  public static void main(String[] args)
  {
    int yard,feet,inches;   // the 3 distance values
    int totInch;        // the total number of inches
    double totCm;           // the distance in centimetres
    double metres;      // the final result in metres

// display a heading
    System.out.println();
    System.out.println("This program converts yards,feet,"
    + "inches to metres");
    System.out.println("--------------------------------"
      + "----------------");
    System.out.println();

// input distance in yards, feet and inches
    System.out.print("  Enter yards  > ");
    yard = Keyboard.getInt();
    System.out.print("  Enter feet   > ");
    feet = Keyboard.getInt();
    System.out.print("  Enter inches > ");
    inches = Keyboard.getInt();
    System.out.println();

// do the conversion
    totInch = ((yard*3)+ feet)*12+inches;
                // 1 yard = 3 feet, 1 foot = 12 inches
    totCm = totInch*2.54;           // 1 inch = 2.54 cm
    metres = totCm/100;

// display the results
    System.out.println("The distance in metres is " + metres);
    System.out.println();
  }
}
```

ConvertToMetres.java

In order to declare a variable both its name and data type must be defined. Examples of variable declarations (with explanatory comment) are

```
int yard,feet,inches;       // the 3 distance values
int totInch;                // the total number of inches
double totCm;               // the distance in centimetres
double metres;              // the final result in metres
```

Variables must be declared before they can be used, and they are usually declared together at the beginning of the method or block in which they are to be used. They should be declared in the method where they are needed - if a certain variable is needed in a method it should be declared in that method and not just at the beginning of the class that contains the method.

In computer languages, the names you choose for variables, classes, methods etc are called identifiers. An **identifier** in Java

- may consist of any combination of letters, digits, and the underscore character (_)

`total, X, n, no1, numStudents, num_students` are all valid identifiers

- must start with a letter;

A23 is valid, 23A is not.

- spaces are not allowed;

`my_name` is valid, `my name` is not

- upper- and lower-case letters are different;
  `TotalAmount`, `totalamount` and `totalAmount` are all distinct identifiers.

It is important that the identifier chosen for a variable is meaningful and conveys a sense of what the variable represents. This makes programs easier to read and understand, and assists in debugging and maintenance of programs. Generally, lowercase is used for variable names, with names that consist of more than one word having capitals for the inner words to make it easier to read, for example `numOfValues`. Identifiers used as class names and method names generally have initial capitals, for example `MyFirstProgram` or `DisplayResults`.

To summarise - the form of a variable declaration is one of

VARIABLE DECLARATION
type name;
type name1,name2,name3;
type name = value;

The declaration introduces one (*first form*) or more (*second form*) variables of the given type. The last form can be used to initialise a variable at the same time as it is declared.

More examples of variable declarations are

```
double   weight;         // in kilograms
int      age;            // in whole years
int      day,month,year;// date of birth
double   vat = 14.0;         // % VAT
double   price,          // in Rand
         cost;           // total amount paid
```

It is a good idea to keep declarations neat and tidy, with identifiers and types lined up. It is also a useful habit to indicate what the variables are to be used for if this is not immediately obvious from their names, as well as to give some supporting information (such as units) if appropriate.

4.3 Constants

Some values used in a program are fixed and can be written as literal (or constant) numeric values in the program - eg. the conversion factor for inches to cm (2.54) in the previous example program.

```
totCm = totInch*2.54;
```

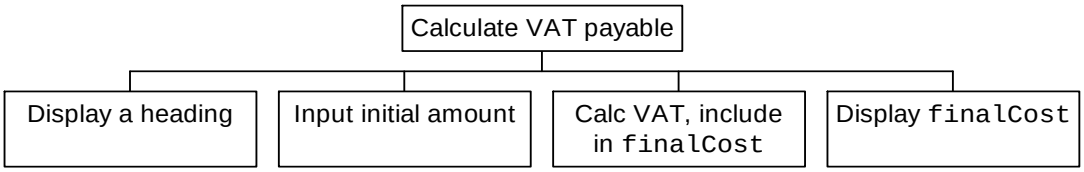
It is often better to give names to these fixed values as it makes them easier to remember and use, particularly if they appear more than once in a program. In Java, named data items whose values are not going to change (in the program) are known as constants, and are declared as

```
static final double inToCm = 2.54;
static final double vatRate = 14.0;
static final int numStudents = 120;
```

CONSTANT DECLARATION
`static final` *`type name = value`*`;`

The modifier `static` means that it applies to the entire class, and `final` indicates that its values cannot be changed during the program - it is constant. Because it is a class field, a constant declaration must appear at class level and may not be inside any method, even `main`.

The advantage in using constants is twofold: firstly statements that use constants are generally easier to understand because descriptive names are used rather than numbers which have no intrinsic meaning (`numStudents` is more meaningful than 120 which might refer to anything, even the speed limit), and secondly if the value the constant represents does change (for example VAT increases to 16%), it is easier to merely change the value of the constant than finding every time 14.0 appears in the program representing VAT and changing it to 16.0.




```
/*
* Reads an amount and determines how much VAT must be added
* -------------------------------------------------------
*/
import Utilities.Keyboard;

public class CalcVAT
{
  static final double vatRate = 14.0;

  public static void main(String[] args)
  {
    double   basicCost,     // the initial amount
             VAT,       // the VAT on this amount
             finalCost;     // the final cost with VAT

// display a heading
    System.out.println();
    System.out.println("This program calculates the VAT"
      + " payable");
    System.out.println("-------------------------------"
      + "--------");
    System.out.println();

// input the initial amount
    System.out.print("  Enter initial amount > R ");
    basicCost = Keyboard.getDouble();
    System.out.println();

// calculate the VAT and the final amount
    VAT = basicCost * vatRate/100.0;
    finalCost = basicCost + VAT;

// display the results
    System.out.println("  VAT rate is " + vatRate + "%");
    System.out.println("  The final amount inclusive of "
      + "VAT is R" + finalCost);
    System.out.println();
  }
}
```

CalcVAT.java

In this example, vatRate is declared as a constant and assigned the value 14.0 just inside the class CalcVAT, and it is then used twice in the program - to calculate the VAT payable and when displaying the VAT rate used.

4.4 Assignment

Once a variable has been declared, it can be given a value in an assignment statement, eg.

```
totCm = totInch*2.54;
```

The form of the assignment statement is

ASSIGNMENT STATEMENT
variable = expression;

where an expression can be a literal value

```
myAge = 21;
```

or the value of another variable of the same type

```
hisSalary = herSalary;
```

or the result of an arithmetic calculation

```
average = (num1+num2+num3)/3;
```

or the result of a call to an input method

```
basicCost = Keyboard.getDouble();
```

or the result of a call to any appropriate method

```
total = SumNos();
```

An assignment statement is used to store a value in the memory location referenced by the variable name. It replaces any value the variable might previously have had. The assignment statement is read as "*myAge is assigned the value 21*" or "*myAge becomes 21*".

The data types of the variable and the value assigned to it must be compatible. We must assign `int` to `int`, `double` to `double`, and we cannot assign `double` to `int`. However, the one exception is that we can assign `int` to `double`, because real numbers include the integers. So an assignment statement of the form

```
numStudents = 105;
```

is perfectly acceptable, as is

```
int randValue;
double cost;
randValue = 13;
cost = randvalue;
```

in fact, assignments across the numeric types in order of size are valid:

`byte` → `short` → `int` → `long` → `float` → `double`

so you can assign `byte` to `int` or `long` to `float` etc.

If you need to assign a longer data type to a shorter one you must use type casting, and explicitly specify that you are converting to the new type by putting the required type in brackets .

TYPE CAST
(type) expression;

Eg. `int randValue;`

```
double cost;
cost = Keyboard.getDouble();
randValue = (int) cost;
```

This will truncate the floating point value, `cost`, to an integer by truncation - ie. the decimal fraction will be dropped and the integer portion will be stored in `randValue`.

if `cost` is 16.99, `randValue` will be assigned the integer 16.

If `cost` is 16.35, `randValue` will be assigned 16.

If you wish to round a floating point number to the nearest integer, then the method `Math.round` is provided by Java for this purpose:

```
randValue = Math.round(cost);
```

In this case, if `cost` is 16.99, `randValue` will be assigned 17,
if `cost` is 16.35, `randValue` will be assigned 16.

`Math.round` will round a `double` to a `long`, or a `float` to an `int` (same sized data types). To round a `double` to an `int` you need to use type casting

```
randValue = (int)Math.round(cost);
```

If you are using type casting to convert, for example, `int` to `byte`, you must make sure that the value stored in the `int` is not too large to fit into the `byte`.

Eg.
```
int bigClass = 500;
byte numStudents;
numStudents = (byte) bigClass;
```
will return a syntax error because the largest value that can be stored in a variable of type `byte` is 255.

4.5 Arithmetic expressions

The arithmetic operators that are used in Java expressions are

`+`	addition	`sum = num1 + num2;`
`-`	subtraction	`diff = num1 - num2;`
`*`	multiplication	`product = num1 * num2;`
`/`	division	`quotient = num1 / num2;`

If all the operands are integer types, the result will be type `int`; if some of the operands are floating point types the result will be type `double`. (Which is why the usual data types used are `int` and `double` - to avoid having to do numerous type casts).

The division operator `/` behaves differently depending on whether the operands are integer or floating point. For floating point operands, `/` performs real division but for integer operands, `/` performs integer division, even if the variable in which the result is to be stored is floating point.

```
double realAnswer;
int intAnswer;
realAnswer = 7.5 / 2.5;        // will store 3.0
intAnswer = 5 / 2;             // will store 2
realAnswer = 5 / 2;            // will store 2.0 (integer division)
realAnswer = 5.0 / 2;          // will store 2.5
intAnswer = 7.5 / 2.5;         // will give a syntax error
intAnswer = (int) 7.5 / 2.5    // will store 3
```

There is an additional operator in Java (as in many other programming languages), and that is the modulus operator, `%`, which returns the remainder after integer division.

```
int intAnswer, remainder;
intAnswer = 7 / 3;             // will store 2
remainder = 7 % 3;             // will store 1
```

Brackets may be used in arithmetic expressions, and expressions in brackets are evaluated first. The order of evaluation is the familiar

brackets first; then `* / %`; finally `+ -`

Within precedence levels, evaluation takes place from left to right, and operators act only on values immediately adjacent to them. Thus `a/b*c` would be evaluated as `(a/b)*c`, and if $\frac{a}{bc}$ is intended, it should be written as `a/(b*c)`.

Multiplication is written explicitly, and the multiplication operator `*` must be used. Implicit multiplication of the form `2a` or `5(x+y)` is wrong and should be written as `2*a` or `5*(x+y)`.

For example:

(a+b)(a-b)	`(a+b)*(a-b)`
$\frac{5x+3y}{xy}$	`(5*x+3*y)/(x*y)`

to calculate the gradient of the line between 2 points (x1,y1) and (x2,y2):

$\frac{y2-y1}{x2-x1}$	`gradient = (y2-y1)/(x2-x1)`

to convert degrees Celsius to Fahrenheit:

$degC \times \frac{9}{5} + 32$	`degF = degC*5/9 + 32;`

to convert degrees Fahrenheit to Celsius:

$\frac{5}{9}(degF - 32)$	`degC = (degF-32)*5/9;`

Other useful functions are provided by Java in the `Math` class

x^2	`Math.pow(x,2)`	*will raise the first argument to any power*
$\sqrt{x}$	`Math.sqrt(x)`	*will calculate the square root of the argument*
sin x	`Math.sin(x)`	
abs(x)	`Math.abs(x)`	*will return the positive value of the argument*

In addition, the Math class provides a useful `double` constant

π	`Math.PI`	*the closest double to pi*

To calculate the area of a circle of radius r:

πr^2 `area = Math.PI*r*r`

or `area = Math.Pi*Math.pow(r,2)`

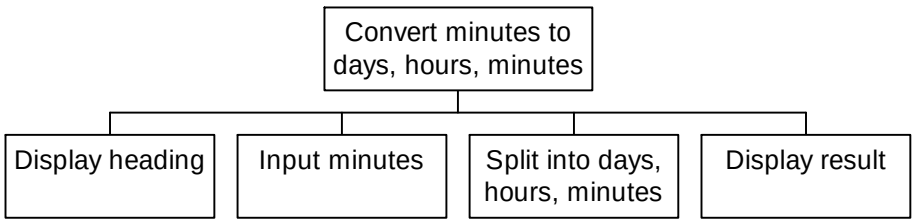
To calculate the distance between two points (x1,y1) and (x2,y2):

$$\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

```
dist = Math.sqrt(Math.pow(x2-x1,2)+Math.pow(y2-y1,2))
```
or
```
dist = Math.sqrt((x2-x1)*x2-x1)+(y2-y1)*(y2-y1))
```

4.6 Complete examples

Write a program to determine how many days, hours and minutes there are in a given number of minutes.



The interesting part is how to divide the minutes into days, hours and minutes. We know there are 60 minutes in an hour, so if we divide the total minutes by 60 we should get the number of hours, and the remainder will give us the number of minutes left. Then we can divide the hours by 24 to get the number of days, and the reminder gives the number of hours left.

```
/*
* Reads a time in minutes and splits it into days, hours
* and minutes
* -------------------------------------------------------
*/
import Utilities.Keyboard;

public class SplitMinutes
{
  static final int hoursInDay = 24;      // constants
  static final int minsInHour = 60;

  public static void main(String[] args)
  {
    int totMins;        // the total minutes as input
    int totHrs;         // total hours - used to convert
    int days,hours,mins;   // the 3 variables for the
                        // split up time
// display a heading
    ...   < code omitted >
```

```
// input total minutes
    System.out.print("Enter total minutes  > ");
    totMins = Keyboard.getInt();

// do the conversion
    totHrs = totMins / minsInHour;
    mins = totMins % minsInHour;
    days = totHrs / hoursInDay;
    hours = totHrs % hoursInDay;

// display the result
    System.out.println(totMins + " minutes is " + days
      + " days, " + hours + " hours and " + mins
      + " minutes");
    System.out.println();
  }
}
```

SplitMinutes.java

Consider the values of the variables during execution of this program:

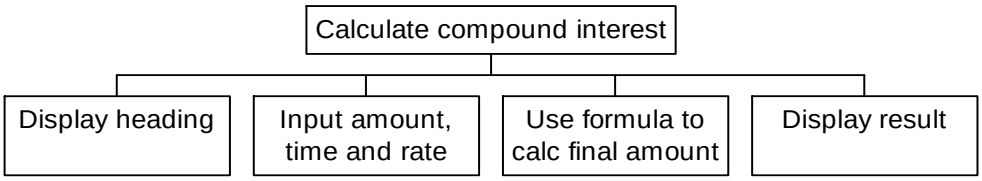
	totMins	totHrs	days	hours	mins
initially	0	0	0	0	0
`totMins = Keyboard.getInt();`	2000	0	0	0	0
`totHrs = totMins / minsInHour;`	2000	33	0	0	0
`mins = totMins % minsInHour;`	2000	33	0	0	20
`days = totHrs / hoursInDay;`	2000	33	1	0	20
`hours = totHrs % hoursInDay;`	2000	33	1	9	20

As expected, the result of executing the program with the input value 2000 is

```
C:\Java 1.2\bin\java.exe   SplitMinutes
Working Directory - C:\javaprogs\4DataTypes\

This program splits a time in minutes into days, hours and
minutes
------------------------------------------------------------

Enter total minutes  > 2000
2000 minutes is 1 days, 9 hours and 20 minutes
```

Write a program to calculate the result of investing a sum of money (Amt) at a given interest rate (r%) for a number of years (P) - for example, R1000 for 10 years at 15% interest pa.

The compound interest formula is $\text{Amt} \times \left(\frac{100 + r}{100}\right)^{P}$




```
/*
* Calculates compound interest - determines final amount
* after investing an amount for a number years at R%
* -------------------------------------------------
*/
import Utilities.Keyboard;

public class CompInterest
{
  public static void main(String[] args)
  {
    double   amount,        // the initial amount invested
             years,     // the number of years invested
             rate,      // the annual interest rate %
             finalAmt;      // the final amount

// display a heading
    System.out.println("Compound interest calculation");
    System.out.println("-----------------------------");
    System.out.println();

// input amount, years and interest rate
    System.out.print("  Enter initial amount    > R");
    amount = Keyboard.getDouble();
    System.out.print("  Enter years invested    > ");
    years = Keyboard.getDouble();
    System.out.print("  Enter interest rate (%) > ");
    rate = Keyboard.getDouble();
    System.out.println();

// calculate the final amount
    finalAmt = amount * Math.pow((100+rate)/100,years);

// display the result
    System.out.println("R" + amount + " invested for "
      + years + " years at " + rate + "% interest");
    System.out.println("yields R" + finalAmt);
    System.out.println();
  }
}
```

CompInterest.java

When this program is run with the values R1000, 10 years and 15% interest, the output is somewhat surprising:

```
C:\Java 1.2\bin\java.exe   CompInterest
Working Directory - C:\javaprogs\4DataTypes\

Compound interest calculation
-----------------------------

  Enter initial amount    > R1000
  Enter years invested    > 10
  Enter interest rate (%) > 15

R1000.0 invested for 10.0 years at 15.0% interest
yields R4045.557735707907
```

It would be nice if we could control the number of decimal places that are displayed so that the normal Rand and cents are shown. This will be discussed in the next chapter.

Exercises

4.1 Modify the `SplitMinutes` program to convert a number of seconds to weeks, days, hours, minutes and seconds. Test your program with 1000000 seconds.

4.2 Modify the `CompInterest` program to compound the interest monthly instead of annually. (Multiply the years by 12 to get months; divide the interest rate by 12 to get a monthly rate.)

4.3 Write a program that will calculate and display how many times a human heart will beat in a given lifetime if it beats once a second. (Assume 365.25 days a year). Input the number of years in a lifetime.

4.4 Write a program to read in a radius r and to calculate and display the

area of the circle	πr^2
circumference of the circle	$2\pi r$
volume of the sphere	$\frac{4}{3}\pi r^2$
surface area of the sphere	$4\pi r^2$

4.5 Write a program to calculate the distance between 2 points (x1,y1) and (x2,y2)

distance = $\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$

4.6 Write a program which, given 2 points (x1,y1) and (x2,y2), displays the equation of the line $y = mx + c$ passing though the 2 points, where

the gradient $m = \frac{y2 - y1}{x2 - x1}$ and the constant $c = y1 - m \times x1$

5. Output and Input

In the programs written so far, we've used output and input statements without really understanding what they actually do. This chapter considers them in more detail.

5.1 Output

All output (or display or printing - the terms are often used interchangeably) is handled by methods supplied as part of the Java language. The two methods used are

`System.out.print` and `System.out.println`

In both cases, `System` refers to the universally available class `System`, which contains an object called `out` that is automatically connected by Java to the screen of your computer. A call to the methods `print` or `println` (which we'll informally refer to as output statements) will display the specified data items on the screen of your computer.

OUTPUT STATEMENTS
`System.out.println(`*`items`*`);`
`System.out.println();`
`System.out.print(`*`items`*`);`

The difference between `print` and `println` is that `println` always ends the line so that the next output will appear on a new line, whereas `print` does not end the line so the next output will continue on the same line.

Consider the following program:

```
/*
* Output demonstration
* --------------------
*/

public class OutputDemo1
{
  public static void main(String[] args)
  {
    System.out.println("This message appears on one line.");
    System.out.println("This one appears on the next line.");
    System.out.println();
    System.out.print("A blank line is left before this ");
    System.out.print("message displayed on one line.");
    System.out.println();
    System.out.println("--- End of output example ---");
  }
}
```

OutputDemo1.java

The output produced when it is run is

```
C:\Java 1.2\bin\java.exe   OutputDemo1
Working Directory - C:\javaprogs\5OutputInput\
This message appears on one line.
This one appears on the next line.

A blank line is left before this message displayed on one line.
--- End of output example ---
Process Exit...
```

The first output statement

```
System.out.println("This message appears on one line.");
```

displays the first line and ends that output line, so the second output statement

```
System.out.println("This one appears on the next line.");
```

is displayed on the second line. This output line is also ended, so the next statement

```
System.out.println();
```

displays on a new line, but because no data items are specified in the brackets, nothing is displayed and the effect is to leave a blank line. (The empty brackets are required.)

The 4th output statement

```
System.out.print("A blank line is left before this ");
```

calls the other method, `print`, which does not end the line output so the contents of the 5th statement continue on the same output line. Note the space at the end of the 4th statement. If it was missing,

```
System.out.print("A blank line is left before this");
System.out.print("message displayed on one line.")
```

the output would appear as

```
A blank line is left before thismessage displayed on one line.
```

The 6th output statement

```
System.out.println();
```

displays nothing and ends the line. However, because it is preceded by a call to `print` it does not have the effect of leaving a blank line but instead merely ends the current line.

Because the output produced by the program is squashed up between Java's messages, I prefer to leave a blank line at the beginning and end of a program, or to demarcate the output in other ways such as shown in the next example.

The data items in these output statements are all **string literals**, which is any sequence of characters enclosed in quotes, such as

```
"Hello"
"75.3%"
"A.N. Other"
```

To display a quote itself it must be preceded by a backslash `\` which is an escape character - it does not form part of the string but enables the next character to do so.
For example,

```
System.out.println("My name is \"Jane\".");
```

would display

```
My name is "Jane".
```

Another useful escape character is `\n` which represents newline and can be used to end a line or go to a new line in the middle of a string. For example,

```
System.out.print("Hello\nthere\neveryone");
```
would display
```
Hello
there
everyone
```
and
```
System.out.print("\n\n\n");
```
would leave 3 blank lines.

A string must all be on one line, so if it is too long to fit on a single line it must be split into 2 strings which are then joined with a plus + (known as concatenation). For example,
```
System.out.println("This program was written by "
  + "Jane Meyerowitz.");
```
would display
```
This program was written by Jane Meyerowitz.
```
Again, note the space at the end of the string (.. by ") so that the final spacing is correct.

Numeric values are also output using the print and println methods. For example
```
System.out.println(2*3+10);
```
would display
```
16
```

To combine strings and numeric values as output, concatenation (+) is used.
```
System.out.println("The answer is " + (2*3+10));
```
will output
```
The answer is 16
```

Similarly, numeric variables can be output. The program fragment
```
int num1 = 23;
int num2 = 14;
int answer = num1+num2;
System.out.println("The sum of " + num1 + " and " + num2
  + " is " + answer);
```
will display
```
The sum of 23 and 14 is 37
```

Again, note the use of spaces at the ends of the strings. If they are omitted
```
System.out.println("The sum of" + num1 + "and" + num2
  + "is" + answer);
```
the output is
```
The sum of23and14is37
```

Recall the program `CompInterest.java` at the end of the last chapter which calculated compound interest. The output statements
```
System.out.println("R" + amount + " invested for "
  + years + " years at " + rate + "% interest");
System.out.println("yields R" + finalAmt);
```
display
```
R1000.0 invested for 10.0 years at 15.0% interest
yields R4045.557735707907
```
The `print` and `println` methods expect the data items in the parentheses to be strings. If a numeric value is included Java automatically calls its `toString` method to convert the numeric value to a string which is displayed by the `print` method. In the case of floating

point numbers `toString` has no information on how many decimal places are to be displayed, so it displays them all.

The manner in which you define the formatting of output in Java is complex, so a package and methods have been provided for you to use which simplifies the formatting and layout of numeric values. The package `Utilities` provides a class `Formatter` which has a method `format` that will output numeric values in a specified number of positions, and with a specified number of decimal places for floating point numbers.

FORMATTING
`Formatter.format(`*`fpvalue`*`,`*`width`*`,`*`decimals`*`);`
`Formatter.format(`*`intvalue`*`,`*`width`*`);`

where *`fpvalue`* is a `float` or `double` value (variable, constant or expression)
`intvalue` is an `int` or `long` value (variable, constant or expression)
`width` is the total number of positions the value must be displayed in
`decimals` is the number of decimal places to be displayed

Consider the following demonstration program:

```
/*
* Formatter demonstration
* -----------------------
*/
import Utilities.Formatter;

public class OutputDemo2
{
  public static void main(String[] args)
  {
    double num1 = 1234.5678;
    int num2 = 12345;

    System.out.println("\n---------------------------------");
    System.out.println("Double formatting");
    System.out.println("  num1 = "+Formatter.format(num1,9,4));
    System.out.println("  num1 = "+Formatter.format(num1,7,2));
    System.out.println("  num1 = "+Formatter.format(num1,7,1));
    System.out.println();
    System.out.println("Integer formatting");
    System.out.println("  num2 = "+Formatter.format(num2,5));
    System.out.println("  num2 = "+Formatter.format(num2,6));
    System.out.println("  num2 = "+Formatter.format(num2,3));
    System.out.println();
    System.out.println("---------------------------------");
  }
}
```

OutputDemo2.java

Most importantly, in order to use this class it needs to be imported as it is not automatically imported by Java. This is done right at the beginning of the program, after the initial comments but before the class definition:

```
import Utilities.Formatter;
```

This instructs Java to permit the use of any methods in the `Formatter` class of the `Utilities` package. Should you forget to include this import statement Java does not recognise the reference to the class and you will get an error message for each time it is used in the program. Compiling this program without the import statement resulted in 6 error messages of the type

```
OutputDemo2.java:17: Undefined variable or class name: Formatter
    System.out.println("  num1 = " + Formatter.format(num1,9,4));
                                    ^
```

After correction, the output displayed by this program is

```
C:\Java 1.2\bin\java.exe   OutputDemo2
Working Directory - C:\javaprogs\5OutputInput\

---------------------------------
Double formatting
  num1 = 1234.5678
  num1 = 1234.57
  num1 =  1234.6

Integer formatting
  num2 = 12345
  num2 =  12345
  num2 = 12345

---------------------------------
Process Exit...
```

Comparing the output produced with the formatting used:

`Formatter.format(num1,9,4)` displays `1234.5678`
a total of 9 positions with 4 decimals

`Formatter.format(num1,7,2)` displays `1234.57`
a total of 7 positions with the decimal rounded to 2 places

`Formatter.format(num1,7,1)` displays `1234.6`
a total of 7 positions with the decimal rounded to 1 place, and because only 6 positions are needed there is a space before the number.

`Formatter.format(num2,5)` displays `12345` using 5 positions

`Formatter.format(num2,6)` displays `12345` using 6 positions
with a space before the number because only 5 positions are needed

`Formatter.format(num2,3)` displays `12345` using 5 positions
because the width specified (3) is too small to display the number
it is disregarded and the number is displayed with all its digits

To summarise, if the width specified is larger than needed, the number it is right justified in the field with spaces before it. If the width specified is smaller than is needed it is disregarded and the number is displayed in the number of digits required. The decimals are always displayed using the number of positions given, rounded if necessary.

Generally, the *width* specifier is used to tabulate or line up columns of data, and `Formatter` is not usually used for integer data unless this tabulation is required. With floating point values however, we often want to specify the number of decimal places but don't know the total field size required, and to define too large a width "just in case" means there are blank spaces in the output. In this case it is convenient to specify a width of 1 and the required number of decimals, which means the value will be displayed in the exact number of digits needed and the correct number of decimal places.

The output statements from the compound interest program (`CompInt2.java`) should be formatted as

```
System.out.println("R"+Formatter.format(amount,1,2)+
   " invested for "+years+" years at "+rate+"% interest");
System.out.println("yields R"+Formatter.format(finalAmt,1,2));
```

and will then display the rand values correctly

```
R1000.00 invested for 10.0 years at 15.0% interest
yields R4045.56
```

As an example of tabulation, the program `OutputDemo3.java` calculates and displays the square root and the cube root of some integers. The results are output twice, first with the integer value (`num1`) unformatted

```
System.out.println(num1+ Formatter.format(Math.sqrt(num1),10,3)
  + Formatter.format(Math.pow(num1,cube),12,5));
```

and secondly with the integer value (`num1`) formatted.

```
System.out.println(Formatter.format(num1,3)
  + Formatter.format(Math.sqrt(num1),10,3)
  + Formatter.format(Math.pow(num1,cube),12,5));
```

The effect of formatting to tabulate data is clearly seen in the output

```
Tabulating columns of data
--------------------------

  Integer not formatted
num   sq.root   cube root
---   -------   ---------
1     1.000     1.00000
10     3.162     2.15443
100    10.000     4.64159

    Integer formatted
num   sq.root   cube root
---   -------   ---------
  1     1.000     1.00000
 10     3.162     2.15443
100    10.000     4.64159
```

5.2 Input

As has already been mentioned, reading data in from the keyboard to use in a program is very cumbersome in Java. In order to simplify matters, a package and methods have been provided for you to use (as with formatting) to allow for the input of numeric values. The package `Utilities` provides another class `Keyboard` which has different methods for inputting different types of data.

KEYBOARD INPUT
intVar = Keyboard.getInt();
longVar = Keyboard.getLong();
floatVar = Keyboard.getFloat();
doubleVar = Keyboard.getDouble();
charVar = Keyboard.getChar();

The method `getInt` will accept an `int` value and assign it to the `int` variable specified; similarly for the methods `getLong` (`long` values), `getFloat` (`float` values) and `getDouble` (`double` values). The method `getChar` accepts a single character from the keyboard and assigns it to the `char` variable specified.

The ENTER key must be pressed after the value has been typed to send it to the program. Until this has happened the value typed can be changed - characters can be deleted and modified - and then ENTER is pressed to send the value for processing.

In order to use these methods, the Keyboard class must be imported

```
import Utilities.Keyboard;
```

If you're also using Formatter to format your output, you import them both with 2 statements

```
import Utilities.Keyboard;
import Utilities.Formatter;
```

or alternatively you can use an asterisk (`*`) as a "wild card" to import all classes in the `Utilities` package which will import both `Keyboard` and `Formatter`.

```
import Utilities.*;
```

When one of the Keyboard methods is called the program pauses and waits for the user to enter some data and press the ENTER key. The data is then input, and if there is no error the value is assigned to the specified variable. For numeric input, runtime errors occur

- if a floating point value is entered when calling `getInt` or `getLong` (however integer values may be entered for `getFloat` and `getDouble`)
- if a value that is too large for the data type is entered
- if an alphabetic or special character is entered
- if more than one value is entered on a line

Any single character is accepted for `getChar`, and its UNICODE character code is stored in the char variable specified. Note that only a single character can be entered and the ENTER key must be pressed after that character is entered. (To input a word strings must be used - more complex so discussed much later)

As an example of character input, consider a program that inputs a character and then displays a heart made up of that character:

```
/*
* Character input demonstration
* ----------------------------
*/
import Utilities.Keyboard;

public class CharInputDemo
{
  public static void main(String[] args)
  {
    char c;    // the character to be used in drawing a heart

  // Input the character to use
    System.out.println();
    System.out.print("Enter the character to be used to "
      + "draw a heart > ");
    c = Keyboard.getChar();
    System.out.println();

  // Draw the heart
    System.out.println("    " +c+c+ "   " +c+c  );
    System.out.println("   "+c+c+c+c+" "+c+c+c+c);
    System.out.println("   " +c+c+c+c+c+c+c+c+c );
    System.out.println("    " +c+c+c+c+c+c+c    );
    System.out.println("     "  +c+c+c+c+c      );
    System.out.println("      "   +c+c+c        );
    System.out.println("       "    +c          );

    System.out.println();
  }
}
```

CharInputDemo.java

When this program is run, it pauses at the `Keyboard.getChar` statement and waits for the user to enter a character. The user types the character (say **O**) and the program completes its execution:

```
Enter the character to be used to draw a heart > O

    OO   OO
   OOOO OOOO
   OOOOOOOOO
    OOOOOOO
     OOOOO
      OOO
       O
```

If a different character is entered, a different heart is drawn:

```
Enter the character to be used to draw a heart > #

    ##   ##
   #### ####
   #########
    #######
     #####
      ###
       #
```

One comment about this program - spaces have been left at the beginning of each line so that the heart appears slightly indented and not at the extreme left hand edge of the window. Also, the layout of the data items in the brackets of the `println` statements was purely to make them look better for the human reader of the program and have nothing to do with the layout of the output produced. The statements below would display the identical output.

```
// Draw the heart
  System.out.println("    "+c+c+"   "+c+c);
  System.out.println("   "+c+c+c+c+" "+c+c+c+c);
  System.out.println("   "+c+c+c+c+c+c+c+c+c);
  System.out.println("    "+c+c+c+c+c+c+c);
  System.out.println("     "+c+c+c+c+c);
  System.out.println("      "+c+c+c);
  System.out.println("       "+c);
```

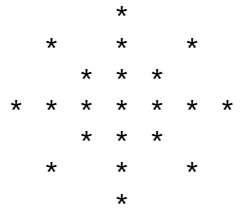
Exercises

For these exercises write clear, well commented programs that read in the required data with suitable prompts, perform the calculations and display the results. Ensure your output looks good. In all cases test the programs thoroughly by running them with different sets of data for which you have calculated the answers so that you can check whether they work correctly.

5.1 In Computer Science, your final mark is calculated from your assignment marks, your test marks and the exam mark in the ratio: assignments 10%, tests 15%, exam 75%. Write a program that reads in the total percentages for assignments, tests and the exam and calculates the final percentage mark obtained.

5.2 At the beginning of a journey the reading on a car's odometer is S kilometers and the fuel tank is full. After the journey the reading is F kilometers and it takes L litres to fill the tank. Write a program that reads in S, F and L and outputs the fuel consumption in km per litre and in litres per 100km correct to 4 decimal places.

5.3 Write a program to draw a snowflake. Input the character to use to form the snowflake.



5.4 Write a program to draw a house. Input the characters to use to draw the roof and the walls.

```
     +++++
    ++++++++
  ++++++++++
   000000000
   000000000
   000000000
```

5.5 A farmer has a rectangular field that is L metres long and W metres wide. He grows maize in this field, and his costs (seed, fertilizer, labour) are Rand C per square metre. He gets a yield of Y kg of maize per square metre. If maize is currently selling for Rand M per tonne (1000 kg), what profit does he get from this field?

Write a program that reads in values for L, W, Y, C and M and determines his profit.

5.6 The well known formula to calculate the roots of the quadratic equation

$ax^2 + bx + c = 0$

is

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

Write a program which reads in values for the coefficients a, b and c and calculates the two roots. Test your program by calculating the roots of

$x^2 - 5x + 6 = 0$	(2 and 3)
$2x^2 - 3x - 5 = 0$	(2.5 and -1)
$x^2 - 10x + 25 = 0$	(5 and 5)

5.7 In order to pay off a mortgage of \$P in N years on which interest is charged and computed annually at R% pa, \$A must be repaid every year, where

$$A = \frac{Pr(1+r)^N}{(1+r)^N - 1} \quad \text{and} \quad r = \frac{R}{100}$$

Write a program that reads in values for the amount borrowed P, the years N and the interest rate R and calculates the annual repayments A

5.8 Sue has bought a new car that can go 10% faster than her old one. Previously, on a trip to her parents home in Pofadder she would leave home at some time (eg 06.00) and arrive at her parents house some hours later (eg 15.35). If she leaves home at the same time driving her new car, what time will she now arrive at her parents home? Your program must read in the departure and arrival times using the 24hour clock (0600 and 1535).

6. Structure and Methods

There is more to programming than just writing code that achieves a particular purpose. It is also important that the program itself is "good". The structure of a program defines how the different parts of a program interact, and it is important that in addition to a program doing what it is intended to do, it is well structured. The program should be correct, readable, reusable and efficient.

6.1 Properties of a good program

Correctness

It seems a case of stating the obvious to say that a program must be correct, but in fact this is one of the most difficult tasks - to ensure that a program does exactly what is intended, and that it is bug free. The major portion of software costs are people costs - the salaries of programmers hired to write programs and debug, maintain or modify them. If by good programming practice the programmer's task can be made easier the benefits are significant.

There are three steps in ensuring correct programs

- understand the problem to ensure that you design a program that solves the problem and does what it is supposed to do, not what you would like it to do or is easier to do.
- follow good programming practice while writing a program - planning, structure, clarity, readability. It makes a program easier to understand, debug and modify.
- test the program thoroughly once it is working. Some simple guidelines - run the program with cases for which you know the outcome so that you can check the result is correct. Don't just choose the obvious cases - try input from a range of possible values (zero, positive and negative, large and small, integer and real, as appropriate)

Readability

A program has often to be understood by many people, not just the developer, and in order to be able to understand and reason about a program we must be able to read it. Factors to consider are

- use of comments to explain the meaning of the different parts and their interrelationship
- good, consistent layout, with careful use of indenting and blank lines
- meaningful identifiers - `numStudents, examMark` instead of `S` and `em` or worse still `Spot` and `Tinker` (your dogs names!)

It is also helpful to develop a style convention - be consistent in your use of capitalisation, always declare your variables at the beginning of the program, group declarations together, have opening and closing scope brackets in consistent places (after a statement, at the beginning of the next line) etc. Companies may have their own style, layout and documentation conventions and you will be expected to follow them.

Reusability
As mentioned earlier, a large amount of time and money is spent on developing software, so its a good idea where possible to reuse software that has already been written and tested by someone else. Java's packages are designed for just that - instead of each programmer writing a class to deal with (eg.) times and dates, one is provided. There are two aspects to reusability

- using or adapting existing classes in a program;
- writing our own classes so they are as general as possible without compromising readability or efficiency.

Efficiency
Programs should be written thriftily, to achieve their task simply and directly and be as clear and concise as possible. Unnecessary complications should be avoided, as should "clever tricks" that may implement an algorithm slightly more efficiently but are obscure, difficult to understand and nearly impossible to debug. Some suggestions are

- declare variables in the methods where they are needed, don't declare a whole lot up front in the main method "just in case".
- use methods to carry out common or similar tasks to avoid unnecessary duplicate statements
- if a calculation or portion of a calculation is to be used more than once, don't recalculate each time - do the calculation once and store the result.

6.2 Methods

In one of our early examples in chapter 3 - Simple Programs, methods were used to structure a program by writing it using a number of subprograms or modules which each achieved a particular sub-goal of the overall program. These modules or methods were controlled by the main method which called the modules in the correct order and controlled the flow of data between them, so that the overall goal was achieved.

You've also used methods supplied by the Java language (eg. `println`) and from packages (eg. `getInt, format`).

A method is a group of declarations and statements that is given a name and may be called upon by this name to perform its intended action. For example, in the early example

```
static double SumNos()
```

is called by

```
total = SumNos();
```

and

```
static void DisplayAnswer(double answer)
```

is called by

```
DisplayAnswer(total);
```

The first (`SumNos`) is an example of a typed method that returns a value - in this case `double`. When it is called the statements of the method are executed and the value calculated is returned to the statement that called the method and "used" - here it is assigned to a variable `total`.

The second (`DisplayAnswer`) is called for its effect. It is declared as `void` - it does not return a value - but it receives some data (a `double` variable which it will refer to as `answer`) to use in its execution.

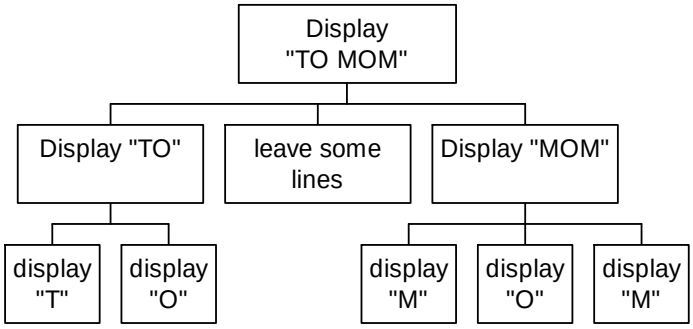
As another example of using methods to structure a program, consider a program to display a Mother's Day message on the screen, such as

```
*****
  *
  *
  *
  *

 ***
*   *
*   *
*   *
 ***


*   *
** **
* * *
*   *
*   *

 ***
*   *
*   *
*   *
 ***

*   *
** **
* * *
*   *
*   *
```

We can plan a program to do this using a style of programming known as ***top-down design*** where we start at the top and consider what needs to be done to achieve the goal, and break the overall goal into sub-goals each of which can be solved independently. In this case in order to achieve the goal of displaying the message we need first to display the word "TO", then after a few blank lines the word "MOM". Each of these are sub-goals, so we can consider them independently

- *what do we need to do to display "TO"?*
  display the block letter "T", then the block letter "O"

- *what do we need to do to display "MOM"?*
  display the block letter "M", then "O", then "M" again.



This provides the structure for our program, and we start at the top and write our main method as if the other methods existed:

```
public static void main(String[] args)
/*-----------------------------------
* the main method calls other methods to display
* the words TO and MOM with blank lines in between
*/
{
  DisplayTO();
  System.out.println();
  System.out.println();
  DisplayMOM();
}
```

ToMom.java (a)

Now we write the methods `DisplayTO` and `DisplayMOM`:

```
static void DisplayTO()
//---------------------
// This method calls other methods to display T and O
{
  DisplayT();
  DisplayO();
}

static void DisplayMOM()
//----------------------
// This method calls other methods to display M, O and M
{
  DisplayM();
  DisplayO();
  DisplayM();
}
```

ToMom.java (b)

And now we can write the methods to actually execute the output statements. Note that although 5 letters are displayed we only need to write 3 methods as the methods `DisplayO` and `DisplayM` are each called twice.

```
static void DisplayT()
```

```
//---------------------
// This method displays *'s making up the letter T
{
  System.out.println("*****");
  System.out.println("  *  ");
  System.out.println("  *  ");
  System.out.println("  *  ");
  System.out.println("  *  ");
  System.out.println();        // leave a blank line
}

static void DisplayO()
//---------------------
// This method displays *'s making up the letter O
{
  System.out.println(" *** ");
  System.out.println("*   *");
  System.out.println("*   *");
  System.out.println("*   *");
  System.out.println(" *** ");
  System.out.println();        // leave a blank line
}

static void DisplayM()
//---------------------
// This method displays *'s making up the letter M
{
  System.out.println("*   *");
  System.out.println("** **");
  System.out.println("* * *");
  System.out.println("*   *");
  System.out.println("*   *");
  System.out.println();        // leave a blank line
}
```

ToMom.java (c)

When a program is executed control is passed to the main method. The first statement is the call to `DisplayTo`, so control is transferred to the first statement of `DisplayTo` which is a call to `DisplayT`, so control is transferred there, and the sequence of `println` statements are executed. When the end of `DisplayT` is reached, control is transferred back to `DisplayTo` and the next statement is the call to `DisplayO`, so control is transferred there, and its `println` statements are executed. Then control is transferred back to `DisplayTo`, but the end has now been reached so control is transferred back to the main method and its next statement is executed. Execution continues in this manner until the end of the entire program is reached.

This program will always display the message made up of asterisks. It is possible to change it so that each method is passed the character to use in forming the letter - this is done by modifying the methods so that they can receive a parameter and use it, and specifying a value for the parameter inside the brackets when the method is called.

For example,

```
static void DisplayT(char c)
```

```
//--------------------------
// This method displays the letter T made of the
// char passed in as parameter c
{
  System.out.println(" "+c+c+c+c+c);
  System.out.println("   "  +c    );
  System.out.println("   "  +c    );
  System.out.println("   "  +c    );
  System.out.println("   "  +c    );
  System.out.println();        // leave a blank line
}
```

ToMom2.java (a)

If `DisplayO` is also changed to accept a parameter, `DisplayTO` becomes

```
static void DisplayTO()
//---------------------
// This method calls other methods to display the
// letters T and O, and passes to them as a parameter
// the char to use
{
  DisplayT('+');
  DisplayO('@');
}
```

ToMom2.java (b)

As another variation, we could input the character to be used to draw the letters, and pass it on to `DisplayTO` and `DisplayMOM` which will pass it on to the methods they call. In this example, a method is used to ask for and read in the character and this character is then returned to the main method and passed on to the other methods.

```
public static void main(String[] args)
//-------------------------------------
// the main method calls other methods to display
// the words TO and MOM with blank lines in between

{
  char ch;                 // character to be used

  ch = InputCharacter();   // get the character from
                      // the input method
  DisplayTO(ch);           // pass ch on to DisplayTO
  System.out.println();
  System.out.println();
  DisplayMOM(ch);          // pass ch on to DisplayMOM
}
```

ToMom3.java (a)

`DisplayTO` receives the character parameter, and passes it on to `DisplayT` and `DisplayO`:

```
static void DisplayTO(char ch)
//---------------------------
```

```
// This method calls other methods to display the letters
// T and O, and passes to them the char to use
{
  DisplayT(ch);
  DisplayO(ch);
}
```

ToMom3.java (b)

These ToMom examples illustrate the different forms of methods declaration and the corresponding ways of calling a method (with or without parameters, returning a value or not).

METHOD DECLARATION
modifiers type methodName (*parameters*) { *variable declaration*; *statements*; return *expression*; *//typed method only* }

The ***modifiers*** you can use are `public`, `static` and `final`. At this early stage all your methods will be `static`, and the main method must be `public` as well.

type is the data type of the value that the method will return, or `void` if the method is called merely for its effect and does not return a value. For example,

```
static void DisplayMOM()          from ToMom.java
static char InputCharacter()      from ToMom3.java
static double SumNos()            from ReadSumNos2.java
```

If a method is typed (not void) then the value to be sent back to the calling method must be specified in the **`return`** statement. The value returned may be a constant, a variable, or an expression. For example, in method `InputCharacter`, the character is read into a variable and the value of this variable (`charToUse`) is returned to the main method.

```
static char InputCharacter()
// This method asks the user to enter the character to use,
// and then returns it to the main method
{
  char charToUse;

  System.out.print("Type the character to use > ");
  charToUse = Keyboard.getChar();
  System.out.println();
  return charToUse;
}
```

The brackets after the method name are compulsory, even if there are no parameters. If there are parameters, they are listed as *`data_type parameter_name`* pairs, with commas separating them if more than one pair. For example

```
static void DisplayTO(char ch)
static int AddNums(int num1, int num2)
```

Methods are called by their name. If the method is void, then the method name (with any associated parameters) forms a statement by itself.

```
DisplayTO('#');
```

If the method is typed, then the item of interest is the value it returns and the method name appears in an assignment statement or an output statement or anywhere else it would be appropriate to use the value that the method returns.

```
ch = InputCharacter();
```

or `System.out.println("The answer is " + AddNums(17.3,25.4));`

When using parameters, the parameters defined in the method declaration are known as ***formal parameters***, while the parameters that are passed to the method when it is called are known as ***actual parameters*** or ***arguments***. It is essential that there are the **same number and type** of actual as formal parameters, and that they are **in the correct order**. Formal parameters are always identifiers - when the method is called the value of the actual parameter is associated with the identifier and used in the method in place of the identifier. Actual parameters may be variables, constants or expressions that supply a value of the required type. Some valid examples are:

Method declaration	**Method call**
`void XYZ()`	`XYZ();`
`void XYZ(int num1)`	`XYZ(13);`
`void XYZ(int num1, double num2)`	`XYZ(1,1.5);`
`void XYZ(char ch, int val)`	`XYZ('*',5);`
`int XYZ(double val)`	`ans = XYZ(17.5);`
`double XYZ(int num1, int num2)`	`inverse = 1/XYZ(4,5);`

Some invalid examples are

`void XYZ(int num1)`	`XYZ();`
incorrect number of arguments	
`void XYZ(int num1)`	`XYZ(1,2,3);`
incorrect number of arguments	
`void XYZ(int num1)`	`XYZ(1.5);`
incorrect argument type	
`void XYZ(int num1, double num2)`	`XYZ(1.5,2);`
incorrect argument order	
`void XYZ(int num1)`	`ans = XYZ(23);`
incorrect method call - XYZ is void	
`double XYZ(int num1)`	`int ans = XYZ(23);`
method returns double, cannot assign to int variable	
`int XYZ(double val)`	`XYZ(5.3);`
value returned is not used -	
does not give a compilation error but poor programming	

There are a number of advantages to using methods:

- Giving a task a name makes it easier to refer to. Naming reduces the need for comments as it makes the code more **self-documenting**.

- Code that calls clearly named methods is **easier to understand** than code in which all the actions take place in the main method. Programs which use methods are easier to understand because of the way they break the program up into manageable sections. If all the code were in the main method we would be confronted with the full complexity of the program.

- If a method is used to perform a task it can be **used many times** in the same program by merely invoking the method repeatedly. The code for the task need not be reproduced each time it is needed (eg. `DisplayO` and `DisplayT`).

- A method can be saved in a package of useful methods and imported by any program that needs it. Thus a method can be **reusable** in many programs. (eg. `Keyboard.getInt()`)

- Once a method is written and properly tested it can be used without any further concern for its inner workings - we can focus on *what* it does and not *how* it does it.

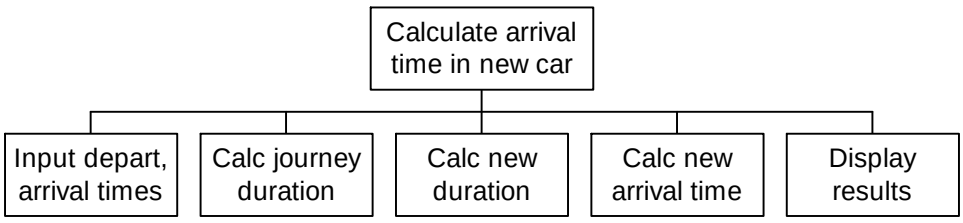
- Large, complicated systems are written by teams of programmers. By structuring the system in a modular fashion it can be divided into modules small enough to be worked on by one team. Each of these modules can in turn be broken down into submodules small enough to be designed by an individual programmer.

As an example of using methods to structure a program, consider a variation on Exercise 5.8:

> Sue has bought a new car that can go 10% faster than her old one. Previously, on a trip to her parents home in Pofadder she would leave home at a certain time and arrive at her parents house some hours later. Driving her new car, and given a new departure time, what time will she now arrive at her parents home? Your program must read in the original departure and arrival times and the new departure time using the 24hour clock.

In order to solve this problem we must

- read in the departure and arrival times
- use the original times to calculate the time the journey took
- reduce this time by 10% because the new car goes faster than the old one
- calculate the arrival time by adding the new journey time to the new departure time and expressing as a time of day.



With this structure in mind we can write the main method:

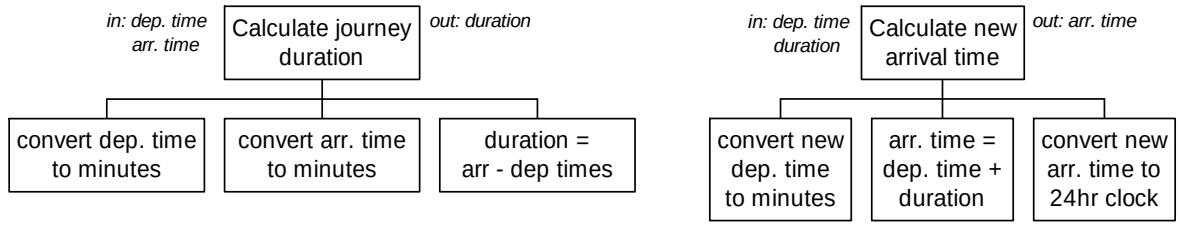
```
public static void main(String[] args)
//-----------------------------------
// the main method reads the departure and arrival times
// and calls other methods to do the calculations and
// display the results.
{
  int departOld,   // departure time (24hr clock)
      arriveOld,   // arrival time in old car
      departNew,   // new departure time
      arriveNew;   // arrival time in new car
  int oldJourney,  // time journey took in old car (mins)
      newJourney;  // time journey took in new car (mins)

  DisplayHeading();

// Input departure and arrival times
  System.out.print("  Enter original departure time  > ");
  departOld = Keyboard.getInt();
  System.out.print("  Enter arrival time - old car   > ");
  arriveOld = Keyboard.getInt();
  System.out.print("  Enter departure time - new car > ");
  departNew = Keyboard.getInt();

  oldJourney = CalcOldJourneyTime(departOld,arriveOld);
  newJourney = (int)(0.9 * oldJourney);

  arriveNew = CalcNewArrivalTime(departNew,newJourney);
  DisplayResults(departOld,arriveOld,departNew,arriveNew);
}
```

NewCar.java (a)

The main method inputs the old departure and arrival times, and the new departure time. Then a method is called to calculate the duration of the original journey. This method is typed and returns an `int` value - the duration - and is passed 2 parameters to use in its calculations - the departure and arrival times. At this stage we are not concerned with how it will do this - we merely assume that we have a method that will accepts the 2 values and calculate the required result.

The next task is the calculation of the journey duration in the new car. This could be written as a separate method but since it is a simple calculation and is only used once in the program, it is done in the main method. Then another method is called to calculate the new arrival time. It is passed 2 parameters - the departure time in the new car and the journey duration, and returns a value - the new arrival time. Finally an untyped method is called to display the results and is passed in the values to be displayed.

Now that the structure is in place we can plan the missing methods:



There are 3 occasions when we need to convert a 24hr time to minutes. It seems to make sense to write a single method to achieve this and use it the 3 times, each time passing it the appropriate 24hr value. So we write the methods as if this method exists:

```
static int CalcOldJourneyTime(int depart, int arrive)
//---------------------------
// this method converts both times to minutes and subtracts
// them to get the original duration of the journey
{
  int departMins,  // the departure and
      arriveMins;  // arrival times in mins after midnight
  int time;        // the journey time in minutes

  departMins = CalcMins(depart);
  arriveMins = CalcMins(arrive);
  time = arriveMins - departMins;
  return time;
}
```

NewCar.java (b)

Notice that `CalcMins` is called twice here, that different values are supplied as the argument (actual parameter) and that the value returned is assigned to a different variable each time. The `return` statement specifies that the variable `time` (which represents the journey duration) is the value that must be returned.

```
  static int CalcNewArrivalTime(int depart, int journey)
//---------------------------
// this method determines the new arrival time in minutes after
// midnight, then expresses it in the 24 hour clock.
{
  int departMins,  // the new departure and
      arriveMins;  // arrival times in mins after midnight
  int hr,          // the arrival hour
      min,         // the arrival minute
      time24;      // the arrival time using 24hr clock

    departMins = CalcMins(depart);
    arriveMins = departMins + journey;
    hr = arriveMins / 60;
    min = arriveMins % 60;
    time24 = hr*100 + min;
    return time24;
}
```

NewCar.java (c)

`CalcMins` is called again in this method - to convert the new departure time to minutes after midnight - and then the journey duration is added to get the new arrival time which is then converted back to the 24 hour clock.

Finally the method to convert from 24 hour clock to minutes after midnight is written. Notice that the formal parameter, `time24`, receives the value of a different argument each time it is called, does the calculation with this value and returns the value of its variable `timeMins`.

```
static int CalcMins(int time24)
//-----------------
// Converts a time in 24hr clock to minutes after midnight
{
  int hr,          // the hour portion of the 24 hour time
      min,         // the min portion of the 24 hour time
      timeMins;    // the time converted to minutes

  hr = time24 / 100;        // integer division
  min = time24 % 100;       // remainder after integer division
  timeMins = hr*60 + min;
  return timeMins;
}
```

NewCar.java (d)

When this program is run the effect is

```
Program to calculate arrival time in a new car
----------------------------------------------

  Enter original departure time (24hr clock)  > 0600
  Enter arrival time when using old car       > 1530
  Enter departure time when using new car     > 0700

In the old car
  the departure time was 600
  the arrival time was   1530
In the new car
  the departure time is  700
  the arrival time is    1533
```

You should always test your programs thoroughly. Are you sure that this answer is correct?

The first time I ran this program I used simple values (see next page), and because the depart time was zero (midnight) I could calculate the minutes involved easily (600) and 90% of this is 540 which is 9 hours so as a first attempt it seemed OK.

```
Program to calculate arrival time in a new car
----------------------------------------------

  Enter original departure time (24hr clock)  > 0000
  Enter arrival time when using old car       > 1000
  Enter departure time when using new car     > 0000

In the old car
  the departure time was 0
  the arrival time was   1000
In the new car
  the departure time is  0
  the arrival time is    900
```

Then I checked using a time with non-zero departure time so I could see the total minutes calculation was still OK, then different old and new depart times. If possible, just change 1 value at a time so that if you detect an error it is easier to isolate what caused it. Use simple values to start with, but once you are satisfied that it seems to be working, use more complex values and check using a selection of inputs with different characteristics.

6.3 Scope of variables

At this stage, and with this exercise as an example, lets consider the **block structure** of Java programs and the **scope** of the variables.

A Java program consists of a number of **blocks** which are demarcated by the open and close curly brackets `{ }`. Everything inside a block is considered a unit and exists more or less independently of other blocks. In particular, each method is a block. Blocks group related statements together, and define where one section of code ends and another begins. Blocks may themselves contain sub-blocks - for example, each class is a block and contains a number of methods each of which are blocks.

Each variable declared in your program has a particular area or **scope** in which it is valid and can be used. The scope of a variable is the block in which that variable is defined. At this stage, we'll consider a block as a method, although in the next section you'll see that methods themselves often contain a number of blocks. So if a variable is declared in the main method, it can only be used in the main method. If you want to use its value in another method you must explicitly pass it to that method as an argument, and the other method must have formal parameters to receive it.

For example, if we didn't pass arguments to the method `CalcOldJourneyTime`

```
    oldJourney = CalcOldJourneyTime();
```

and wrote `CalcOldJourneyTime` to use `departOld` and `ArriveOld` from the main method

```
  departMins = CalcMins(departOld);
  arriveMins = CalcMins(arriveOld);
  time = arriveMins - departMins;
```

we would get the following compilation errors

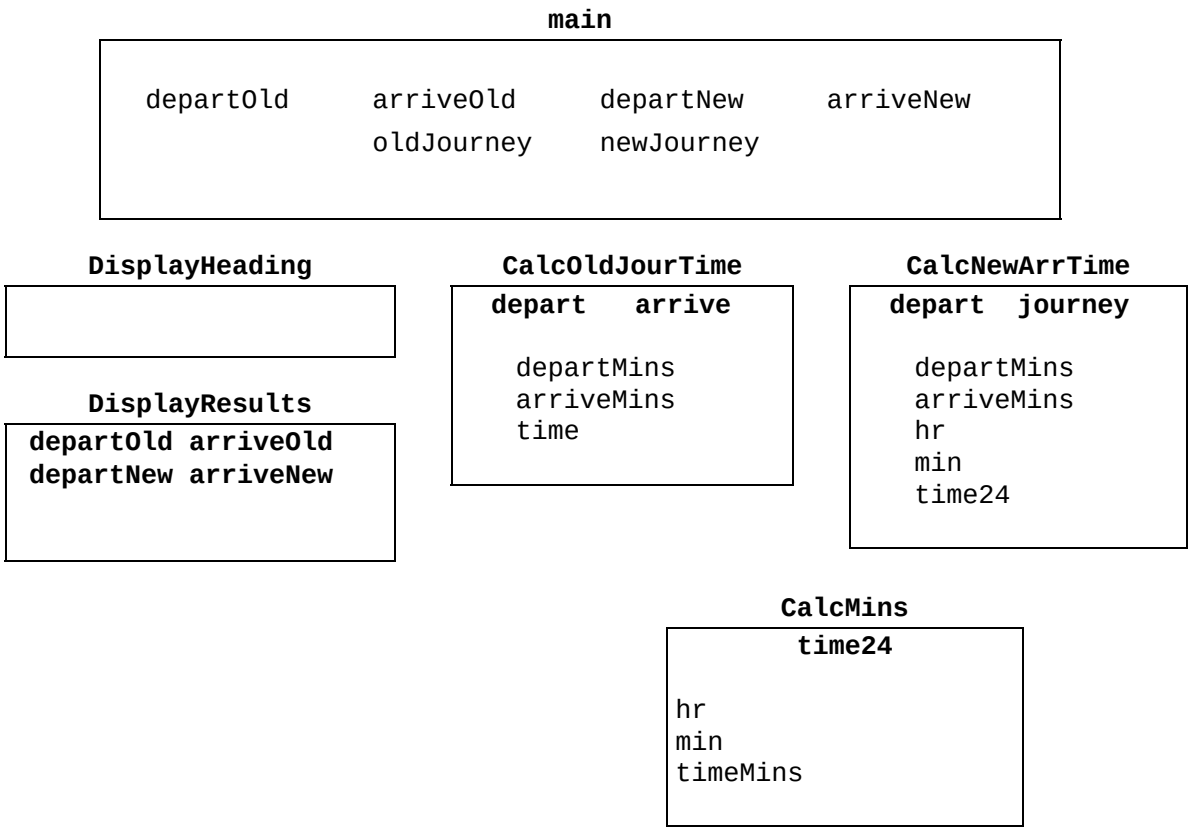
```
NewCar.java:62: Undefined variable: departOld
    departMins = CalcMins(departOld);
                          ^
NewCar.java:63: Undefined variable: arriveOld
    arriveMins = CalcMins(arriveOld);
                          ^
```

Because `departOld` and `arriveOld` are not declared within the block of `CalcOldJourneyTime` they are not known in that method.
Another consequence of this independence of variables within blocks is that the same identifier (variable names) can be used in different blocks, or different names can be used to refer to the same value in different blocks. This sounds confusing, and can be if the programmer uses identifiers indiscriminately without consideration of what the variable is representing. At times however, it is useful and indeed necessary.

For example, the duration of the original journey is called `oldJourney` in the main method and `time` in `CalcOldJourneyTime`, while the identifiers `departMins` and `arriveMins` are used in both `CalcOldJourneyTime` (to refer to the original departure and arrival times expressed as minutes after midnight) and in `CalcNewArrivalTime` (to refer to the new departure and arrival times expressed as minutes after midnight). The identifiers `hr` and `min` are used in a number of places to refer to hours and minutes in whatever context applies in the method (block) in which they are used.

The blocks and the scope of variables in the `NewCar` program are:

`class NewCar`



Any variables declared in the class block (ie. before the main method) are available to the entire class. However, with the exception of constants that should be declared here, it is not good programming practice (at this stage).

Exercises

6.1 Modify the `ToMom` program so that 2 characters are input and are used to draw alternate letters. Then modify the `ToMom` program so that 2 characters are input and are used to draw alternate lines of each letter.

6.2 Write a program to display the following shapes (one below the other) each using different characters which you have input. Structure your program to use methods to draw the common portions of each shape (`circle`, `triangle`, `rectangle`, `arrowhead`).

6.3 Write a program that will read in two distances expressed as yards, feet and inches, convert them to metres and add them (result in metres). (1 yard=3 feet, 1 foot=12 inches, 1 inch=2,54 cm)

6.4 A worker in a factory must punch a time card whenever he enters or leaves the premises. On a particular day he arrives in the morning, leaves at lunchtime, returns after lunch and leaves again at the end of the day. Write a program that inputs the 4 times (24hour clock) and his hourly rate of pay, and calculates how much he earns that day.

6.5 Restructure your programs from Chapter 5 to use methods where appropriate. For most of them you will need to input the data in the main method, and then pass these values to a calculation method which calculates and returns the result, and the input values and result are then passed to a display method which displays the result attractively.

7. Selection

Programs that use methods and repetition as their structuring mechanisms can be very complicated but they are entirely predictable. Interesting programs are those which can choose between different courses of action and respond flexibly to different circumstances.

7.1 Boolean expressions

When evaluating different circumstances in a program you need to be able to determine whether 2 values are equal or unequal or what their relative ranking is. Just as arithmetic formulas evaluate numeric expressions to obtain a numeric value, so too do condition formulas evaluate boolean expressions to obtain one of the two boolean values: **true** or **false**.

The simple boolean operators used in Java to compare numeric values are

==	equal to	!=	not equal to
>	greater than	>=	greater than or equal to
<	less than	<=	less than or equal to

The results of such an expression can then be stored in a boolean variable, for example

```
boolean hasDP;
double clMark;

hasDp = clMark>=50;
```

Boolean variables can be displayed using the print statements, for example

```
System.out.println("Classmark is "+clMark+" hence DP="+hasDP);
```

which will output

```
Classmark is 65 hence DP=true
```

or

```
Classmark is 35 hence DP=false
```

Boolean variables can be combined using the operators

&&	and	!	not
||	or	^	xor

The results of these operations are

and &&	false	true
false	false	false
true	false	true

not !	
false	true
true	false

or ||	false	true
false	false	true
true	true	true

xor ^	false	true
false	false	true
true	true	false

So, extending the previous example, we can write

```
boolean hasDP; isaPass;
double classMark; examMark

hasDp = classMark>=50;
isaPass = hasDP && examMark>=50
```

or with other criteria

```
isaPass = examMark>=50 || (hasDp && examMark>=40)
```

There is precedence between the operators and `!` has the highest precedence and will be evaluated first, followed by `&&` which will always be evaluated before `||` so we could have written this last statement as

```
isaPass = examMark>=50 || hasDp && examMark>=40
```

Also, as shown, the arithmetic operators take precedence over the boolean operators so brackets are not needed and the comparisons will always be executed first before the boolean results are combined using the boolean operators.
Eg. the bracket in this statement are not required (unlike many other languages)

```
isaPass = (examMark>=50) || hasDp && (examMark>=40)
```

Note that the form of and `&&` and or `||` shown here are the "short-circuit" operators, which means that the first operand is evaluated, and depending on the result, the second operand may not be evaluated at all, because

- for a `&&` b, if a is false, it means that the result will always be false so there is no need to evaluate b.
- for a`||`b, if a is true, it means that the result will always be true so there is no need to evaluate b.

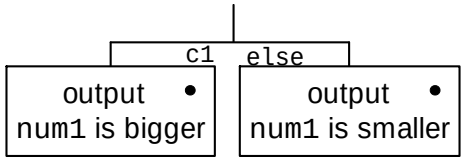
This means it is quite valid to write an expression such as

```
b!=0 && a/b>1;
```

because if b is zero the division will not be attempted since `b!=0` will return false.

7.2 if-else Statements

The if-else statement is used to choose one of two alternative actions based on the result of a boolean value.




```
c1 : if num1>num2
```

If the boolean expression (num1>num2) evaluates to true then "num1 is bigger" is output, otherwise "num1 is smaller" is output.

Consider the following program

```
/*
* Illustrates a simple if-else statement
* --------------------------------------
*/
import Utilities.Keyboard;

public class SimpleIf
{
  public static void main(String[] args)
  {
    int num1,num2;

    System.out.print("Enter a number > ");
    num1 = Keyboard.getInt();
    System.out.print("Enter another number > ");
    num2 = Keyboard.getInt();

  // the selection
    if (num1>num2)
      System.out.println(num1+" is bigger than "+num2);
    else
      System.out.println(num1+" is smaller than "+num2);

    System.out.println();
  }
}
```

SimpleIf.java

The output from this program if 3 and 5 are entered is

```
Enter a number > 3
Enter another number > 5

3 is smaller than 5
```

and if 21 and 15 are entered the output is

```
Enter a number > 21
Enter another number > 15

21 is bigger than 15
```

If the condition tested is true, the statement(s) immediately following the condition are executed (the *then-part*), if false the statements following the else are executed (the *else-part*).

The form of the if-else statement is

IF STATEMENT

```
if (boolean expression)
  statement;
else
  statement;
```

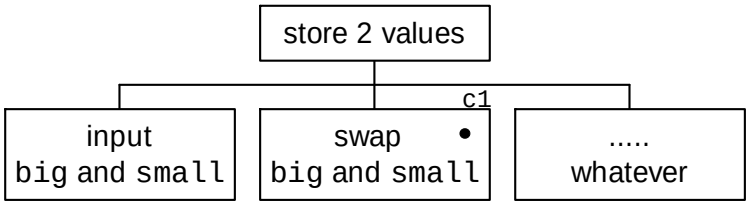
As is usual, `statement` can either be a single statement in which case brackets are not required, but if more than one statement needs to be executed for the then-part or else-part they must be bracketed with `{ }` to form a block.

The boolean expression may consist of any expression that returns a boolean value

- a simple expression of the type
```
if (a%2==0)
  System.out.println(a+" is even");
else
  System.out.println(a+" is odd");
```
- a more complex expression such as
```
if ((let>='a' && let<='z') || (let>='A' && let<='Z'))
  System.out.println(let+" is a letter of the alphabet");
else
  System.out.println(let+" is not a letter of the alphabet");
```
- a boolean variable which has had the result of a boolean operation stored in it such as
```
boolean isaPass;
isaPass = examMark>=50 || (hasDp && examMark>=40);
if (isaPass)
{
  System.out.println("Congratulations");
  numPass++;
}
else
{
  System.out.println("Bad luck - try harder");
  numFail++;
}
```
- the result of calling a method that returns a boolean value
```
if (hasPassed(classMark,examMark))
  numPass++;
else
  numFail++;

static boolean hasPassed(double class; double exam)
{
  boolean pass;
    :
  statements to assign true or false to pass
    :
  return pass;
}
```

The else-part is optional - often we wish to do something if a condition is true but there is no action required if it is false.

For example, assume we are inputting 2 values and need make sure the larger value is in the variable called `big`, and the smaller value is in the variable called `small`. So we input the first value into big and the second into small, and if they are the wrong way round we swap them, if not then nothing needs to be done:




```
c1 : if big<small
```

In Java:

```
if (big<small)
{                       // exchange the values
  int temp = big;
  big = small;
  small = temp;
}
… etc
```

IfnoElse.java

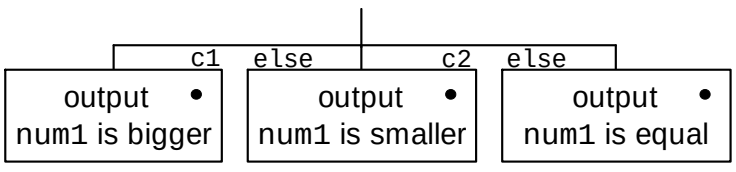
When using if statements it is important to consider whether all possible cases have been catered for. For example, in out first example (`SimpleIf.java`), we determined which of 2 values input is larger. But what if they were the same - what would have happened then?

```
if (num1>num2)
  System.out.println(num1+" is bigger than "+num2);
else
  System.out.println(num1+" is smaller than "+num2);
```

The boolean expression (`num1>num2`) would return false so the else-part will be executed and the output obtained would be (eg.)

```
9 is smaller than 9
```

This is clearly not what is wanted. The problem is that we were assuming that only 2 possible cases existed - a number is either bigger or smaller that another - whereas there are in fact 3 cases - bigger, smaller or equal.



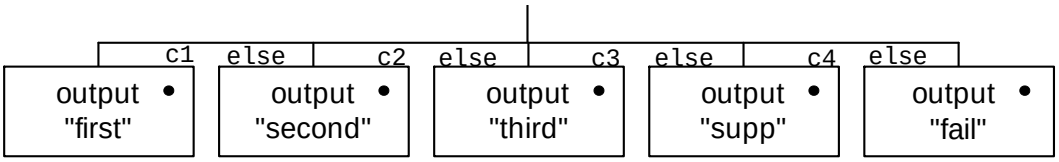

```
c1 : if num1>num2
c2 : if num1<num2
```

In order to make decisions that involve more than 2 possibilities we use cascading ifs:

```
if      (num1>num2)
  System.out.println(num1+" is bigger than "+num2);
else if (num1<num2)
  System.out.println(num1+" is smaller than "+num2);
else    // num1=num2
  System.out.println(num1+" is equal to "+num2);
```

The first if checks for the one possibility and if true the then-part is executed. If false one possibility has been eliminated and the next if is contained in the else-part and checks a second possibility, and so on. The last condition is the default because if a number is not smaller or larger than another then it must be equal, so the last condition does not need a test although the comment clarifies what state exists at this point and is helpful for the human reader. Layout is important here, and helps the user to understand that a series of successive, related conditions are being tested.

Another example that illustrates this clearly is classifying marks:




```
c1 : if mark>=75
c2 : if mark>=60
c3 : if mark>=50
c4 : if mark>=40
```

```
if      (mark>=75)
  System.out.println(mark+" is a first");
else if (mark>=60)
  System.out.println(mark+" is a second");
else if (mark>=50)
  System.out.println(mark+" is a third");
else if (mark>=40)
  System.out.println(mark+" is a supp");
else    //mark<40
  System.out.println(mark+" is a fail");
```

part of `ClassifyMark1.java`

Note that the conditions are carefully ordered so that each eliminates a certain range of values, and the statement that checks whether mark is over 50 can only be reached if it is less than 60, thus establishing the range.
The following examples appear at first glance to be similar, but are incorrect:

```
if (mark>=75)
  System.out.println(mark+" is a first");
if (mark>=60)
  System.out.println(mark+" is a second");
if (mark>=50)
  System.out.println(mark+" is a third");
if (mark>=40)
  System.out.println(mark+" is a supp");
```

```
else     //mark<40
  System.out.println(mark+" is a fail");
```

part of `ClassifyMark1x.java`

In this example the ifs are not contained within the else-part of the previous if so in fact are completely independent tests. Eg, a mark of 80 would result in the following output

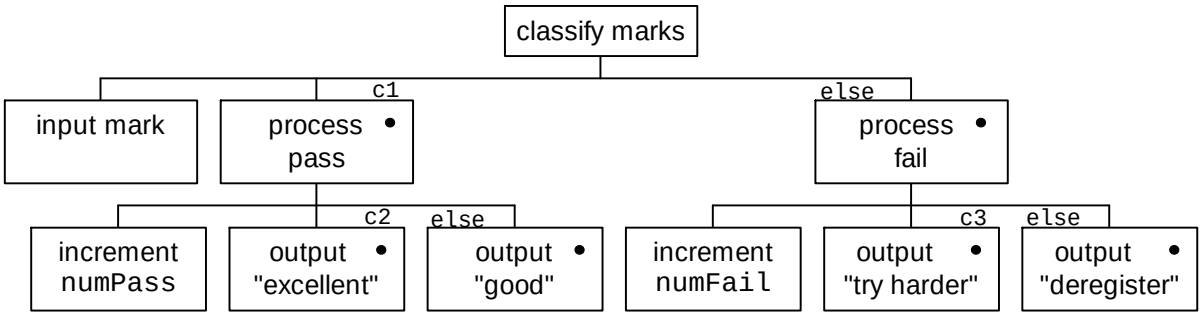
```
80 is a first
80 is a second
80 is a third
80 is a supp
```

because the all the if statements are executed and all the tests are true for a mark of 80.

This next example will not give an incorrect answer, but is clumsy and less efficient because of the repeated testing involved.

```
if (mark>=75)
  System.out.println(mark+" is a first");
if (mark<75 && mark>=60)
  System.out.println(mark+" is a second");
if (mark<60 && mark>=50)
  System.out.println(mark+" is a third");
if (mark<50 && mark>=40)
  System.out.println(mark+" is a fail/supp");
if (mark<40)
  System.out.println(mark+" is a fail");
```

Sometimes conditions contain sub-conditions, in which case a more "classic" type of nested if can be used. For example, we want to count the number of passes and fails, and within each category to classify them further:




```
c1 : if mark>=50
c2 : if mark>=75
c3 : if mark>=40
```

```
// classify using nested ifs
    if (mark>=50)        // a pass
    {
      numPass++;
      if (mark>=75)
        System.out.println("  Excellent!");
      else
        System.out.println("  Good");
```

```
      }
      else              // a fail
      {
        numFail++;
        if (mark>=40)
          System.out.println("  Try harder");
        else
          System.out.println("  Deregister");
      }
```

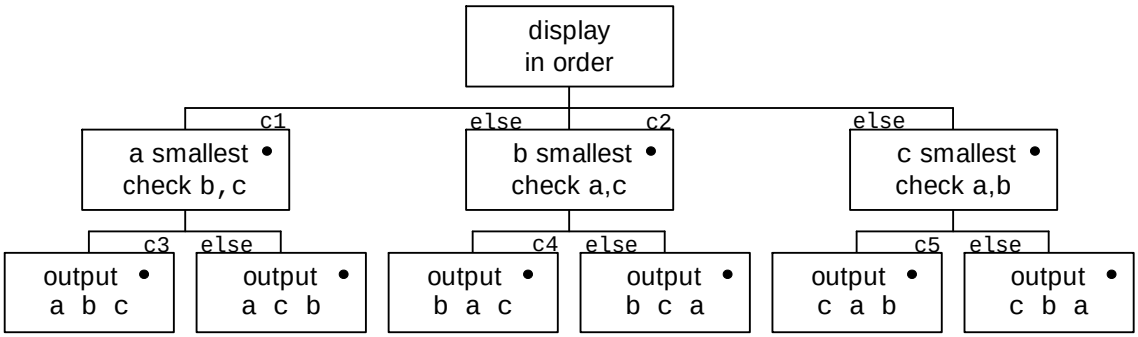
part of `ClassifyMark2.java`

1. Consider a program to output 3 numbers (say `a,b,c`) in ascending order.

   The difficulty here is to keep track of where one is in the process of determining which is smallest, next smallest, … etc. There are in fact 6 possible orderings:

   `a b c` `a c b` `b a c` `b c a` `c a b` `c b a`

   The simplest approach is to determine which of 3 categories the values fall into (a smallest, b smallest or c smallest) and then check further within each category.

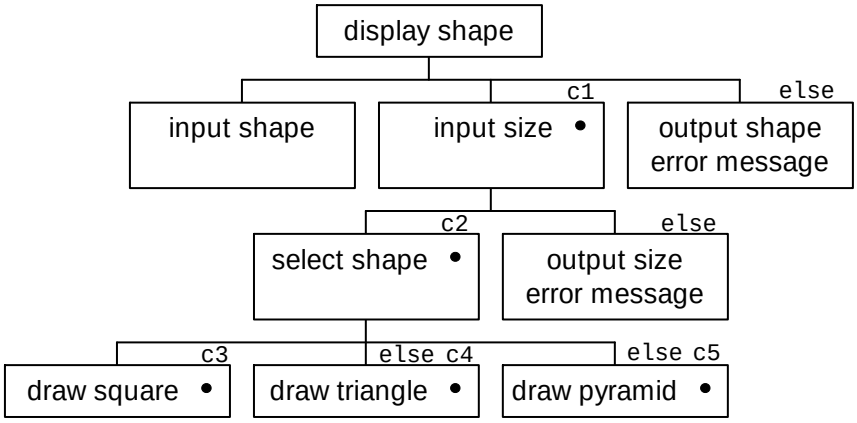



```
c1 : if a<=b and a<=c
c2 : if b<=a and b<=c
c3 : if b<=c
c4 : if a<=c
c5 : if a<=b
```

2. Consider a program to draw either a square, a triangle or a pyramid out of stars. The user must input which shape is wanted and also the size, which must be between 3 and 20.

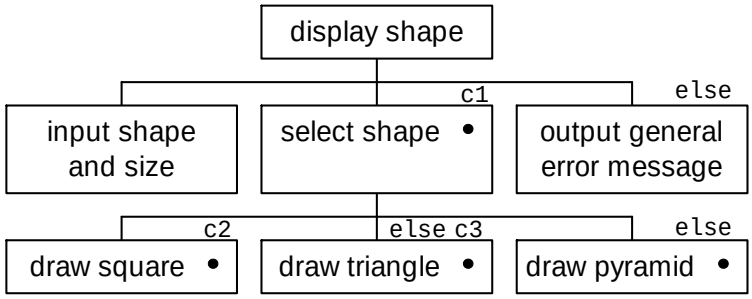
   There are a number of approaches, all of which are valid

   - enter the shape, if it is valid then enter the size, if that is valid then use successive ifs to choose which shape to draw. (`DrawShape1.java`)



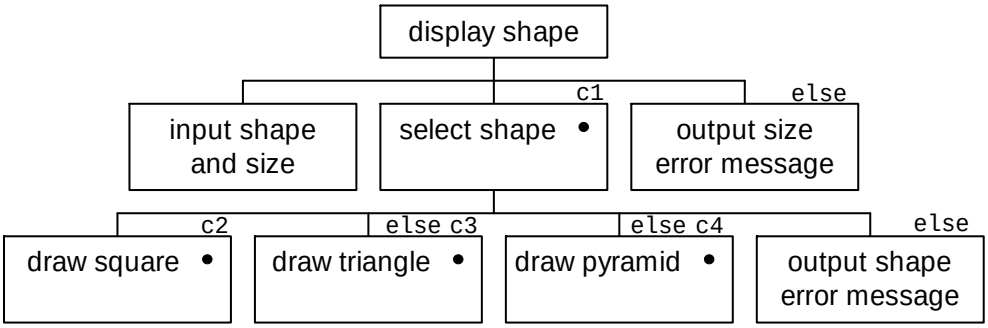

```
c1 : if shape = square or pyramid or triangle
c2 : if size>=3 and <=20
c3 : if shape = square
c4 : if shape = triangle
c5 : if shape = pyramid
```

- enter the shape and size, check both are valid, then use successive ifs to choose which shape to draw. Here separate methods that return boolean values can be used to do the actual validation, and make the main method much clearer and simpler. (`DrawShape2.java`)




```
c1 : if shapeOK and sizeOK
c2 : if shape = square
c3 : if shape = triangle
```

- enter the shape and size, if the size is valid then use successive ifs to select the shape to draw, and if no match for a shape is found, then shape is not valid. (`DrawShape3.java`)




```
c1 : if size>=3 and <=20
c2 : if shape = square
c3 : if shape = triangle
c4 : if shape = pyramid
```

7.3 switch Statements

The if statement is a binary choice statement - it lets you choose between two possible courses of action. Sometimes you want to choose between a number of possible values of a variable, and successive if statements can become cumbersome and tedious - for example, if we wanted to choose between displaying a square, circle, triangle, pyramid or rectangle we would require a statement such as

```
if      (shape=='S' || shape=='s')
    DrawSquare(size);
else if (shape=='T' || shape=='t')
    DrawTriangle(size);
else if (shape=='C' || shape=='c')
    DrawCircle(size);
else if (shape=='R' || shape=='r')
    DrawRectangle(size);
else if (shape=='P' || shape=='p')
    DrawPyramid(size);
else
    System.out.println("Error - invalid shape character");
```

In this case, where a choice is being made between specific values of a single variable an alternate form of choice statement can be used - the **switch** statement.

```
switch (shape)
{
  case 'S':
  case 's': DrawSquare(size);
            break;
  case 'T':
  case 't': DrawTriangle(size);
            break;
  case 'C':
  case 'c': DrawCircle(size);
            break;
  case 'R':
  case 'r': DrawRectangle(size);
            break;
  case 'P':
  case 'p': DrawPyramid(size);
            break;
  default:
      System.out.println("Error - invalid shape character");
}
```

The format of the switch statement is

SWITCH STATEMENT

switch (switch expression)
{ **case** *value : statement*; **break;**
  case *value : statement*; **break;**
  :
  default *: statement*;
}

The switch considers the value of the switch expression and tries to find a match among the case values. If a match is found, then the corresponding statements are executed and the `break` transfers control out of the switch. If no match is found, the statements following the `default` keyword are executed.
Some points to note:

- the switch expression must be integer or character, no reals are allowed;
- the case values must be the same type as the switch expression;
- the case values may appear in any order, but may only appear once;
- each statement may have one or more case values which are specified each with the word `case` and separated from the next case by a colon `:`

eg.

```
switch (digit)
  {
    case –1:
    case -3:
    case –5:
    case –7:
    case -9:
        System.out.println("Negative odd"); break;
    case 0:
        System.out.println("Zero"); break;
    case 1:
    case 3:
    case 5:
    case 7:
    case 9:
        System.out.println("Positive odd"); break;
    default:
        System.out.println("Even, or more than 1 digit");
  }
```

- the case values must be actual constant values - they may not be variables or ranges

eg.

```
case <0 : statement; break;
```

is not allowed, and nor is

```
int  ans;
 :
case ans : statement; break;
```

because `ans` is a variable and only constant case values may be used

- the default statement is optional - if all the options have been dealt with in the case values it can be omitted.
- statements may be compound and bracketed with `{ }`

eg.

```
switch (month)
  { case 4: case6: case 9: case 11:    // Apr,June,Sept,Nov
      numDays = 30; break;
    case 2:                    // Feb - check leap year
    { check = year%4;
      if (check=0)
        numDays = 29;
```

```
      else
        numdays = 28;
    } break;
    default: numDays = 31;         // all 31 day months
  }
```

- if the `break` is omitted, the program will "drop through" and execute the statements for the next case as well

eg.

```
switch (letter)
  { case 'A': case 'E': case 'I': case 'O': case 'U':
      System.out.println("is an upper-case vowel");
    case 'a': case 'e': case 'I': case 'o': case 'u':
      System.out.println("is a lower-case vowel");
    default:
      System.out.println("is a consonant");
  }
```

if `letter` is `'E'`, the output from this switch will be

```
is an upper-case vowel
is a lower-case vowel
is a consonant
```

As a simple example, consider a program that inputs a date of birth as three integers representing day, month and year, and outputs it in full:

```
// output date in full
  System.out.print("The date is " + day + " ");
  switch (month)
  { case 1  : System.out.print("January");   break;
    case 2  : System.out.print("February");  break;
    case 3  : System.out.print("March");     break;
    case 4  : System.out.print("April");     break;
    case 5  : System.out.print("May");       break;
    case 6  : System.out.print("June");      break;
    case 7  : System.out.print("July");      break;
    case 8  : System.out.print("August");    break;
    case 9  : System.out.print("September"); break;
    case 10 : System.out.print("October");   break;
    case 11 : System.out.print("November");  break;
    case 12 : System.out.print("December");  break;
    default : System.out.print("*error*");
  }
  System.out.println(" " + year);
```

part of `OutputDate.java`

The output from this program is

```
Input a date as 3 integers:
  day (1-31)     > 15
  month (1-12)   > 9
  year (eg 1999) > 1999

The date is 15 September 1999
```

Using a switch rather than a succession of ifs improves a program by making it clearer and easier to read; it emphasises that the choice depends on different values of the same variable; and it shortens it by taking out repeated instances of the variable being tested. If the

conditions are right - integer or character variable being tested which can take on a number of different fixed values - then a switch statement should be used instead of successive ifs.

Exercises

7.1 Write a program that inputs the class mark and exam mark for a student and determines whether or not he has passed the course. The requirement is that a student must get at least 50% for the class mark and at least 40% for the exam mark, that the class mark counts one third and the exam mark two thirds of the final mark, and the final mark must be at least 50% for a pass.

7.2 Write a program that allows the user to enter a temperature and whether it is in degrees Fahrenheit (F) or degrees Centigrade (C), and then converts it to the other and displays the result with a suitable message.

$$°F = °C \times \frac{9}{5} + 32 \qquad °C = (°F - 32) \times \frac{5}{9}$$

7.3 Write a program that inputs four numbers in any order and displays them in ascending order.

7.4 Write a program that inputs a circle (the co-ordinates of the centre and the radius) and a point (its co-ordinates), and determines if the point lies inside the circle, on its circumference, or outside the circle. Run your program using the circle centred at (2,3) with radius 5, and check the points (6,6); (1,-3) and (0,0).
(Hint: compare the distance from the point to the centre of the circle, with the radius)

7.5 Write a program that will input the co-ordinates of three points and determine whether or not they lie on a straight line. Run your program with the sets of points
(1,1); (4,8.5); (-3,-9.5); and (1,4); (4,1) and (-3.5,8.5)
(Hint: the lines between any two pairs of points must have equal gradients)

7.6 Write a program that reads in a date as three integer values representing day, month and year (4digits eg 1999), and checks whether it is a valid date.
Note: Jan, March, May, July, Aug, Oct, Dec have 31 days;
April, June, Sept and Nov have 30 days;
Feb has 28 days, except for a leap year (year is divisible by 4) which has 29 days.

7.7 There is a method in the `Math` package called `random()` that will return a (`double`) random number between 0.0 and 1.0. To use it to generate a number between 0 and 100 you could use a statement such as

```
randomNo = 100*Math.random();
```

or for an integer,

```
randomInt = (int)Math.round(100*Math.random());
```

Write a program that you could use to test a child's arithmetic, by generating a sum involving integers between 0 and 10, display it and ask the user to input the answer, then check whether the answer is correct.

7.8 Modify the program you wrote for question 5.6 to find the roots of a quadratic equation, by determining before calculating the roots whether the equation has

- no real roots (b^2-4ac is negative)
- one real root (b^2-4ac is zero)
- two real roots (b^2-4ac is positive)

Appropriate messages should be output in each case.

Test your program with the following equations:

$4x^2 - 28x + 49 = 0$
$3x^2 + x + 1 = 0$
$-x^2 + 3.5x - 17 = 0$
$-x^2 + 4.8x -3.45 = 0$

7.9 Write a program which reads in the lengths of three lines and determines:

- whether or not they can form a triangle (the length of the longest side (x) must be less than the sum of the lengths of the two shorter sides (y and z)).
- if the sides can form a triangle, what sort of triangle (may fit into more than one category)
  - right angled (use Pythagorus: $x^2 = y^2 + z^2$)
  - acute angled ($x^2 < y^2 + z^2$)
  - obtuse angled ($x^2 > y^2 + z^2$)
  - equilateral (all 3 sides equal length)
  - isosceles (2 sides equal length)
  - scalene (no two sides the same length)

Use boolean variables and methods where necessary.

8. Repetition

In many instances you need to repeat a sequence of instructions a number of times, for example, read in marks for 20 students, or display the square roots of all numbers between 1 and 100. Another type of loop is where you want to repeat some actions but don't know exactly how many times, for example, input marks and stop when a negative mark is entered, or while the user enters the wrong answer output an error message and ask them to try again. The first type is called fixed or counting repetition because you know in advance how many times to iterate; the second type is conditional repetition because the loop is repeated while set of circumstances exists. We will consider counting repetition using for loops here; conditional repetition will be covered in chapter 9.

8.1 Simple for loops

Consider the following simple program which lists the numbers between 1 and 10:

```
/*
* Displays a range of numbers
* ---------------------------
*/
public class NumberLoop
{
  public static void main(String[] args)
  {
    System.out.println();

    // the loop
    for (int num=1;num<=10;num++)
    {
      System.out.println(num);
    } // end of loop

    System.out.println();
  }
}
```

NumberLoop.java

The output generated by this program is

```
1
2
3
4
5
6
7
8
9
10
```

The statement that controls the looping is the for statement

```
for (int num=1;num<=10;num++)
{ … }
```

The way this loop is interpreted is

- the loop control variable (`num`) is initialised to 1
- as long as `num` is less than or equal to 10 the statements in the curly brackets are executed (the `println` statement outputs `num`)
- after each execution of these statements, `num` is incremented by 1
- the last 2 steps are repeated until `num` exceeds 10, and then execution continues to the statement following the loop.

If the for statement is changed to

```
for (int num=5;num<=12;num++)
{
  System.out.println(num);
}
```

then the numbers 5, 6, 7, … 11, 12 will be output.

And if the for statement is changed to

```
for (int num=-3;num<=3;num++)
{
  System.out.println(num);
}
```

then the numbers -3, -2, -1, 0, 1, 2, 3 will be output.

More formally, a for loop consists of 4 parts:

FOR LOOP
for (*initialise*; *check*; *update*) { *body* }

The *initialise* part introduces a loop variable which "controls" the number of repetitions of the loop. It often has the form

`int` *identifier = initial value*

although other types of loops are also possible.
Eg. `int num=1;` defines the loop variable `num` and assigns it the initial value of 1

The *check* part provides for the termination of the loop. It compare the loop control variable to some final limiting value and repeats the loop until it reaches that value.
Eg. `num<=10;` allows the loop to continue until `num` is over 10

The *update* part specifies the value by which the loop control variable is to be updated each iteration. In the example here we use a form of assignment we haven't encountered so far, but which is an extremely convenient shorthand notation:

`num++` is equivalent to `num = num+1`

Technically, any statement that changes the value of `num` can be used here but assignments of this nature are most often used.

Other shorthand forms of the assignment statement are

INCREMENT AND DECREMENT ASSIGNMENTS	
`variable ++`	*add one to variable*
`variable --`	*subtract 1 from variable*
`variable += expression`	*add value of expr to variable*
`variable -= expression`	*subtract value of expr from variable*

For example,

`count ++`	is equivalent to	`count = count-1`
`down --`	is equivalent to	`down = down-1`
`inTwos += 2`	is equivalent to	`inTwos = inTwos+2`
`less -= 5`	is equivalent to	`less = less-5`

The *loop body* is contained within curly brackets and consists of all the statements that are to be repeated. Each statement within the loop body is executed first with the initial value of the loop variable, then they are all re-executed with the next value, then with the next, … and so on until the final terminating value is reached. The one exception to this is if the terminating condition is false initially, when the loop body will not be executed at all.

In this loop:

```
for (index=10; index<=1; index--)
```

index is initialised to 10, the terminating condition is checked and found to be false so the loop body is not executed at all and execution continues at the next statement after the loop body.

Always remember that the sequence of events in the execution of a for loop is

- initialise
- check
- execute body } *if condition*
- update } *holds true*
- check
- execute body } *if condition*
- update } *is still true*
- check

…… etc until the check fails

If the loop body consists of only a single statement then the curly brackets are not required - for example

```
for (int num=1;num<=10;num++)
  System.out.println(num);
```

In the example above, the loop variable was "used" in the statements of the loop body - it was displayed. This is not a requirement - the loop variable merely counts through the iterations. For example, consider this loop which outputs "Hello" a number of times:

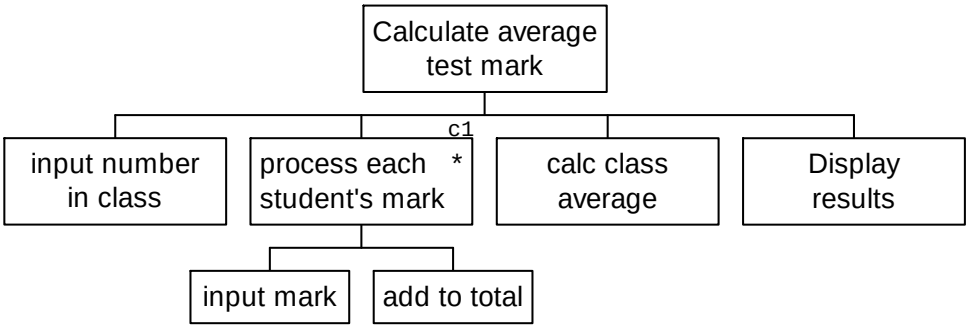
```
for (int count=0;count<3;count++)
  System.out.println("Hello");
```

The output produced is

```
Hello
Hello
Hello
```

In this example `count` is initialised to zero and the loop repeats while `count` is less than 3 - in other words a total of 3 times (0,1,2). This form is often used (initialising to 0 and looping while less than the number of iterations required) when the value of the loop variable is not of significance and we are merely repeating some statements "*n times*". When we work with arrays (later) an array holding 10 elements is indexed 0 to 9 so this range of loop variable values is particularly appropriate.

As another example, consider the following program which asks how many students are in the class, reads in their test marks and calculates the class average.




```
c1 : for st from 1 to numStudents
```

The method that uses a for loop to process each student in turn is shown below:

```
static double ProcessStudentMarks(int numSt)
//-------------------------------
// For each student, inputs a mark and adds it to the total.
// After all marks entered, calculates the average
{
  double mark;          // a student's mark
  double total = 0;         // the running total
  double av;            // the average mark

  for (int st=1;st<=numSt;st++)
  {
    System.out.print("  Enter the mark for student "+st+" > ");
    mark = Keyboard.getDouble();
    total += mark;      // add mark to total
  } // end of loop

  av = total/numSt;
  return av;
}
```

part of `ClassAverage.java`

The variable mark is used to input each students mark in turn. Once it has been added to the running total there is no further need to store an individual student's mark so it is used again for the next mark. Only one variable is needed to input as many marks as are required. Notice the use of the loop variable `st` to annotate the request for input.

The output from this program is

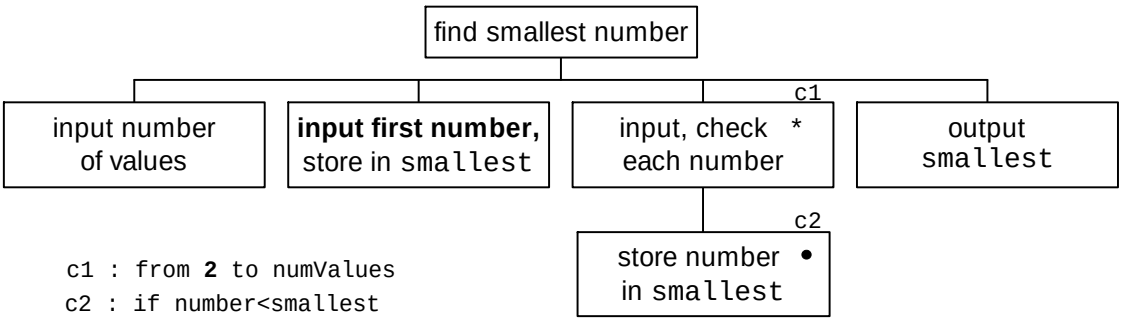
```
Calculation of average test mark for a class
--------------------------------------------

How many students wrote the test? 5

  Enter the mark for student 1 > 34
  Enter the mark for student 2 > 100
  Enter the mark for student 3 > 66
  Enter the mark for student 4 > 50
  Enter the mark for student 5 > 45

The class average for the 5 students is 59.0
```

Some other examples using if statements:

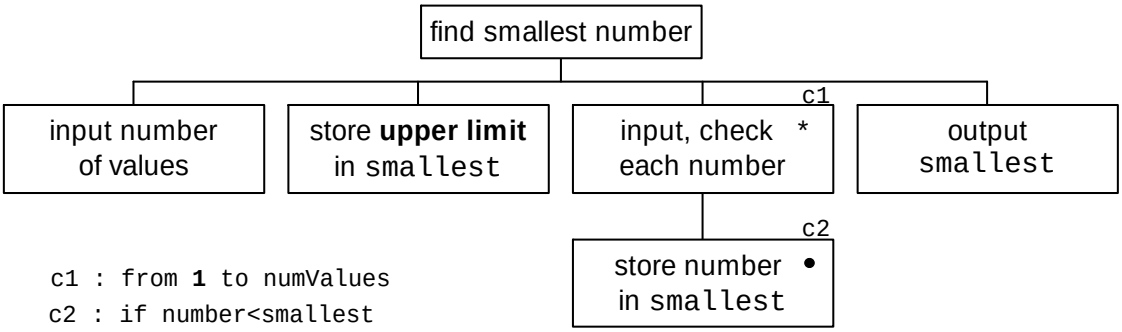
1. Consider the problem of finding the smallest (or largest) of a sequence of numbers that are input one by one.

   The way to solve this is to remember the smallest number so far, and as each number is input to compare it to the smallest-so-far and if it is smaller we can replace the smallest-so-far with the number just input. How do we start the whole process? There are 2 possible ways:

   - the first number input is obviously the smallest-so-far (it is the only number so far!) so we store the first number as the smallest-so-far, and then process all the other numbers in a loop. (`FindSmall1.java`)

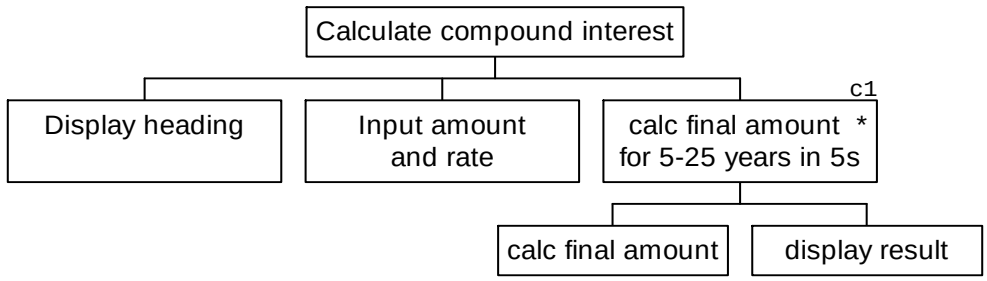


- if we know the range of the numbers (eg. they are all positive, or are between -100 and +100) so that we can identify an upper limit, we can pre-load that as the smallest-so-far, because we know that the first value input has to be smaller than it so will replace it. Then all the numbers can be processed in a loop.
  (`FindSmall2.java`)



As an example of using an update value other than a simple increment, consider a variation on the compound interest program used in Chapter 4:

> Write a program to calculate the result of investing a sum of money (Amt) at a given interest rate (r%) for a number of years (P), where P varies in 5 year intervals between 5 and 25 years.



The loop required is

```
  System.out.println("     years    final amount");
  System.out.println("     -----    ------------");

// loop to calculate the final amount for different periods
  for (int years=5;years<=25;years+=5)
  {
    finalAmt = amount * Math.pow((100+rate)/100,years);
    System.out.println(Formatter.format(years,9)
       + Formatter.format(finalAmt,16,2));
   }
```

part of `CompInterest3.java`

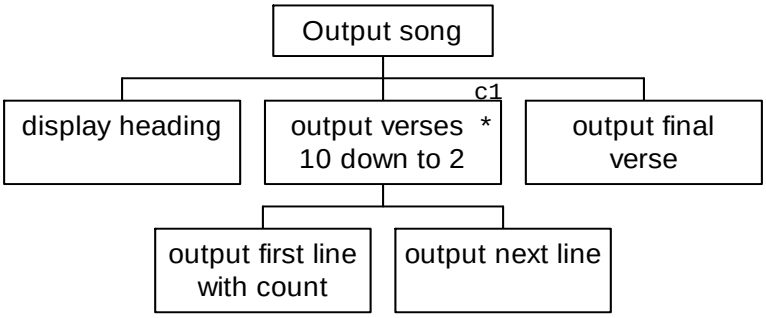
The loop variable `years` steps through 5 to 25 in intervals of 5, and for each value the final amount is calculated and displayed. In order for the output to appear neatly, a heading for

the columns of output should be displayed before executing the loop - you only want the heading displayed once, not each time a new value is calculated.
The output obtained from the full program is

```
Enter initial amount    > R1000
Enter interest rate (%) > 15

     years     final amount
     -----     ------------
        5         2011.36
       10         4045.56
       15         8137.60
       20        16366.54
       25        32918.95
```

As a final example of a simple loop, this one counting backwards, consider a program to output the words of the well-known song

There were 10 in the bed and the little one said "Roll over, roll over"
So they all rolled over and the one fell out.
There were 9 in the bed …… etc




```
c1 : for verse from 10 down to 2
```

Because the last verse is different to the others, the loop must output the lines of verses 10,9,…,2 and the last verse must be output after the loop.

```
// loop to output the verses
    for (int verse=10;verse>1;verse--)
    {
    System.out.println("There were "+ verse +" in the bed and "
      + "the little one said\n  \"Roll over, roll over\"");
    System.out.println("So they all rolled over and "
      + "the one fell out.\n");
    }

// output final lines
    System.out.println("There was 1 in the bed and "
      + "the little one said\n  \"Goodnight, sleep tight!\"");
    System.out.println();
```

part of `TenInBed.java`

Notice the use of the escape character (\) to output the quotation marks in the text.

The last portion of the output of this program is

```
There were 3 in the bed and the little one said
  "Roll over, roll over"
So they all rolled over and the one fell out.

There were 2 in the bed and the little one said
  "Roll over, roll over"
So they all rolled over and the one fell out.

There was 1 in the bed and the little one said
  "Goodnight, sleep tight!"
```

In these for loops the loop control variable has been declared in the for loop

```
for (int num=0; num<10; num++)
```

As a result its scope is limited to the for loop block and it can not be used outside this block. If you need to be able to access the loop variable after the for loop then it must be declared with the other variables at the beginning of the method and not in the for loop itself. Then its scope will be the entire method, not just the loop body.

For example,

```
for (int years=5; years<=25; years+=5)
{ statements };
System.out.println("The last period used was " + years);
```

will result in the compilation error

```
Undefined variable: years
```

while the code

```
int years;
for (years=5; years<=25; years+=5)
{ statements };
System.out.println("The last period used was " + years);
```

will compile and execute correctly.

8.2 Nested loops

Assume we want to write program to display a triangle of stars such as

```
*
**
***
****
```

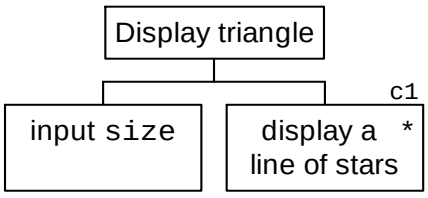
where the size of the triangle is input.

If the size was fixed -eg `4` - we could write program consisting merely of `println` statements such as

```
System.out.println("*");
System.out.println("**");
System.out.println("***");
System.out.println("****");
```

This won't work for an arbitrary size of triangle.
If we consider this more closely we notice that in line 1 we need to output 1 star, in line 2 we need to output 2 stars, in line 3 we need to output 3 … etc. Can we somehow use this relationship to produce the triangle?

If we use a for loop to output the lines we could have something like



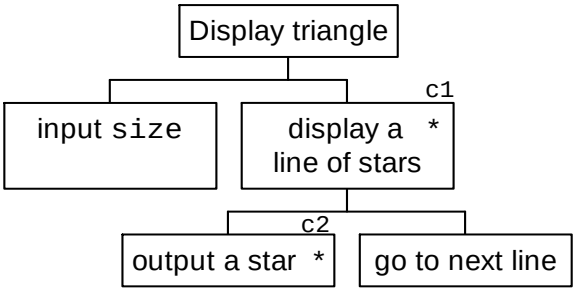

```
c1 : for lineno from 1 to size
```

This could be written as

```
System.out.println("What size triangle do you want? ");
size = Keyboard.getInt();
for (int lineno=1;lineno<=size;lineno++)
{
   display lineno stars on 1 line;
}
```

Now we need to decide how to display exactly `lineno` stars on each line.

The easiest way is to use another for loop, that will repeat `lineno` times and displays 1 star each time.




```
c1 : for lineno from 1 to size
c2 : for count from 1 to lineno
```

This gives us the final program to input the size and draw a triangle of stars of that size:

```
public static void main(String[] args)
{
  int size;                // the size triangle to draw

  System.out.println("What size triangle do you want? ");
  size = Keyboard.getInt();
  System.out.println();

  // loop through the lines
  for (int lineno=1;lineno<=size;lineno++)
  {
    for (int count=1;count<=lineno;count++)
      System.out.print('*');    // output a star lineno times
    System.out.println();       // go to next line
  }
}
```

Triangle.java

This system of a loop within a loop is called a nested loop. For each iteration of the outer loop the nested (inner) loop is executed. Notice that a print statement (not a println) is used to output each star so they appear on the same line, and that an explicit println is used after the inner for loop to move to the next line before the end of the outer loop is reached, thus ensuring that each line of stars appears on a separate line.

The output displayed by this program for a size of 4 is

```
What size triangle do you want? 4

*
**
***
****
```

To illustrate the point that each statement in the inner loop is executed for each iteration of the inner loop, and the inner loop is executed for each iteration of the outer loop, consider the following code fragment:

```
int count = 0;
for (int outer=1;outer<=10;outer++)
{
  for (int inner=1;inner<=6;inner++)
    count++
}
System.out.println(count);
```

What value of `count` will be output?
Make sure you understand why you get the answer (60).

Nested loops aren't limited to just one level - they can be nested as deep as is necessary. The same principles apply - the outer loop is executed for the specified number of iterations, and any loop contained within the outer loop is executed the specified number of times for each repetition of the outer loop, and any loop contained within the inner loop is executed the specified number of times for each iteration of the inner loop (which is executed each iteration of the outer loop) …… etc.

So if we extend the previous example what value of `count` will now be output?

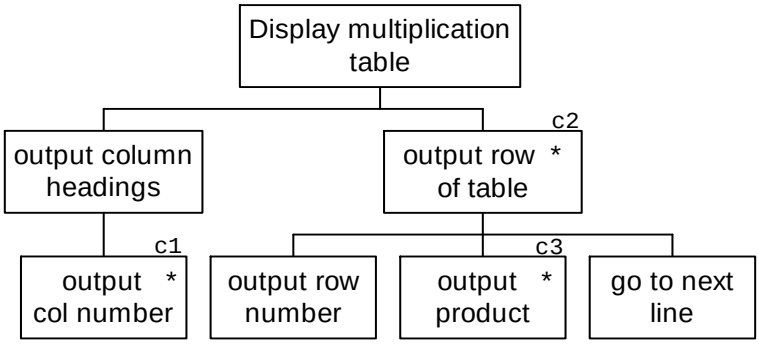
```
int count = 0;
for (int outer=1;outer<=10;outer++)
{
  for (int inner=1;inner<=6;inner++)
  {
    for (int innest= 1;innest<=4;innest++)
      count++
  }
}
System.out.println(count);
```

Make sure you understand why you get the answer (240).

Consider a program to generate a multiplication table that can be used to look up the product of any two integers between 1 and 9. The type of output required is:

```
        1   2   3 ..   9

1:      1   2   3  9
2:      2   4   6 18
3:      3   6   9 27
:
9:      9   18  27 81
```

Notice that as well as the 9x9 table of products, you also need to display labels for the columns and rows.




```
c1 : for col from 1 to 9
c2 : for row from 1 to 9
c3 : for col from 1 to 9
```

So in the program itself the first thing to do is to output the column headings which involves a loop to display each column number. Then we can write the row loop which displays the row number as a label, then loops through all the columns in the row and displays the product, then goes to the next line. Formatting must be used so that the columns line up nicely and the whole effect is pleasing.

```
System.out.println("Multiplication Table");
System.out.println("--------------------");
System.out.println();

// output column headings
System.out.print("      ");           // space for row labels
for (int col=1;col<=9;col++)          // output column labels
{
  System.out.print(Formatter.format(col,5));
}
System.out.println();                 // go to next line
System.out.println();                 // leave a line

// loop through the rows
for (int row=1;row<=9;row++)          // for each row
{                                  // output row label
  System.out.print(Formatter.format(row,5)+":");
  for (int col=1;col<=9;col++)        // output product
  {
    System.out.print(Formatter.format(row*col,5));
  }
  System.out.println();               // go to next line
} // end of row loop
```

MultiplyTable.java

The output displayed by this program is

```
Multiplication Table
--------------------

          1    2    3    4    5    6    7    8    9

    1:    1    2    3    4    5    6    7    8    9
    2:    2    4    6    8   10   12   14   16   18
    3:    3    6    9   12   15   18   21   24   27
    4:    4    8   12   16   20   24   28   32   36
    5:    5   10   15   20   25   30   35   40   45
    6:    6   12   18   24   30   36   42   48   54
    7:    7   14   21   28   35   42   49   56   63
    8:    8   16   24   32   40   48   56   64   72
    9:    9   18   27   36   45   54   63   72   81
```

8.3 Loops using other datatypes

All the loops we've used so far have been written with integer loop variables. In fact you can write loops using any appropriate simple data type. In particular, real number loops and character loops can be useful at times.

A loop that will display the letters in the alphabet is

```
for (char letter='A';letter<='Z';letter++)
  System.out.print(letter+" ");
```

and a loop that will display all characters whose codes are between `'A'` and `'z'` (`AlphaLoop.java`) is

```
for (char ch='A';ch<='z';ch++)
  System.out.print(ch);
```

A loop that will sum all the numbers between 1.4 and 2.5 in steps of 0.1 (`RealLoop.java`) is

```
for (double num=1.4;num<=2.5;num+=0.1)
{
  System.out.println(num);
  sum = sum+num;
 }
System.out.println(sum);
```

However the output from this is somewhat surprising for 2 reasons: firstly the extra decimals that are output, and secondly, the fact that the loop stops at 2.4 instead of continuing to 2.5.

```
 1.4
 1.5
 1.6
 1.7000000000000002
 1.8000000000000003
 1.9000000000000004
 2.0000000000000004
 2.1000000000000005
 2.2000000000000006
 2.3000000000000007
 2.400000000000001
----
20.900000000000002
```

The reasons are that not all decimal fractions can be represented exactly in a finite number of bits, 0.1 being one of these values. So when 0.1 is added to `num` each iteration it is actually a value slightly larger than 0.1 that is being used which starts to have an effect after a few iterations. We can do away with the extra decimal places by using formatting to display only 1 decimal place, but that will not solve the second problem. The loop terminates early as a result of the additional value being added each iteration. The terminating condition is

```
num<=2.5;
```

but when we expect `num` to be 2.4 it is in fact slightly larger than 2.4, so incrementing by 0.1 will make `num` larger than 2.5, the check will return false and execution will cease one iteration too soon.

This illustrates that extreme care must be taken when dealing with decimal fractions and the programmer should always be aware of the potential for rounding errors. However, if the loop variable is a real number that is incremented by an integer amount then there is generally no problem.

```
for (double real=1.5;real<100;real++)
```

One thing you must **never** do with a for loop is change the value of the loop control variable yourself. The whole point to using a for loop is that you want to iterate a certain number of times, and that is what any other reader of your program expects. The for loop itself will update the value of the loop control variable and control the number of iterations. An example of what I mean is

```
// count in twos
for (int count=1;count<20;count++)
{
    count = count+1;
    System.out.println(count);
}
```

In this loop count is initialised to 1, and the loop body is executed which increments count to 2 and this is output. The for loop now increments count to 3, the loop body is executed and count is incremented to 4 … etc. The overall effect is to execute the loop 10 times and output the numbers 2, 4, .. 20. However, without studying the loop body closely the impression the reader gains is that this loop will run from 1 to 19 in increments of 1.

It does work, but it is **extremely poor programming**.

Exercises

7.1 Write a program to calculate and display the sum of the series

$$\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{N}$$

where the value N is input. Display your answer to 4 decimal places.

7.2 You have been told that the sum of a series of numbers 1+2+3+4+5+…+n can be calculated as n(n+1)/2. You want to check if this is in fact so. Write a program to read in some number N and to display the sum 1+2+3+…+N, and also the result of N(N+1)/2.

7.3 Write a program to read in N followed by N values, and to output each number, the running total, and the average value to date.

7.4 Write a program to input 2 real numbers and to output the squares and square roots of the numbers in the range from the first value in increments of 1. Make sure your output is neatly tabulated with suitable headings.

7.5 Write a program to input an integer number N, and to output all the bonds of N. For example, if N=5, your program should output

```
Bonds of 5
0 + 5 = 5
1 + 4 = 5
2 + 3 = 5
3 + 2 = 5
4 + 1 = 5
5 + 0 = 5
```

7.6 Write a program to display all the integers between 10 and 59 by generating the tens and units values separately. Your output should display the numbers 10..19 on the first line, 20..29 on the next, … etc.

```
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
 :
50 51 52 53 54 55 56 57 58 59
```

7.7 Write a program to input a size and to display an upside-down triangle of that size
Eg. for size = 4

```
****
***
**
*
```

7.8 Write a program to input a size and to display a triangle of that size with a straight right edge as shown below (size =4)

```
   *
  **
 ***
****
```

7.9 Write a program to draw a number of triangles N of a given size (N and size both input) across the page as shown (N=3, size=4)

```
*       *       *
**      **      **
***     ***     ***
****    ****    ****
```

Hint: This program involves 3 levels of loops - to draw a row of a single triangle - repeated N times across the page - for size rows

7.10 A useful rule of thumb is the "rule of 72" : if the annual interest rate is R% then a fixed sum of money will double in value in a period of 72/R years. (For example, if the interest rate is 12% it will take roughly 6 years for an investment to double).

Write a program to test the accuracy of this rule by tabulating, for each value of R from 1 to 36 the values of R, 72/R, and the actual time using the formula

$$\text{years} = \frac{2}{\ln(\frac{100+R}{100})}$$

Your results should be neatly tabulated with suitable headings, and all fractions should be rounded to 1 decimal place.

7.11 Write a program to output the words of the song

1 men went to mow, went to mow a meadow,
1 man and his dog, went to mow a meadow.
2 men went to mow, went to mow a meadow,
2 men,
1 man and his dog, went to mow a meadow.
3 men went to mow, went to mow a meadow,
3 men,
2 men,
1 man and his dog, went to mow a meadow
… etc

for up to 9 men, using 2 nested loops in your solution.

7.12 Write a program to output a table (to 1 decimal place) showing the fuel required for a journey of 100 to 1000 km (in 100 km steps) at speeds of 60, 70, 80, 90 or 100kph. Your output should be neatly tabulated with appropriate headings.

Fuel consumption (in km/litre) is given by

$$\text{consumption} = \frac{12}{1+\left(\frac{\text{speed}-80}{\text{speed}}\right)^2}$$

7.13 If an amount of money Amt earns R% interest over a period of P years, then at the end of that period the resulting amount will be

$$\text{total} = \text{Amt} \times \left(\frac{100+R}{100}\right)^P$$

Write a program to calculate and display a table showing the resultant amount if a sum of R1000.00 is invested at 10%, 12%, 14%, 16%, 18%, 20% for periods of 1 through 20 years. Your results should be neatly tabulated with suitable headings.

max, min, 2 max etc

7.1 In an ice skating competition, a mark between 0 and 10 is awarded by each of N judges (N>2). The mark awarded to a competitor is obtained by discarding the highest and lowest of the N marks and averaging the remainder. Write a program which inputs N followed by the N marks and outputs the mark obtained.

Extend your program by doing simple input validation. If N is 2 or less, don't input any marks. If a mark is negative use 0 instead, and if a mark is over 10 use 10 instead.

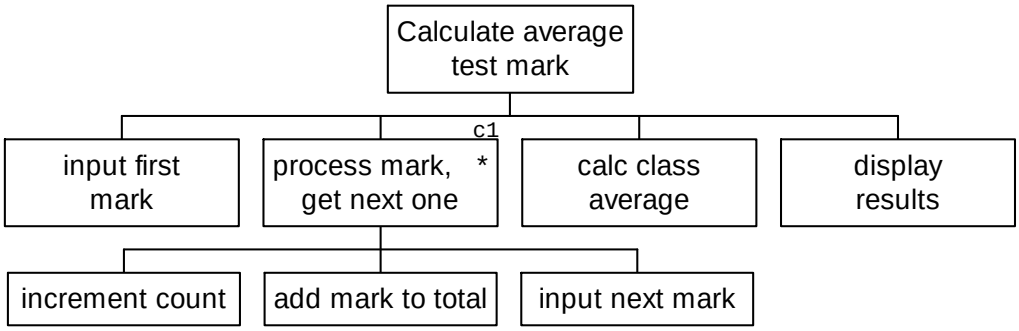
7.2 Write a program that will input a positive number and display all its factors. Try and make your program as efficient as possible.

9. Conditional Loops

Repetition or looping was introduced in chapter 7 - there we dealt with for loops that repeat a sequence of actions a pre-determined number of times. The other type of loops are those where you don't know in advance how many iterations are needed - they are determined by the state of events during the execution of the loop. Such **conditional loops** are characterised by having a condition that is tested, either at the beginning or the end of the loop, that determines whether or not to repeat the sequence of actions. The two forms of conditional loops are **while loops**, where the condition is stated and tested before the sequence of actions is executed, and **do-while loops** where the condition is stated and tested after the sequence of actions has been executed.

9.1 while Loops

Assume we need to read in and average a number of test marks but we don't know exactly how many students wrote the test. It is possible to count all the scripts, and write a program that reads in the number of tests and then uses a for-do loop to input and add up the marks, but this is tedious and prone to error. Another solution is to signal that the last test mark has been input by adding an extra dummy mark at the end that has an impossible value - say -999, and write a program that watches for that value and recognises that the end has been reached when that mark is input. Such a program using a while loop is:



c1 : while mark not -999

```
// Enter first mark
  System.out.print("  Enter mark for student 1 ");
  mark = Keyboard.getDouble();

  while (mark!=-999)
  {
    countSt++;     // increment count of students who wrote
    total += mark; // add mark to total
                   // get the next mark
    System.out.print("  Enter mark for student "+(countSt+1));
    mark = Keyboard.getDouble();
  } // end of while loop

  classAv = total/countSt;
```

AvMarkWhile.java

The program asks for the first mark to be input, and then commences the while loop which first tests that the mark is not -999 (ie. it is a valid mark) and then increments the count of students (needed to compute the average), adds the mark to the running total and asks for the next mark to be entered. The output from this program is

```
Calculation of average test mark for a class
--------------------------------------------
  (use -999 to signal the end of the marks)

  Enter the mark for student 1 > 50
  Enter the mark for student 2 > 60
  Enter the mark for student 3 > 70
  Enter the mark for student 4 > 80
  Enter the mark for student 5 > -999

The class average for the 4 students is 65.0
```

WHILE LOOP

while (*boolean expression*)
{
  body
}

This is interpreted as

"while the condition holds true execute the statements in the loop body".

The crucial point here is that the boolean expression is evaluated first and if it holds true the statements in the loop body are executed, then the condition is evaluated again and if it is true the body is executed again, and so on until the condition is false when execution continues with the next statement after the loop body. A while loop is a "test-at-the-beginning" loop, and the possibility exists that if the condition is initially false the loop will not be executed at all.

There are a number of requirements when using a while loop:

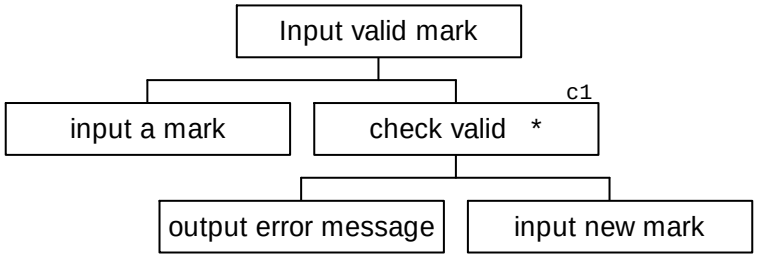
- the variables used in the boolean expression must be initialised so that a valid check can be done initially, otherwise the loop may be working on incorrect or undefined information.
  For example, in the `AvMarkWhile.java`, if the first mark is not input outside the loop then the test may fail initially because `mark` is undefined.
  And initialising `mark` to 0 when it is declared won't help either, because then an invalid value is used for the first mark. (`AvMarkWhileX1.java`)

- the values of variables used in the expression must be changed inside the loop body - if they are not changed the condition can never become false and the loop will never terminate. If the `getDouble` is omitted in `AvMarkWhile.java` then a new mark will never be input and the condition will be tested with the first mark each time, and because it is not -999 the loop will never end. (`AvMarkWhileX2.java`) *When running this program use the "stop run" icon at the right of the Kawa toolbar to stop the infinite loop*.

- the values of the variables are usually changed as the last statements in the loop body so that the new values are tested in the boolean expression before they are used to ensure that the loop conditions still hold. Asking for a new mark at the beginning of the loop before the statement to add the mark to the running total will mean that -999 will be included in that total. (`AvMarkWhileX3.java`)

- it must be possible for the loop to terminate eventually. If -999 is never entered then the program will continue to ask for marks indefinitely. Its often a good idea to output the condition for termination to assist the user.

Another common use of a while loop is for ensuring that only valid data is entered. To extend the previous example, assume that the mark entered must be a valid percentage in the range 0-100%, unless of course it is the sentinel value.



`c1 : while mark not -999 && (mark<0 || mark>100)`

When planning a conditional loop, you should always check that the requirements have been satisfied:

- condition variables are initialised: *yes - a mark is entered before the loop*
- condition variables are changed inside the loop: *yes - a new mark is input*
- condition variables are changed after any other processing in the loop: *yes - the last statement is the input statement*
- the loop will eventually terminate: *hopefully the user will enter a valid mark*

The Java code is

```
// input mark
  System.out.print("  Enter % mark for student "+num+" > ");
  mark = Keyboard.getDouble();

// check it is valid and keep asking to re-enter till it is
  while (mark!=-999 && (mark<0 || mark>100))
  {            // only gets here if it is invalid
    System.out.print("    invalid mark - please re-enter > ");
    mark = Keyboard.getDouble();
  }
```

part of `ValidMarkWhile.java`

The while loop tests the condition at the beginning and only enters the loop if the condition holds true, so the condition in this while statement tests for an invalid mark, and if it returns false (ie. the mark is valid) then the loop body is not executed, but if it returns true the loop body is entered and the user is prompted to re-enter the mark.

Running this program gives

```
Calculation of average test mark for a class
--------------------------------------------
  (use -999 to signal the end of the marks)

  Enter % mark for student 1 > 60
  Enter % mark for student 2 > -10
    invalid mark - please re-enter > 70
  Enter % mark for student 3 > 110
    invalid mark - please re-enter > -1
    invalid mark - please re-enter > 80
  Enter % mark for student 4 > -998
    invalid mark - please re-enter > 999
    invalid mark - please re-enter > -999

The class average for the 3 students is 70.0
```

The first mark input is 60 which is valid so the test is false and the loop body is not executed. The next mark is -10 which is out of range so the test returns true and the loop body is entered to display the error and allow the mark to be re-entered. This time a valid mark is input so execution continues with the next statements. The third mark entered is 110 which is out of range so the user is prompted to re-enter, again an invalid mark (-1) is input so the user is prompted to re-enter, and only once a valid mark is input does the program continue. Finally two more invalid marks are entered and rejected before the program accepts the sentinel value and the program ends.

A common mistake is to use an if statement instead of a while (`ValidMarkWhileX.java`):

```
if (mark!=-999 && (mark<0 || mark>100))
{       // only gets here if it is invalid
  System.out.print("    invalid mark - please re-enter > ");
  mark = Keyboard.getDouble();
}
```

This only allows a single invalid mark to be entered each time, and will accept the second mark entered without checking whether or not it is OK.

```
  Enter % mark for student 1 > -10
    invalid mark - please re-enter > 60
  Enter % mark for student 2 > -1
    invalid mark - please re-enter > 110
  Enter % mark for student 3 > -999

  The class average for the 2 students is 85.0
```

In this example, the invalid mark of 110 is accepted because no checking is done on the second value entered.

As another example of using a while loop, consider the problem of inputting any (positive) integer value and calculating the sum of its digits.

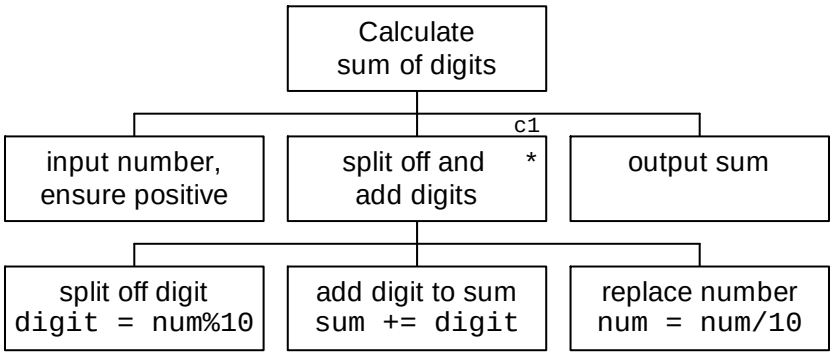
Eg. if 1234 is entered the program must calculate the sum 1+2+3+4 = 10

We cannot use a for-loop because we don't know how many digits there are in the number. The easiest way to solve this is to split off the low-order digit (eg. giving 123 and 4) and add it to a running total, and to keep doing this until all the digits have been split off in which case the number is left as 0.

Considering the number 1234, we will have

	number	digit	sum
initially:	1234		0
	123	4	4
	12	3	7
	1	2	9
	0	1	10

So the condition for our loop to keep going must be while the number is >0 , and in the loop we split off the low-order digit (remainder after integer division by 10), add it to the total, and replace the number with what is left (integer division by 10).




```
c1 : while number > 0
```

Check that the requirements have been satisfied:

- condition variables are initialised: *yes - a number is entered before the loop*
- condition variables are changed at the end of the loop: *yes - number is replaced with `num/10` as the last statement*
- the loop will eventually terminate: *yes - number is positive and is decreasing by a factor of 10 each iteration so will eventually reach 0*

```
// Enter number
     System.out.print("Enter an integer value > ");
     num = Keyboard.getInt();
     num = Math.abs(num);       // work with positive value

// loop to split off and add digits
     while (num>0)
     {
       digit = num % 10;     // split off low-order digit
       sum += digit;         // add it to sum
       num = num/10;         // replace num with remaining value
     }
     System.out.println("The sum of its digits is " + sum);
```

AddDigitsWhile.java

Depending on the requirements of the situation, a positive value can be ensured by using a while loop to enforce the input of a positive number (similar to the previous example), or merely using the absolute value of the number entered as is done here.

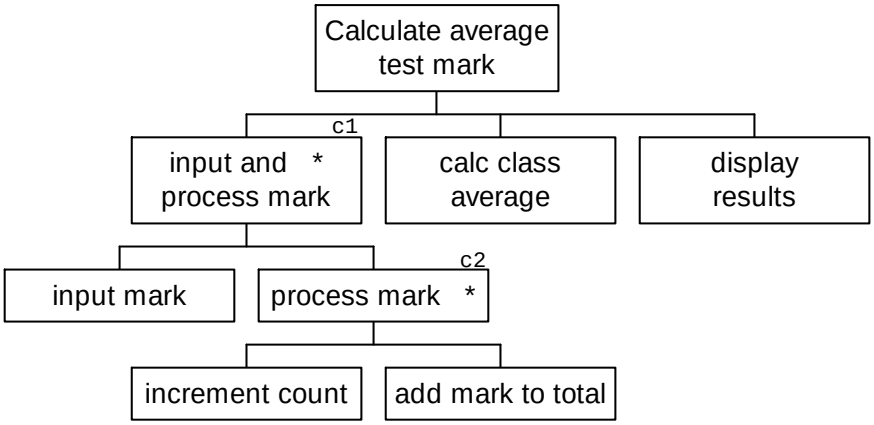
The output from this program is

```
Calculation of sum of digits
----------------------------

Enter an integer value > 520174
The sum of its digits is 19
```

9.2 do-while Loops

The conditional loop that tests the condition after the loop body has been executed is the do-while loop. It has the property that the loop body is executed before the condition is checked, and hence this loop is always executed at least once. This means that the condition variables do not need to be initialised before the loop, but because the condition is only checked at the end, care must be taken not to process unwanted values.

Consider the program we had initially to average a sequence of numbers terminating with a sentinel value:




```
c1 : do-while mark not -999
c2 : if mark not -999
```

The mark is input at the beginning of the do-while loop, but since it might be the sentinel value its processing must be guarded with an if statement to ensure that the sentinel value is not included in the total.

```
// Loop to process marks
  do
  { System.out.print("  Enter mark for student "+(countSt+1));
    mark = Keyboard.getDouble();

    if (mark!=-999)// if not sentinel value
    {
      countSt++;   // increment count of students who wrote
      total += mark;   // add mark to total
    }
  } while (mark!=-999); // check condition and end if false

  classAv = total/countSt;
```

part of `MarkAvDo.java`

The output from this program is:

```
Calculation of average test mark for a class
--------------------------------------------
  (use -999 to signal the end of the marks)

  Enter the mark for student 1 > 80
  Enter the mark for student 2 > 50
  Enter the mark for student 3 > 50
  Enter the mark for student 4 > -999

The class average for the 3 students is 60.0
```

Alternatively, the program could be structured similarly to that used with the while loop, with initialisation outside the loop and the next mark input at the end of the loop body, but in this case you must be certain that at least 2 marks are input, one valid mark and the sentinel value, since the loop will always be executed at least once (`MarkAvDo2.java`). It seems far safer just to use a while loop.

The format of a do-while loop is

DO-WHILE LOOP
do {   *body* } **while** (*boolean expression*)

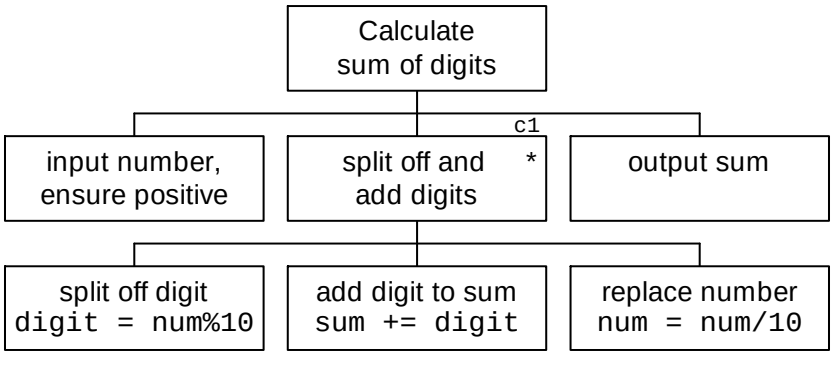
Note that the curly brackets { and } are required even if the body consist of just one statement.

The requirements when using a do-while loop are similar to those for a while loop:

- the values of variables used in the expression must be changed inside the loop body - if they are not changed the condition can never become false and the loop will never terminate.
- it must be possible for the loop to terminate eventually.
- care must be taken to ensure that the loop can validly execute at least once, and that terminating values of condition variables are not inadvertently processed.

Considering the second while loop example - ensuring valid marks are input - it does not make sense to use a do-while loop because if valid data is entered you do not want to execute the loop body even once to ask for the data to be re-entered. A while loop is far better suited here.

The third while loop example - adding the digits of a number - can be successfully rewritten as a do-while loop, since in most cases a non-zero value is entered so we need to execute the loop body at least once, and even if zero is entered the digit split off is zero so the correct answer (0) is still obtained:



c1 : do-while number > 0

```
// loop to split off and add digits
  do
  {
    digit = num % 10;       // split off low-order digit
    sum += digit;       // add it to sum
    num = num/10;       // replace num with remaining value    }
while (num>0);
```

part of `AddDigitsDo.java`

Exercises

9.1 Write a program that inputs a number of single digits and combines them into an integer value. The end of the digits is signalled with a negative value.
For example, inputting

2
3
4
-1

would result in the value 234 being output.

9.2 Write a program to determine how many terms of the series

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots$$

are needed before the total first exceeds 5.

9.3 Modify your solution to exercise 8.10 (ice skating scores) to ensure that the number of judges N is input as >2, and that all the scores are input as numbers between 0 and 10.

9.4 A common test for divisibility by 3 is to add the digits of a number, then the digits of the sum, then the digits of this sum, … and so on till a single digit sum is obtained. If this value is 3, 6, or 9 then the original number is divisible by 3. Write a method that uses this test to determine whether the integer value passed as parameter is divisible by 3. Then use this method in a program that inputs a number and determines whether it is divisible by 3 or 6 (even and divisible by 3).

9.5 An iterative formula for estimating the square root of a positive number `value` is

$$guess_{n+1} = \frac{1}{2}(guess_n + \frac{value}{guess_n})$$

Write a program that reads in `value` and uses this iterative method to find the square root. The iteration should continue until the difference between successive approximations is $<10^{-3}$. Your program must ensure that `value` is positive. Use `value`/2 as your initial guess.

9.6 Write a method `IsaPrime` that determines whether or not the integer number passed as parameter is a prime number (has no factors other than 1 and itself). Use this method in a program to display all prime numbers between 1 and 100.

9.7 Using your `IsaPrime` method, find the first value of `n` for which n^2`-n+41` is not a prime. (n= 0,1,2,…)

9.8 The highest common factor (HCF) of two positive integers is defined as

HCF(A,B)	=	A	if A=B
	=	HCF(A-B,B)	if A>B
	=	HCF(B,A)	if A<B

So to find the HCF iteratively, repeated subtract the smaller number from the larger number until the two become equal.

Write a program that inputs 2 positive numbers (ensure they are positive) and calculates their HCF as described above.

10. Classes and Objects

Object oriented programming focuses on a program as a group of inter-relating objects or "things". With the procedural, top-down approach, we focused on what needs to be done and how to do it. While this still has a role to play in defining the functionality of objects, the main emphasis in object oriented programming is what are the objects, what are their properties or attributes, what behaviour or capabilities do they have, and how do they inter-relate and interact with other objects.

10.1 An introduction to objects, classes, members and constructors

To start with, what is an object? Objects are all around us. Things such as people, cars, pencils, clocks, books … are all objects. Each of these has certain properties or attributes, each has the ability to perform certain actions and these actions may have an effect on other objects in the world. Object oriented programming is a style of programming that views a system as consisting of a number of objects that interact with each other. The role of the programmer is to define the objects in a system, identify their attributes and the actions they can perform and how they interact with each other. Object-oriented programming is about designing objects that all work together to accomplish a particular task.

In procedural (top-down) programming the approach is to tackle a problem by breaking it down into its component subproblems, and then to write a procedure or method to deal with each of these. Its intuitive and orderly and effective for solving complex problems. The focus is on “what needs to be done”, and then “how do we do it”.

In object oriented programming the system is viewed as consisting as a number of interacting objects and the primary focus in on the definition of the entities or objects in the system. We need to consider what they are like and what they can do. Each object is an instance of self-contained entity (class) with its own properties and behaviours. We then write an application program that uses these objects to achieve some result or effect. The application program creates (instantiates) the objects and specifies how they must interact.

When designing objects we more correctly are designing **classes**. A class is a blueprint or template which defines the **general characteristics** of a class of objects. On the other hand, an object is a concrete realisation of a particular instance of a class.

For example, consider a Car class.
The properties (attributes) of a car may include

- its make and model,
- its colour,
- its engine capacity,
- its maximum speed,
- the capacity of its fuel tank
- its odometer reading,
- its mass ……etc

The behaviours or functionality it may have include the ability

- to calculate its fuel consumption,
- to calculate the time it will take to travel a certain distance,
- to update its odometer reading,
- to determine how far it can travel before it runs out of petrol …… etc.

These would be the attributes and methods defined for a general Car class. Each instance of a car will have its own particular values for each of these attributes.

Assume we need to have three cars in a program. We would then create (instantiate) three car objects using our general Car class as a template, and supply them each with their individual attribute values.

```
Car myCar = new Car(“Renault Megane”,silver,1600,195,...);
Car hisCar = new Car(“Toyota Hilux”,red,3000,180,...);
Car mikesCar = new Car(“Ferrari F2002”,red,3000,500,...);
```

The three objects all have exactly the same kinds of attributes each with their own unique values, and have exactly the same kinds of functionality although the results will vary depending on the individual attribute values (for example, fuel consumption is dependant on engine capacity, distance before it runs out of petrol is dependant on fuel consumption and petrol tank capacity.)

When designing classes of objects you don’t need to include all possible attributes of the object – only those that a relevant to the task at hand or may be useful if the object is to be extended. There should always be some form of identification attribute, even if it doesn’t appear to be immediately required. For example, considering the car Class above, other possible attributes are purchase price, number of gears, boot capacity, but the last two are unlikely to be relevant in a fleet management system where the user is concerned with the costs of owning vehicles, but may be relevant in a car dealer system where information about the physical characteristics of each car is needed. You have to make a value judgement about what is relevant and what is not for each class you are designing.

There are a number of important characteristics of object oriented programming:

- **Encapsulation**: objects are self-contained modules containing the information and the means to perform given tasks. Each object knows the values of its own attributes and how to perform its own functions.
- **Interface**: each object must know exactly what its interface is and how it interacts with the outside world. It knows what information it can receive from outside (and what to do with it), and it knows what information it can make available to the outside world and how to do so.
- **Information hiding**: details of internal data and variables and exactly how it accomplishes its task are hidden from the user. All object design should be on the principle of “need to know”. This enables objects to be written in the most efficient manner (which is not necessarily the most user-friendly, hence a good interface is essential), and for this internal representation to be changed should circumstances warrant it without requiring every program that uses the object to be rewritten.
- **Generality**: objects should be designed to be as general as possible so they can be used in a number of different situations. In other words, objects should be designed not for a particular task but for a particular kind of task.
- **Extensibility**: Objects can be extended or refined to handle related or more specialised tasks (inheritance).

Object oriented programming isn't just applicable to large complex systems consisting of a number of interacting objects. The principles of object orientation can be applied to simple problems of the kind dealt with in introductory programming classes.

Recall an example to find the roots of a quadratic equation, $ax^2 + bx + c = 0$ using

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

The earlier approach focused on the processes that were necessary - input the coefficients, calculate the roots, display the result. In contrast, an object oriented approach focuses on the objects, their attributes and behaviours.

What is an object in this case? A quadratic equation, of course.
What are its properties or attributes? Its coefficients.
What behaviour or capabilities should it have? The ability to display itself; the ability to determine whether it has real roots; the ability to calculate and display these roots.

(There are obviously many possibilities and permutations - for example the roots may be considered an attribute, we may wish the object (quadratic equation) to be able to make its various attributes available for something else to use or display.)

The quadratic equation class must describe the properties and capabilities of the objects the program has to deal with and serves as a "blueprint" or description of all quadratic equations - that there are three coefficients, that the way to check for roots is to examine the discriminant, and that roots are calculated using this expression. Then for each specific quadratic equation we have a quadratic equation object, which is a concrete realisation of the class, and has actual values of the coefficients. We can have many quadratic equation objects, all instances of the quadratic equation class.

In our quadratic equation class we need to provide space for attributes that will store the values of the coefficients, and we need to write methods to carry out the functions required of the class. This class is an independent, stand-alone entity, that defines an "object-type" that can be used in other programs (classes). Other classes that use this quadratic equation class will create instances of quadratic-equation objects that have the full functionality of the class - in other words, each instance of a quadratic equation that is created can store its coefficients, display itself, determine how many real roots it has, and calculate them.

The quadratic equation class is shown on the next page:

```
class QuadEq
/* The quadratic equation class
 * ---------------------------
 * that will display itself, determine the number of roots,
 * and calculate the roots if not imaginary
 */
{
// attributes
//-----------
  private  double a,b,c;    // coefficients  of the equation

  QuadEq (double aa, double bb, double cc)
//------
// constructor - stores coefficient values in the attributes
  { a = aa;
    b = bb;
    c = cc;
  }

  void Display()
//--------------
// Displays the equation
  {
    System.out.print(a+ "x^2 + " +b+ "x + " +c+ " = 0");
  }

  int NumRoots()
//-------------
// determines number of real roots
  {
    double disc = b*b-4*a*c;
    if      (disc < 0)   return 0;
    else if (disc == 0)  return 1;
    else             return 2;
  }

  double GetRoot1()
//-----------------
// calculates one root of the equation.
  {
    if (NumRoots()>=0)
      return ((-b + Math.sqrt(b*b - 4*a*c))/(2*a));
    else
      return 0;
  }

  double GetRoot2()
//---------------- calculates the second root of the equation
  {
    if (NumRoots()>=0)
      return ((-b - Math.sqrt(b*b - 4*a*c))/(2*a));
    else
      return 0;
  }
}
```

class QuadEq *in* SolveQuadEg.java

The class SolveQuadEq performs the task of inputting the values of the coefficients for a particular equation, creating an instance of a quadratic equation with those coefficients, displaying this equation, checking whether it has real roots and if so, displaying them.

```
class SolveQuadEq
//---------------
/* The driver class that inputs the values of the coefficients,
 * creates a quadratic equation object with these coefficients,
 * displays it, checks for real roots, and displays them.
 */
{
  public static void main(String[] args)
  {
    double aa,bb,cc;    // the 3 numbers to be input

// input three values
    System.out.print("  Enter coefficient of x^2 (a) > ");
    aa = Keyboard.getDouble();
    System.out.print("  Enter coefficient of x   (b) > ");
    bb = Keyboard.getDouble();
    System.out.print("  Enter constant           (c) > ");
    cc = Keyboard.getDouble();
    System.out.println();

// instantiate and use QuadEq class

    // create object called myEq
    QuadEq myEq = new QuadEq(aa,bb,cc);

    System.out.print("The quadratic equation ");
    // invoke the class's Display method to display myEq
    myEq.Display();

    // invoke the class's Numroots method to determine
    // how many roots myEq has
    switch (myEq.NumRoots())
    {
      // no roots - display a message
      case 0 : System.out.println("\nhas no real roots");
               break;

      // one root - invoke GetRoot1 method to calculate it
      case 1 : System.out.println("\nhas one real root, " +
                 myEq.GetRoot1());
               break;

      // two roots - invoke GetRoot1 and GetRoot2 methods
      case 2 : System.out.println("\nhas two real roots, " +
                 myEq.GetRoot1() + " and " + myEq.GetRoot2());
               break;
    }

    System.out.println();
  }
}
```

SolveQuadEq.java

An instance of a quadratic equation object is created in the statement

```
QuadEq myEq = new QuadEq(aa,bb,cc);
```

The first `QuadEq` indicates that the object is of class `QuadEq` (in the same way as we declare an integer variable using `int num;`). The identifier chosen for this instance of the class is `myEq`, and the `= new` is the signal to create a new instance of the class type `QuadEq`, and to provide 3 values to that instantiation process, in this case the values of the coefficients. In most cases the class used for declaring the object is the same as the one for creating the instance, although there are times when it need not be so.

In this example we only need a single quadratic equation so only one object, `myEq`, is instantiated. If it was necessary to have a number of equations they would be instantiated and given unique identifiers

```
QuadEq myEq = new QuadEq(1,-5,6);
QuadEq yourEq = new QuadEq(2,-6,5);
QuadEq herEq = new QuadEq(1,1,-6);
```

Each object instantiated has all the attributes and capabilities of its parent class. The attributes are known as **fields**, and can be variables or other objects; the capabilities are known as **methods**. Both the fields and methods of a class are called it **members**.

Methods of a class can reference and manipulate the fields of that class, and can also declare their own fields which are then local to that method and are not accessible to other methods, even in the same class. For example, `NumRoots` has a local field, a `double` variable called `disc`. Methods may have their own inner classes, declared for their own specific purpose, but may not have their own local methods.

As another example of a simple class, consider a `Time` class. This class has fields, `hour` and `min`, defined using the 24 hour clock, and methods to set the time, to display the time in either 12 or 24 hour format, and to update a time by a number of minutes.

```
class Time
/* The time class
 * --------------
 * can be set to a specific value, will display itself in
 * 12 or 24 hour format, and update itself by a number of mins
 */
{
// attributes
//-----------
  private  int hour,min;         // the hour and minute values

  void SetTime(int h, int m)
//-------------
// sets the current time to h hours and m mins.
// Ensures they are always valid values
  {
    hour = h;
    min = m;
    MakeValid();
  }
```

```
  void Update(int m)
//------------
// Updates the time by m minutes
  {
    min = min+m;
    MakeValid();
  }

  void Show12()
//-------------
// Displays the time in 12 hour format
  {
    char amORpm;

    if (hour<12) amORpm = 'a';
    else         amORpm = 'p';
    if (hour>0 && hour<=12)
      System.out.print(hour + ":");
    else if (hour>12)
      System.out.print(hour-12 + ":");
    else // hour==0
      System.out.print("12:");
    if (min<10) System.out.print("0");
    System.out.print(min + " " + amORpm + "m");
  }

  void Show24()
//-------------
// Displays the time in 24 hour format
  {
    if (hour<10) System.out.print("0");
    System.out.print(hour + "h");
    if (min<10) System.out.print("0");
    System.out.print(min);
  }

  private void MakeValid()
// private method that checks that the hour is between 0-23
// and the mins between 0-59 and adjusts if necessary
  {
    while (min>59)
    { min = min-60;
      hour = hour+1;
    }
    while (min<0)
    { min = min+60;
      hour = hour-1;
    }
    while (hour>23)
      hour = hour-24;
    while (hour<0)
      hour = hour+24;
  }
}
```

class Time *in* TimeDemo.java

The driver program that uses this class is very straightforward, and merely instantiates two time objects, `lecture` and `lunch`, displays them when are created, then sets them to specific values, displays them, updates them and displays them again.

```
class TimeDemo
//---------------
/* The driver class to demonstrate using the Time class.
 * It creates two time objects, sets them to different times,
 * displays one in 12 hour and the other in 24 hour format,
 * updates them by different amounts and displays them again
 */
{
  public static void main(String[] args)
  {

// display a heading
    System.out.println();
    System.out.println("Playing with time");
    System.out.println("-----------------");
    System.out.println();

// create time objects
    Time lecture = new Time();
    Time lunch = new Time();

// invoke the show methods to display the initial contents
    System.out.print("Times as created: " + "\n  lecture is ");
    lecture.Show24();
    System.out.print(" and lunch is ");
    lunch.Show24();
    System.out.println("\n");

// set lecture time to time of first lecture
    lecture.SetTime(7,45);
    System.out.print("First lecture starts at ");
    lecture.Show12();
    System.out.print(" and ends at ");
    lecture.Update(45);
    lecture.Show24();
    System.out.println();

// set lunch time to time of lunch break
    lunch.SetTime(12,10);
    System.out.print("Lunch break is at ");
    lunch.Show12();
    System.out.print(" and ends at ");
    lunch.Update(55);
    lunch.Show24();
    System.out.println();
  }
}
```

TimeDemo.java

The methods of a class define its capabilities, and are available for every object created from that class. We need to be able to specify which object we want a method to apply to - for example, which time to display. This is done by prefixing the method with the name of the object whose field we want to access.

Hence we have

```
lecture.SetTime(7,45);
lunch.UpdateTime(55);  etc.
```

In the first case the `SetTime` method will store the values of its parameters in the variables `hour` and `min` of the lecture object, which are quite distinct and separate from the `hour` and `min` variables of the `lunch` object. And even though we only used one quadratic equation object in the previous example, we still need to specify precisely which object we are using and prefix the method with the object name

```
myEq.NumRoots();
```

When a program is executed and the declaration of an object is reached, that object is created. At this point a special method called a **constructor** is called automatically by Java. If no constructor has been defined then all that happens is space is reserved for the objects data values (class `Time` had no constructor). If a constructor has been defined then its statements are executed. A constructor has the same name as the class, for example, in class `QuadEq` a constructor,

```
QuadEq (double aa, double bb, double cc)
```

was defined.
When the object `myEq` was created,

```
QuadEq myEq = new QuadEq(aa,bb,cc);
```

the values supplied were used by the constructor to initialise the data variables.

A class may have more than one constructor, all with the same name as the class but with different numbers and/or types of parameters. Then when the object is declared, the constructor whose parameter list matches the arguments supplied, is used. For example, class `Time` could have had the following constructors defined:

```
  Time()
//------
// constructor - initialise to default value - midnight
  {
    hour = 0;
    min = 0;
  }

  Time (int time24)
//----------------
// constructor - initialise time using the 24 hour format
  {
    int hh,mm;

    hh = time24/100;        // separate out hours and minutes
    mm = time24 % 100;
    SetTime(hh,mm);        // use this method as it stores the
  }                     // values and checks they are valid
```

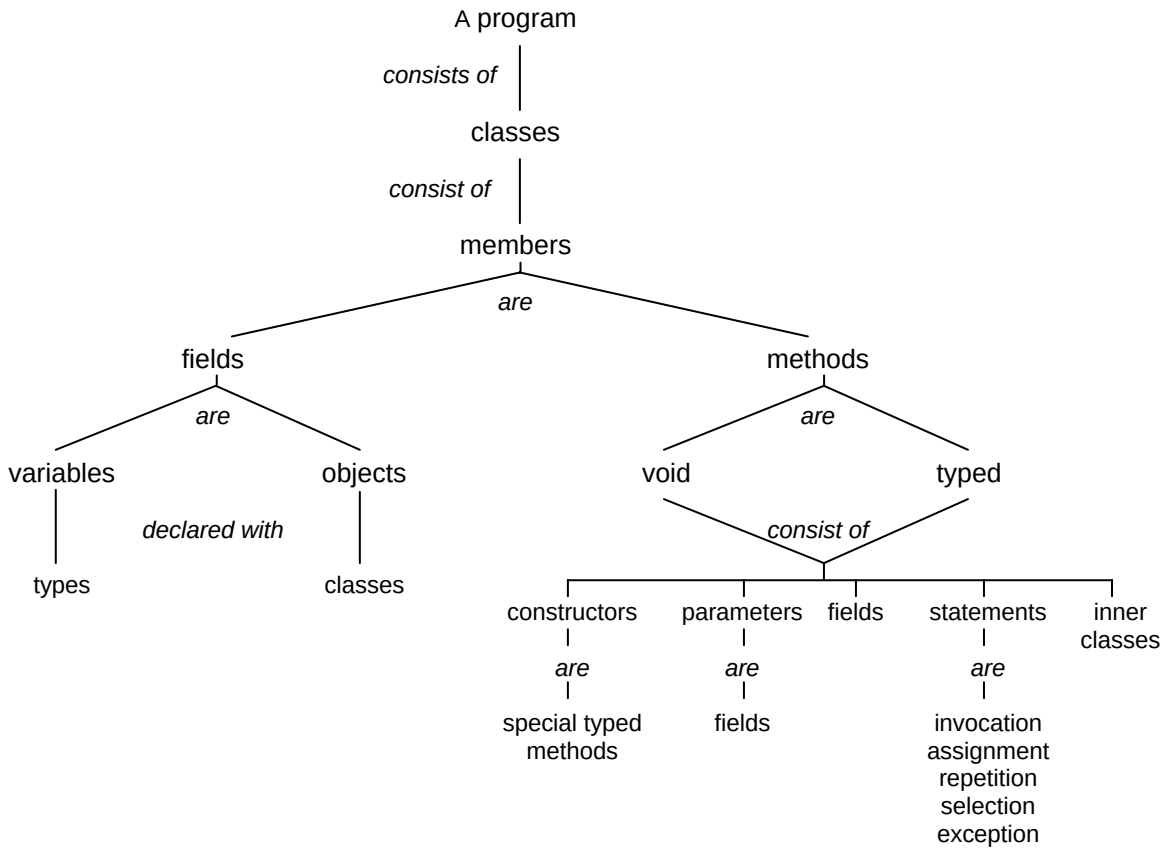
```
  Time (int hh, int mm)
//---------------------
// constructor - provide initial values for hour and min
  {
    SetTime(hh,mm);// use this method as it stores the
  }                    // values and checks they are valid
```

and the statements to create the time objects could be

```
Time lecture = new Time(2035);
Time lunch = new Time(06,75);
```

The first statement to declare `lecture` supplies a single integer value so would use the second constructor, while the declaration of `lunch` uses the third constructor and supplies two integer values.

The following diagram which clearly illustrates the terminology for the parts of a program is from the book *Java Gently*, by Judy Bishop:



10.2 Java packages, classes and objects

Java itself is a relatively simple language, but in addition to the built in data types, keywords and controls (the base language) it also provides a large number of ready-built classes which can be used by any program. When writing a program, rather than having to write every class you might need from scratch, many can be taken from class libraries or packages. This saves considerable time and effort, and also gives you well designed and thoroughly tested classes and objects.

The Java programming language and its packages are intimately connected - the language cannot be used without the packages and some classes are directly recognised and used by the Java language. For example, the **`lang`** package contains the classes `Math` (with methods `Math.sqrt; Math.pow` and many others) and `System` (with field `System.out` which is an object of class type `PrintStream` which has method `println`).
(Hence the statement

```
System.out.println();
```

actually means

"execute the `println` method with the `out` object (the standard output device) of the `System` class".)

Because the `lang` package is fundamental to the Java language it is always accessible, but other packages (including those you or someone else may have written) need to be specifically made available to your program.

PACKAGE ACCESS
`import` java.*`package`*`.*;`
`import` java.*`package.class`*`;`
`import` *`mypackage`*`.*;`

To use one of the Java class libraries we use one of the first two forms. The asterisk indicates that all classes in the package should be accessible, or we can be more selective and choose just one by specifying the class name.
Eg.

```
import java.util.*;
```

or

```
import java.util.Random;
```

Similarly for packages written yourself, or by another user
Eg.

```
import Utilities.*;
```

or

```
import Utilities.Keyboard;
import Utilities.Formatter;
```

(`Utilities` is a package written by Prof Sartori-Angus to provide for simple keyboard input and formatting of output.)

Java only imports the parts of the package that are actually used and there is no performance degradation in importing the whole package using `*`.

To become a competent Java programmer, you will need to become familiar with the packages and classes available. The presence of so many classes is intended to, and does,

encourage reusability. There is no point in re-inventing a class that already exists or that can be extended (more later). The supposed down-side is that programmers have to spend time learning the classes, but the packages are an integral part of the language and Java is not just the base language. The documentation on all the Java packages is provided as online and fully cross-linked and indexed HTML documents. For example, the `Date` class in the `util` package is designed to represent dates and times. An extract from the HTML documentation is shown below:

public class Date

The class `Date` represents a specific instant in time, with millisecond precision.

CONSTRUCTORS

- `Date()`

Allocates a Date object and initializes it so that it represents the time at which it was allocated measured to the nearest millisecond.

- `Date(long)`

Allocates a Date object and initializes it to represent the specified number of milliseconds since January 1, 1970, 00:00:00 GMT.

METHODS

- `after(Date)`
  Tests if this date is after the specified date.
- `before(Date)`
  Tests if this date is before the specified date.
- `compareTo(Date)`
  Compares two dates for ordering.
- `compareTo(Object)`
  Compares this date to another Object.
- `equals(Object)`
  Compares two dates for equality.
- `getTime()`

Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this date.

- `setTime(long)`

Sets this date to represent the specified number of milliseconds since January 1, 1970 00:00:00 GMT.

- `toString()`
  Creates a canonical string representation of the date.

As well as listing the fields and methods, further detail of their declaration and function is given, eg

`setTime`

```
public void setTime(long time)
```

Sets this date to represent the specified number of milliseconds since January 1, 1970 00:00:00 GMT.

Parameters:
`time` - the number of milliseconds.

A simple program that uses this class is shown here.

```
/*
* Demonstration of Java's Date class
* ----------------------------------------
*/
import java.util.*;

public class DateDemo
{
  public static void main(String[] args)
  {
  // create a new Date object
    Date now = new Date();

  // output current time
    System.out.println("\nCurrent time is " + now);

  // update by a billion milliseconds
    long newTime = now.getTime() + 1000000000;
    now.setTime(newTime);

  // output updated time
    System.out.println("Updated time is " + now);

    System.out.println();
  }
}
```

DateDemo.java

The import statement specifies that the `util` package must be available so that its `Date` class can be used. An instance of a `Date` object is created (`now`), and displayed. Println automatically calls the `toString` method to convert `now` to a string which it can output. Then the `getTime` method is called to get the time value stored for `now`, a value is added to it and `setTime` is called to store this new value for `now`. Finally `now` is displayed again.

The output from this program is

```
Current time is Thu Nov 19 08:58:48 GMT+00:00 1998
Updated time is Mon Nov 30 22:45:28 GMT+00:00 1998
```

Originally the `Date` class was more complex and had methods for getting and setting hours, months etc individually. These functions have now been taken over by the `Calendar` class which we will look at in more detail later.

10.3 Designing classes

Methods are a means of grouping statements, while classes are a means of grouping methods. In particular, we group together methods that serve a common purpose or are concerned with providing the same type of function. For example, the `Math` class in Java provides mathematical methods, the `Date` class provides methods for representing dates and times, and the `QuadEq` class written earlier in this chapter serves the purpose of solving quadratic equations.

The design of a class in terms of the methods and data it provides is the cornerstone of object-oriented programming, and central to the way Java is intended to be used. When designing classes, there are some very important considerations to be borne in mind:

- **Cohesion**. A class should be concerned with a single physical entity or set of similar operations. For example, the `Calendar` class manipulates dates and times.

- **Separation of concerns**. Even for a single entity you can have several related classes rather than one class, for example, `Date` merely caters for storing and comparing dates, `Calendar` focuses on their representation and manipulation of the component parts, and formatting and output of dates is handled in another class, `DateFormat` in the `text` package.

- **Information hiding**. A class should reveal to its user only that which it needs to reveal and no more. In this way the data can be protected from misuse, and the class can operate on a secure basis. Methods can also be hidden from the user if they are for use only within the class itself. In my example class `Time`, the hour and minute fields (`hour,min`) are declared as private to prevent unauthorised access by a user of this class, and the method `MakeValid` is also private as it is not needed by users of the class but provides a function within the class in ensuring that only valid data is stored in the `hour` and `min` fields.

- **Data access via methods**. This follows on from the previous point, in that if a data field is hidden from the user to prevent inappropriate changes, there must be some means provided so that the user can access it if necessary. Most classes provide `get` and `set` methods for retrieving and storing values in its data fields. This forces the user to access the fields using the provided methods only, and preserves the security and integrity of the data fields. Examples are the `SetTime` method in my example `Time` class (notice the call to private method `MakeValid` to ensure that the values stored are in range), and `Date`'s `getTime` and `setTime` methods.

- **Object initialisation**. When an object is created, it is efficient to set up initial values so these can be used later. These can either be default values, or values supplied as parameters by the user. These values are then always available with the object and do not have to be supplied repeatedly as parameters. For example, when an instance of a `Date` is created it is initialised with the current date and time. If you allow an object to be created without initial values and rely on the user to set the values the possibility exists that the object may be used before values have been stored.

- **Data Integrity**. It is a basic premise when dealing with objects that they should hold valid data. This is something that you as designer of a class must ensure so that the users (and the methods you write for the class) do not have to keep checking the data values before they are used - the mere fact that they are stored in the class fields should imply they are valid. For example, a date such as 31 Feb should not be stored. Such attempts to store invalid data should be rejected - possibly by returning error codes or setting error flags - and default values stored so that if the user does not check his errors at least the programs will not crash because of illegal values. This is not always relevant - for example, all values for the coefficients in `QuadEq` are possible.

- **Program defensively**. Defensive programming involves anticipating errors and catering for them. (*"If anything can go wrong, it will".*) You need cover all situations to ensure that every part of the program behaves sensibly, even in the face of unexpected circumstances. One of the ways to achieve this is to work on a principle of minimal access - just allow the user access to what they need, don't create unnecessary variables, control the access to class attributes and make available only those methods the user needs to use.

Bearing these points in mind, how do you go about writing a program using classes?

1. Determine the User Requirements

This involves finding out exactly what the program is supposed to do. There is a strong temptation to jump straight in and start writing code - resist it or you could end up with a program that doesn't do what it is supposed to and then have to try and modify it which often ends up as a complete mess.

You need to think carefully about the program, how it is likely to be used, what output is expected and what inputs will be needed to produce this output, what extensions might be required, what generalisations are possible

2. Analysis

This is about building a model of the system using classes and class relationships. This involves identifying the classes that will be needed by the program. How do we go about identifying classes? By a process of brain-storming, applying common sense, and searching the requirements. For large scale systems some more formal analysis approach may also be used, but these are usually adequate for small-scale problems.

The classes we want are those that represent the key abstractions in the program - the things, entities, roles, strategies and data structures. For each class, a list of attributes and public methods need to be identified. The relationships between classes must also be considered.

A class diagram which gives a pictorial view of the classes and their relationships is a useful output from this phase.

3. Design and Coding

Design is about refining the results of analysis to the point where code can be written, while coding is the act of actually writing the code. In large-scale systems these are treated as completely distinct steps, but with small programs these are often rolled into one step. In effect, the design is documented by writing and commenting Java source code.

The design process involves writing a Java class for each design class, deciding on a type for each attribute, declaring each method including parameters and the return type, and then writing each method body to achieve its specified purpose. As code is written, design decisions and implementation choices and decisions should be commented in the code where needed. Before putting effort into writing a class it is worth checking whether there is an existing class, either from a library or from a previous program, that can be used.

4. Testing

Classes need to be tested to ensure they behave as expected. This can often be conveniently done by giving each class a static main method specifically for testing the behaviour of the class. Testing should try to initialise new objects and check they have a valid state, and call each method with appropriate arguments and check the results.

5. Review and iterate

These steps all take place in the framework of iterative development. When you design classes, you may discover an error or oversight in the analysis stage; when you code you may realise that something is wrong with the design; and naturally, testing may expose flaws in analysis, design and coding. When correcting errors it is important to consider the impact of the correction on the full program - too often fixing one error will cause another. In some cases, a full redesign and rewrite may be needed; in others the problem may be more localised.

A useful approach is to start the coding and testing phases simply - develop a minimal program, maybe just initialisation and output (*set* and *get*), and make sure that works correctly, then add functionality bit by bit until the entire system is developed.

Even when the program appears complete it is important to pause and review it in conjunction with the user requirements - does it actually achieve what is required? Often the full implications of the requirements and design do not become apparent until an attempt at a solution has been made.

1.4 Examples

As an example of an object oriented design, consider another program we had before (*Intro to Java*) to calculate the yield of grain from a field:

> A farmer has a rectangular field that is L metres long and W metres wide. He grows maize in this field, and his costs (seed, fertilizer, labour) are Rand C per square metre. He gets a yield of Y kg of maize per square metre. If maize is currently selling for Rand M per tonne (1000 kg), what profit does he get from this field?
> Write a program that reads in values for L, W, Y, C and M and determines his profit.

The requirements are clearly specified here, so the first step is to identify the classes, their attributes and behaviour.

We can use a class to represent the field.
What are its attributes? - its length and width are obvious choices.
And methods? - to calculate and return the area in square metres.
What about setAttribute and getAttribute methods? - the field size is not going to change so the values need only be set at initialisation, and there appears to be no need in this application to access the field dimensions.

Field
| length |
| width |
getArea()

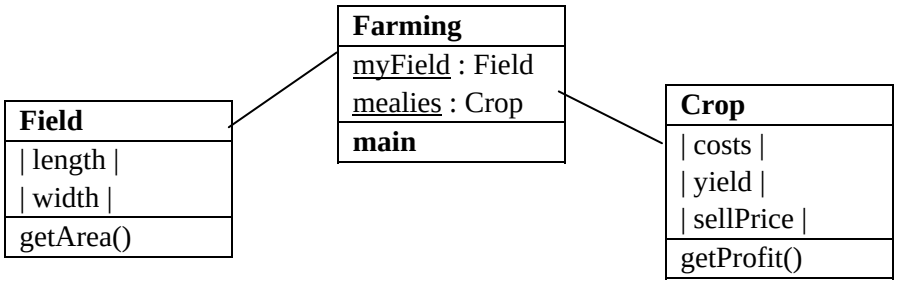
Another class in this example is maize - or more generally, the crop.
Attributes? - costs, yield, selling price
Methods? - to calculate and return the profit per square metre
setAttribute and getAttribute methods? - for this simple application not needed, although one can see a case for them in a larger application.

Crop
| costs |
| yield |
| sellPrice |
getProfit()

The coordinating class (our program), has then to create a field object (`myField`), create a crop object (`mealies`), and use them to determine the profit from growing that crop in the field.

Farming
myField : Field
mealies : Crop
main

The class diagram gives a pictorial view of the classes and their relationships, and is very helpful in seeing the overall picture.



The notation used here for depicting classes is a modification of the UML notation. It uses rectangles to denote classes, with connecting lines and annotations to denote class relationships. Within a class rectangle, the first section gives the class name, the next section lists the attributes and the last section lists the methods. Private fields and methods are enclosed in vertical lines (eg | costs |), and objects are underlined and their classes are given.

The complete program thus consists of 3 classes - the crop class, the field class and the co-ordinating class which contains the main method.

For the field class, we need to decide what data types our attributes need to be, write a constructor that will store the values supplied when the object is instantiated in the object's attributes, and design and write a method that will use the attributes to determine the area of the field. In this case this is trivial - multiply length by width.

```
class Field
/* The field class
 * ---------------
 * that stores the field dimensions, and will
 * calculate and return the area
 */
{
// attributes
//-----------------------------
  private  double length,width;      // dimensions in metres
//-----------------------------

  Field (double len, double wid)
  // constructor - stores the dimension attributes
  { length = len;
    width = wid;
  }

  double getArea()
  // calculates the area of the field in square metres
  {
    return length*width;
  }
}
```

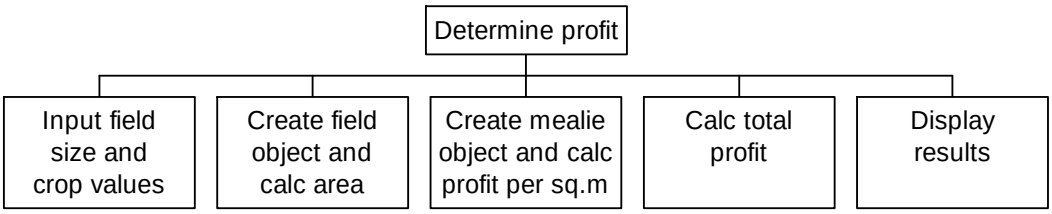
class `Field` *in* `Farming.java`

For the crop class, we need to decide what data types our attributes need to be, write a constructor that will store the values supplied when the object is instantiated in the object's attributes, and design and write a method that will use the attributes to determine crop's profit per square metre. We do this by calculating the income per square metre by multiplying yield by price per sq metre, and then subtracting the costs. Note this method uses a local variable, `income`.

```
class Crop
/* The class representing a crop
 * ----------------------------
 * that stores the costs, yield and selling price of the crop
 * and will calculate and return the profit per sq metre
 */
{
// attributes
//-------------------------
  private  double costs,        // cost per sq m
                  yield,        // yield in kg per sq m
                  sellPrice;    // selling price in R per tonne
//-------------------------

  Crop (double c, double y, double p)
  // constructor - stores values in the crop attributes
  { costs = c;
    yield = y;
    sellPrice = p;
  }

  double getProfit()
  // calculates the profit in R per sq metre
  {
    double income = yield * sellPrice/1000;  // income per sq m
    return income - costs;
  }
}
```

class `Crop` *in* `Farming.java`

The co-ordinating class instantiates the objects it needs and uses them to calculate the total profit. Its design is



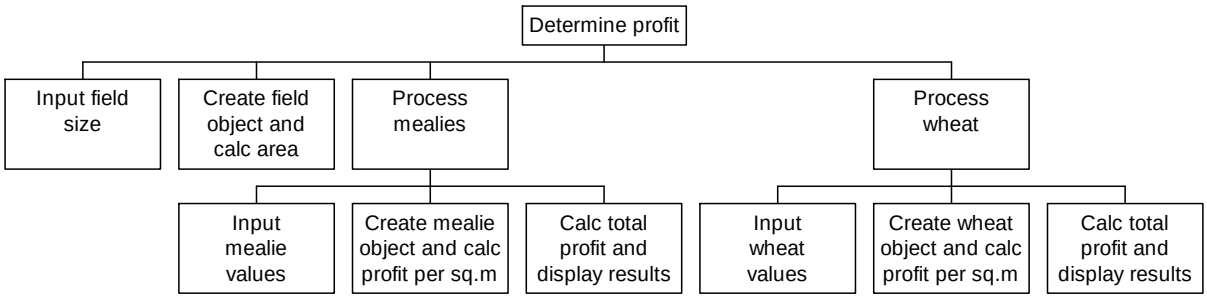
Note that this co-ordinating class must have a main method, and may have other methods if necessary (as may any class). It is good style to declare all the data fields (variables and objects) that are required together at the beginning of the method.

```
class Farming
/*
* The co-ordinating class, containing the main method which
* instantiates a field and a crop, and determines the field
* area, the crop profit and hence the overall profit.
{
  public static void main(String[] args)
  {
// data fields for this class
//----------------------
    double L,W,C,Y,M;       // the values to be input
    Crop mealies;       // the crop object to be created
    Field myField; // the field object to be created
    double area;        // field area in sq m
    double profit;      // crop profit per sq m
    double totalProfit;     // overall profit from this field
//----------------------

// display a heading    < not shown >

// input the required values
    System.out.print("  Enter field length (m)        > ");
    L = Keyboard.getDouble();
    System.out.print("           and width (m)        > ");
    W = Keyboard.getDouble();
    System.out.print("  Enter crop costs (R/sq m)     > ");
    C = Keyboard.getDouble();
    System.out.print("  Enter crop yield (kg/sq m)    > ");
    Y = Keyboard.getDouble();
    System.out.print("  Enter selling price (R/tonne) > ");
    M = Keyboard.getDouble();
    System.out.println();

// create a Field object
    myField = new Field(L,W);
// determine the area
    area = myField.getArea();

// create a Crop object
    mealies = new Crop(C,Y,M);
// determine the profit
    profit = mealies.getProfit();

// calculate total profit
    totalProfit = area * profit;

// Display results
    System.out.println("The field of " + area +
      " sq m earns a profit of R" +
      Formatter.format(profit,1,2) + " per sq m ");
    System.out.println("Overall profit is R" +
      Formatter.format(totalProfit,1,2));
  }
}
```

class `Farming` *in* `Farming.java`

One can immediately see possible extensions - for example, instantiate a number of objects for different crops and compare them to see which is most profitable.

To modify this program to cater for two crops is simple. Classes `Field` and `Crop` are unchanged, just `Farming` is altered to instantiate a second crop object (`wheat`) which is used in the same manner as `mealies`.



It would probably have been better to use a method to input the crop values and do the calculations and call that method twice from `main`, but I wanted to illustrate using more than one instance of the same class in a method.

```
// Instantiate and use a crop object

  // input values for mealies
       :    <code omitted>
  // create a Crop object
    mealies = new Crop(C,Y,M);
  // determine the profit
    profit = mealies.getProfit();
    totalProfit = area * profit;
  // Display results for mealies
       :    < code omitted>

// Instantiate and use another crop object

  // input values for wheat
       :    <code omitted>
  // create the other Crop object
    wheat = new Crop(C,Y,M);
  // determine the profit
    profit = wheat.getProfit();
    totalProfit = area * profit;
  // Display results for wheat
       :    < code omitted>
```

part of `Farming2.java`

As another example of designing a complete program using classes and objects, consider another of the *Intro to Java* exercises:

> Write a program that you could use to test a child's arithmetic, by generating a number of sums involving integers between 0 and 10, and for each sum, display it and ask the user to input the answer, then check whether the answer is correct. Keep score, and finally display the result of the test.

As always, the first step is to identify the classes, their attributes and behaviour.

Question
| value1 |
| value2 |
| answer |
displayQues()
checkAns()
getAns()

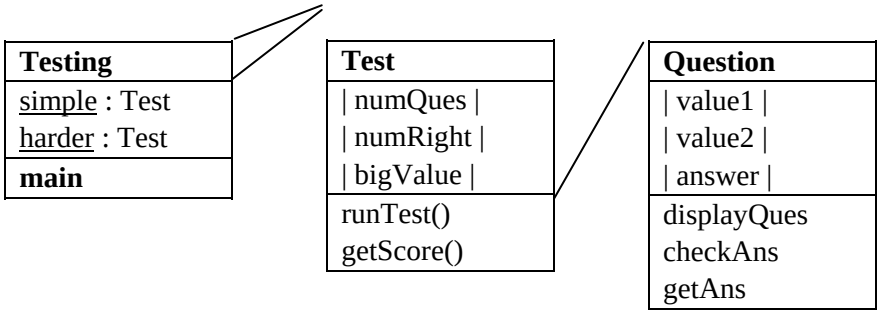
We can have a class for a question.
What are its attributes? - the two values to add (and possibly the answer).
What methods do we require? - one to display the question, another to check if a given answer is right, and one to return the right answer.
Based on the problem requirements, we will generate the values and calculate the answer when the question is created, so we do not need a setAttribute type of method
Other possible attributes if we extend this class are

- what operation to perform instead of merely using addition
- the marks for this question if different marks are possible
- the number of tries permitted (will also need to keep count of the tries)

Test
| numQues |
| numRight |
| bigValue |
runTest()
getScore()

Another class could be a test.
What are its attributes? - the number of questions, a count of right answers, perhaps the largest values to use (not just assume 0-10, but allow any range to be chosen, eg 0-20, 0-1000 etc)
What methods do we require? - one to ask the questions and update the score (which will use our question class), another to return the result of the test.

The coordinating class will merely request a test (or more) and supply the number of questions and the largest value to use.



The question class has 3 attributes, the two values and the answer. The answer does not have to be stored, it could be calculated from the values, but since it is used at least twice (checking the answer and returning the value) it makes sense to calculate it once and store it so that we are certain that the answer used in checking is the same as that made available outside the class. Also, a possible extension is to allow more than one guess.

The constructor for this class must accept a parameter which specifies the upper limit for the values.

```
class Question
/* The class representing a question
 * --------------------------------
 * generates a question and calculates its answer, and will
 * display the question and check if the answer given is right
 */
{
// attributes
//-------------------------
  private  int value1,value2,        // the operands
               answer;               // the result
//-------------------------

 Question(int max)
 // constructor - generates a question with given max value
 {
  value1 = (int)Math.round(max*Math.random());
  value2 = (int)Math.round(max*Math.random());
  answer = value1 + value2;
 }

 void displayQues()
 // outputs the question
 {
  System.out.print(value1 + " + " + value2 + " = ");
 }

 boolean checkAns(int guess)
 // checks whether the guess matches the answer
 {
  if (guess==answer)
    return true;
  else
    return false;
 }

 int getAns()
 // returns the correct answer
 {
  return answer;
 }
}
```

class `Question` *in* `Testing.java`

The test class has 3 attributes, the number of questions in the test, the score and the upper limit for a value in a question. There are 2 constructors, the default one that has no parameters and assumes 10 questions with values between 0-10, and one in which the number of questions and the upper limit for the values must be specified.

The method `runTest()` uses a loop to ask the questions. Inside the loop a question object is created with values in the given range and its display method is called to display the question. The user's answer is input and the questions `checkAnswer` method is called to determine whether the answer is right or wrong. If right, the score is incremented; if wrong, the right answer is displayed. Note that the count of right answers (`numRight`) is set to 0 each time the test is run.

```
class Test
/* The class representing a test
 * ----------------------------
 * that administers a test of a specified number of questions
 */
{
// attributes
//---------------------
  private  int numQues,     // number of questions to ask
               numRight;    // count of right answers
  private int bigValue;     // largest value to use in the test
//---------------------

  Test()
 // constructor - assumes default values
  {  numQues = 10;
     bigValue = 10;
  }

  Test(int num, int max)
 // constructor - gives number of question and biggest value
  {  numQues = num;
     bigValue = max;
  }

  void runTest()
// runs the test - creates a question, displays it,
// inputs the answer, checks it and updates the scores
  {
     Question aQues;// the question object
     int ans;       // the answer entered

     numRight = 0;  // in case test is run more than once
     for (int count=1;count<=numQues;count++)
     {
       aQues = new Question(bigValue);   // create a question
       System.out.print(count + ".  ");
       aQues.displayQues();

       ans = Keyboard.getInt();      // input users answer

       if (aQues.checkAns(ans))      // compare to right answer
       { numRight++;
         System.out.println("Right.");
       }
       else
         System.out.println("Wrong. Ans is " + aQues.getAns());
     }
  }

  int getScore()
// returns the result of the test
  { return numRight;
  }
}
```

class `Test` *in* `Testing.java`

The coordinating program is straightforward - creates two tests, a simpler one of 10 questions with values in the range 0-10 which uses the default constructor, and a harder one

of 5 questions with values in the range 0-100 which passes the values 5 and 100 to the constructor. Each right answer in the harder test is worth 2 marks.

```
class Testing
/*
 * Creates and administers 2 tests
 *   - a "default" one with 10 questions between 0-10
 *   - another one with 5 questions between 0-100
 */
{
  public static void main(String[] args)
  {
  //------------
    Test simple;   // the default test object to be created
    Test harder;       // the more difficult test object
  //------------

// display a heading    < not shown >

// create the first test
    simple = new Test();
    System.out.println("\nFirst test (10 quests, vals 0-10)");
    System.out.println(  "----------");
    simple.runTest();

// create the next test
    harder = new Test(5,100);
    System.out.println("\nNext test (5 quests, vals 0-100)");
    System.out.println(  "---------");
    harder.runTest();

// Display results
    System.out.println("\nFirst test: " + simple.getScore()
      + "/10");
    System.out.println("Next test:  " + harder.getScore()*2
      + "/10");
    System.out.println();
  }
}
```

class `Testing` *in* `Testing.java`

A portion of the output from this program is

```
Next test (5 questions, values up to 100)
---------
1.  72 + 5 = 77
    Right.
2.  44 + 79 = 113
    Wrong. The answer is 123
```

As well as generating random numbers using the `random` method in the Math class, Java also provides a class in the `util` package called `Random`.

public class Random

An instance of this class is used to generate a stream of pseudorandom numbers. The class uses a 48-bit seed, and if two instances of Random are created with the same seed, and the same sequence of method calls is made for each, they will generate and return identical sequences of numbers.

Many applications will find the random method in class Math simpler to use.

CONSTRUCTORS

- `Random()`
  Creates a new random number generator.
- `Random(long)`
  Creates a new random number generator using a single long seed.

METHODS

- `next(int)`
  Generates the next pseudorandom number.
- `nextDouble()`

Returns the next pseudorandom, uniformly distributed double value between 0.0 and 1.0 from this random number generator's sequence.

- `nextFloat()`

Returns the next pseudorandom, uniformly distributed float value between 0.0 and 1.0 from this random number generator's sequence.

- `nextInt()`

Returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence.

- `nextLong()`

Returns the next pseudorandom, uniformly distributed long value from this random number generator's sequence.

- `setSeed(long)`
  Sets the seed of this random number generator using a single long seed.

Once a random object has been created (with or without a seed), the methods `nextInt`, `nextDouble` etc can be called to return the next random number. For double and float values it is probably simpler to use `Math.random`, but for integer and long values a random value in the full range of integers ($-2*10^9$ to $2*10^9$) or longs (-10^{18} to 10^{18}) is returned.

So in the previous example, in class `Question`, we could use

```
private static Random number = new Random();

Question(int max)
// constructor - generates a question with specified max value
{
  value1 = Math.abs(number.nextInt()) % (max+1);
  value2 = Math.abs(number.nextInt()) % (max+1);
  answer = value1 + value2;
}
```

part of `class Question` *in* `Testing2.java`

We create an instance of a `Random` object called `number` without using a specific seed. It is declared as `private` as it is an internal variable, and `static` as we want all instances of `Question` class to share the same object.

Then we construct a value by calling the `nextInt` method (which generates an integer in the range - to + 2 billion), ensure it is positive by calling `Math`'s `abs` method, and constrain it to the range 0-`max` by finding the remainder after dividing by `max+1`.

Exercises

1.1 Using Java's `Date` and `Random` classes, write a program to randomly generate 2 dates, compare them, and display them with the later date second. To generate a date, generate a `long` random number representing a number of milliseconds and construct a date with this value.

1.2 Modify the `Farming` program to compare two fields planted with either mealies or wheat. The farmer wants to know which gives the best profit - mealies in Field 1 and wheat in Field 2, or wheat in Field 1 and mealies in Field 2. Field 1 is 400x500m and has costs of R1.35 per m^2 for mealies, or R1.25 per m^2 for wheat, and gets a yield of 45 kg mealies per m^2, or 35 kg wheat per m^2. Field 2 is 300x400m and has costs of R1.80 per m^2 for mealies, or R1.65 per m^2 for wheat, and gets a yield of 50 kg mealies per m^2, or 45 kg wheat per m^2. Mealies sell for R60.00 per tonne, and wheat for R70.00 per tonne.

1.3 Modify the `Test` and `Question` classes to cater for addition and subtraction. When each question is created it should be passed a randomly determined argument (0/1, true/false etc) to indicate whether it is to add or subtract the values. This argument must be used in generating the question and answer, and in the display method.

1.4 Design object oriented solutions for all the exercises in the *Intro to Java* lecture notes.

1.5 Write a `LineSegment` class that will represent a line segment by the co-ordinates of its endpoints, and provide methods to

- get and set the co-ordinates
- calculate and return the gradient of the line,
- calculate and return the length of the line,
- calculate and display the equation of the line
- test whether a given point lies on the line

Write a program to test your class.

1.6 Write a `Number` class with methods that will determine various characteristics of an integer number. Include methods to

- get and set the number
- check whether it is a prime number
- display its factors
- check whether it is perfect, deficient or abundant

(a number is perfect if the sum of its factors including 1 but excluding the number itself equals the number; it is abundant if the sum of factors is larger than the number; it is deficient if the sum of factors is less than the number.)
Test your class.

11. Streams, Files and Exceptions

11.1 Input and Output Streams

All input and output in Java is accomplished with streams, which is simply a communication path between the program and a source of information or its destination. The advantage of using streams is that all streams are treated the same, and it doesn't matter whether the source of an input stream is the standard input device, a disk file, or even the Internet.

The standard output stream is the screen. `System.out` is the standard output stream object, and is a member of the `PrintStream` class. The `PrintStream` class has methods `print` and `println` which display their arguments on the screen.

The standard input stream is the keyboard, and the Java equivalent of `System.out` is `System.in`, which is the standard input stream object and is a member of class `InputStream`. It is not very useful as is however, as it merely provides a method, `read`, which reads a single byte from the input stream. Of far more use is to create a usable object by supplying `System.in` as construction parameter to another class called `InputStreamReader` which handles character streams rather than byte streams. Bear in mind that the data supplied by `InputStream` is merely a sequence of binary codes which need to be converted to characters, integers etc, so the conversion of this raw data to characters is an improvement. Then this object should be passed to another class, `BufferedReader`, which reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters and lines.

The implications of all this are that in order to declare the keyboard for input we need the statements

KEYBOARD DECLARATIONS

```
import java.io.*;
BufferedReader stream = new BufferedReader
                  (new InputStreamReader(System.in));
```

where *`stream`* is an identifier you choose to denote the stream connected to the keyboard. This then declares a new stream, connected to the standard input device `System.in`, with all the facilities of the `BufferedReader` class. Note that the `java.io` package in which the three classes are defined must be imported before the class statement in which this stream is declared.

One side effect of reading is that something could go wrong, for example, the data might be in the wrong format or might end unexpectedly. Such events are called exceptions and will be dealt with more fully later in this chapter. Suffice to say that Java requires that when we use methods that may cause exceptions (such as `BufferedReader`'s methods) we indicate this by adding the phrase

```
    throws IOException
```

to the method header.

For example, many childrens game programs start off by asking for the users name and then welcoming them. A simple Java program to do this uses input streams and `BufferedReader`'s `readLine` method that will read a line of text from the keyboard. Note that a line of text is defined as all the characters up to the control characters generated when the Enter key is pressed (or the end of the data stream is reached).

```
/*
* Simple greeting program - demonstrates Java input
* ------------------------------------------------
*/
import java.io.*;

public class Hello
{
  public static void main(String[] args) throws IOException
  {
    BufferedReader in = new BufferedReader
        (new InputStreamReader(System.in));

    System.out.print("What is your name? ");
    String name = in.readLine();

    System.out.println();
    System.out.println(" ******************");
    System.out.println("** Hello " + name + "! **");
    System.out.println(" ******************");
  }
}
```

Hello.java

The output from this program is

```
What is your name? Andrew

 ******************
** Hello Andrew! **
 ******************
```

The class BufferedReader provides methods to read a single character and a line of text, as above. Unfortunately if we want to read numeric values things are not as simple - the line of text need to be read and then numeric values of the correct type must be extracted from it. For example, to read an integer value would require

```
int myInt = Integer.valueOf(in.readLine().trim()).intValue();
```

What this does is the following:

- `in.readLine()` reads a line of text from the standard input device
- `trim()` removes any spaces from both ends of the string
- this string provides the argument to the `Integer` class's `valueOf` method, which returns the `Integer` object that this string represents
- and finally `Integer`'s `intValue()` method converts this `Integer` object to an `int` value

To simplify matters, methods to cater for reading numeric values of all types from the keyboard have been grouped together in the `Keyboard` class, which forms part of the `Utilities` package. This class also deals with any exceptions, so the phrase

```
throws IOException
```

is not required in methods that use `Keyboard`'s methods to handle input.

For interest, a portion of the `Keyboard` class (edited to show just the input handling) is shown here.

```
import java.io.*;
public class Keyboard
{
  private static BufferedReader br =
      new BufferedReader(new InputStreamReader(System.in));

  public static long getLong()
//----------------------------
  {
    localLong = Long.valueOf(br.readLine().trim()).longValue();
    return localLong;
  }

  public static int getInt()
//--------------------------
  {
    localInt =
        Integer.valueOf(br.readLine().trim()).intValue();
    return localInt;
  }

  public static float getFloat()
//------------------------------
  {
    localFloat =
        Float.valueOf(br.readLine().trim()).floatValue();
    return localFloat;
  }

  public static double getDouble()
//--------------------------------
  {
    localDouble =
        Double.valueOf(br.readLine().trim()).doubleValue();
    return localDouble;
  }

  public static char getChar()
//----------------------------
  {
    localChar = br.readLine().trim().charAt(0);
                                  // first char in the string
    return localChar;
  }
  public static String readString()
//--------------------------------
  {
    StringBuffer localStringBuffer = new StringBuffer();
    localStringBuffer.append(br.readLine());
    return localStringBuffer.toString();
```

```
  }

  public static void pressEnter()
//-------------------------------
  {
    br.readLine(); // pauses till ENTER is pressed
  }
}
```

edited extract from `Keyboard.java`

In recent versions of Java it is simpler to convert strings to `integer`, `double` and `float` by using:

- `Integer.parseInt(String)` which returns an int
- `Double.parseDouble(String)` which returns a double
- `Float.parseFloat(String)` which returns a float

11.2 File Input and Output

Up till now, all data needed by a program has been input interactively via the keyboard, and all data has been output to the default output device, the screen. However, for large amounts of data, interactive input can rapidly become tedious, and it is preferable to keep the input data required in a file and then read it from there. Likewise, it is useful to be able to send any output to a file so that the results can be kept and printed if necessary.

If a file is to be used it must first be "opened" for input or output, by associating it with a stream. The standard input and output streams, `System.in` and `System.out` are automatically opened and associated with the standard devices by Java - you need to explicitly perform this function for other files that are to be used as input or output streams.

FILE DECLARATIONS
`BufferedReader` *`stream`* `= new BufferedReader (new FileReader("filename"));`
`PrintWriter` *`stream`* `= new PrintWriter (new FileWriter("filename"));`

The first declaration is used to open a file for input (ie. to read from it). Instead of supplying `System.in` as the parameter to the `InputStreamReader` constructor, we create a new object of the `FileReader` class by supplying the actual filename (as a string). This has the effect of connecting an input stream to that file and opening it so that it can be read from. For example, to read from a data file called "`testdata.dat`", we define

```
BufferedReader fin = new BufferedReader
            (new FileReader("testdata.dat"));
```

Similarly, to open file for output (ie. to be able to write to it), we use the second declaration. `PrintWriter` implements all the print methods of `PrintStream` (notably `print()` and `println()`), so in order to be able to use them on a file we create a `PrintWriter` object, which we initialise with a `FileWriter` object created with the name of the file we want to create.

For example, to send output to a file called "myResults.txt", we define

```
PrintWriter fout = new PrintWriter
            (new FileWriter("myResults.txt"));
```

Then to output data to the file we use statements such as

```
fout.println("This is going to a text file");
```

When using files, you must ensure that they are closed before the program ends. This is good programming but not essential for input files, but is crucial for output files otherwise the output sent to them is lost. Both `FileReader` and `FileWriter` have a close method that should be used to close any files, as in

```
fin.close();
```

or

```
fout.close();
```

To simplify input from a file a set of equivalent input methods to those provided in the `Keyboard` class are available, but instead of assuming that the data is coming from the standard input stream, the user must specify the stream object when calling the method.

This class, `FileIO`, is also contained in the `Utilities` package, and provides methods to open an input stream (either by specifying a file or the standard input stream), create a file for output, and various read methods to input data from a specified input stream.

The methods available in `FileIO` are:

CLASS FileIO METHODS	
br = FileIO.open(System.in);	connects a buffered reader object to the standard input stream
br = FileIO.open("*filename*");	connects a buffered reader object to a file for input
pw = FileIO.create("*filename*");	connects a print writer object to a file for output
intVar = FileIO.getInt(*br*);	inputs an `int` value from *br*
longVar = FileIO.getLong(*br*);	inputs a `long` value from *br*
floatVar = FileIO.getFloat(*br*);	inputs a `float` value from *br*
doubleVar = FileIO.getDouble(*br*);	inputs a `double` value from *br*
charVar = FileIO.getChar(*br*);	inputs a single `char` from *br*
stringVar = FileIO.readString(br);	inputs a `String` from *br*

Note that any methods that use `open` or `create` and supply a file name must add the phrase

```
throws IOException
```

to the method header, in case the file does not exist (`open`) or could not be created for some reason (`close`). The file is assumed to be in the same directory as the Java program unless a path is specified as part of the file name. Remember that to use a buffered reader or print writer the class `java.io.*` must also be imported.

The following program demonstrates use of class `FileIO` by reading a file name from the keyboard and then reading a name (`String`), a student number (`long`) and 3 `marks` (`int`, `float` and `double`) from the file and calculates the average mark. All output is to a results file.

```
/*
* Demonstration of FileIO class
* -----------------------------
*/
import java.io.*;
import Utilities.FileIO;
import Utilities.Formatter;

public class FileIOdemo
{
  public static void main(String[] args) throws IOException
// Asks for a file to open, then opens it and inputs a students
// name, number, and 3 marks which are averaged.
// All output is to a results file.
  {
    String filename;        // file to input values from
    String stName;
    long stNum;
    int mark1;
    float mark2;
    double mark3;
    double av;

// get name of file to read from
    BufferedReader in = FileIO.open(System.in);
    System.out.print("\nWhat file to use for marks input? ");
    filename = FileIO.readString(in);

// open files for input and output
    BufferedReader fin = FileIO.open(filename);
    PrintWriter fout = FileIO.create("output.txt");

// display a heading to the results file
    fout.println("This program tests FileIO");
    fout.println("-------------------------");
    fout.println();

// input the values from the input file
    stName = FileIO.readString(fin);
    stNum = FileIO.getLong(fin);
    mark1 = FileIO.getInt(fin);
    mark2 = FileIO.getFloat(fin);
    mark3 = FileIO.getDouble(fin);

// output the results
    av = (mark1 + mark2 + mark3)/3.0;
    fout.println("Results for " +stName+ "  student# " +stNum);
    fout.println("  mark1" + Formatter.format(mark1,10));
    fout.println("  mark2" + Formatter.format(mark2,12,1));
  fout.println("  mark3" + Formatter.format(mark3,12,1));
    fout.println("Average" + Formatter.format(av,13,2));
    fout.println();

    fin.close();   // close the files
    fout.close();
  }
}
```

FileIOdemo.dat

An input file, `FileIOdemo.dat` was set up to contain the following data. Note that the name does not have to be the same - I just do it so I know which data belongs to which program. To set up a file containing input data all you need to do is to use any text editor (Notepad, Kawa etc) and type the data values you want to use much as you would type them when inputting interactively fron the keyboard. Then the file is saved to the same directory as the program and its name is passes to the `open` method.

```
John Smith
987654321
83
73.9
62
```

`FileIOdemo.dat`

When the program is run, the following appears on the screen,

```
C:\java1.2\bin\java.exe   FileIOdemo

What file to use for marks input? FileIOdemo.dat
Process Exit...
```

but the contents of the newly created file `output.txt` are

```
This program tests FileIO
-------------------------

Results for John Smith  student# 987654321
  mark1        83
  mark2        73.9
  mark3        62.0
Average        72.97
```

`output.txt`

If an error was made in entering the name of the file to use for input, the effect would be:

```
C:\java1.2\bin\java.exe   FileIOdemo

What file to use for marks input? FileIO.dat

File FileIO.dat does not exist.

java.io.FileNotFoundException
  at Utilities.FileIO.open(FileIO.java:35)
  at FileIOdemo.main(FileIOdemo.java:30)
Process Exit...
```

11.3 Exceptions

Exceptions are unexpected events that cause a program to fail. In the previous section, dealing with input and output, the sorts of exceptions we may encounter are caused by errors such as "*file not found*", "*illegal format for numeric data*" (eg, a decimal point in an `int` value, or an alphabetic character where a number is expected). Other common examples are arithmetic exceptions (integer divide by zero) and array index exceptions.

Java provides an exception handling mechanism to deal with exceptions when they occur. They are intended to be detected and dealt with so that the program can continue if at all possible. The idea is to identify some statements that may cause an exception when executed, and to write your program to explicitly deal with exceptions. This allows more robust and reliable programs to be created. If an exception is not dealt with in the method we are in, the method is terminated and the exception is passed up to the method that called it. This process may be repeated until eventually an unhandled exception is passed to the Java virtual machine which terminates the program.

In Java, exceptions are ***thrown***, so we ***try*** to execute a statement and should an error occur, some code is provided to ***catch*** any exceptions that are thrown.

```
try
{
  some statements;
}
catch (anException e)
{
  statements to deal with the exception
}
```

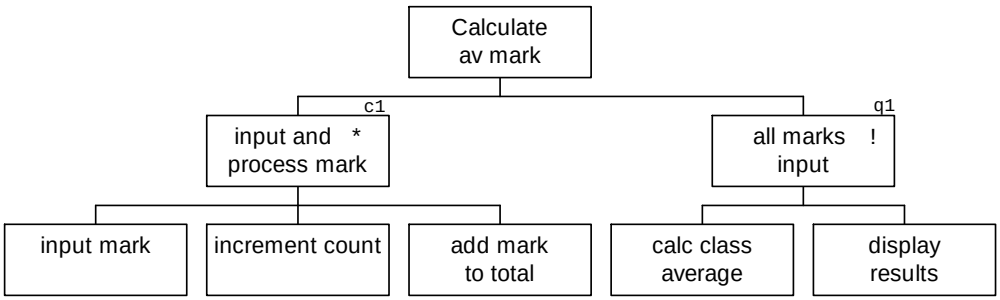
The try block is executed as part of the usual sequence of exception, and if an exception occurs then the execution of the try block is interrupted and control is transferred to the catch block to deal with the exception. There are various kinds of exception and you need to specify which kind you are catching, and there may be more than one catch block to deal with different types of exception. After dealing with the exception the program continues executing the statements following the catch block.

Although in many cases exceptions are used to deal with errors, they can also be used "positively". Consider the previous example where we read marks from a file. We knew there were 3 marks and explicitly programmed for it. But what of the case where we want to (for example) read in marks from a file for an unknown number of students and calculate a class average. As things stand at present, I can think of 2 ways to handle it:

- count the number of marks and include that number at the beginning of the file, so the number of marks is read and a for-do loop is then used to process each mark;
- include a special "sentinel" value at the end of the file to indicate that the last actual mark has been read, and use a while-not-sentinel loop to process each mark.

Both these require modification of the file.

Using exceptions, such variable length data can be easily dealt with, because when the end of a data file is reached, Java throws an `EOFexception` (end-of-file exception) which can be detected. The program is structured so the try block handles the input, and when the end-of-file exception is thrown control passes to the catch block which does the averaging and output of the final result.




```
c1 : loop forever
q1 : EOF exception
```

The program is

```
/*
* Calculates the average test mark for a class of students
* using an exception to detect the end of the file
* ------------------------------------
*/
import java.io.*;
import Utilities.*;

public class MarkAvExcept
{
  public static void main(String[] args) throws IOException
  {
    int countSt = 0;        // count of the number of students
    double mark;   // a student's mark
    double total = 0;       // the running total
    double classAv;         // the average test mark

  // Open input file
    BufferedReader in = FileIO.open("MarkAv.dat");

  // Display heading
      ...   <code omitted>

  // Indefinite loop to process marks
    try
    {
      while (true)
      {
        mark = FileIO.getDouble(in);
        System.out.println("  " + mark);
        countSt++;     // increment count of
        total += mark;     // add mark to total
      }
    }
```

```
catch (EOFException e)
    {
      classAv = total/countSt;
      System.out.println();
      System.out.println("The class average for the " + countSt
        + " students is " + Formatter.format(classAv,1,2));
      System.out.println();
    }
    in.close();
  }
}
```

Exceptions are used in the `Keyboard` and `FileIO` classes to detect and report certain data errors. For example, in `getInt()`, some of the code is

```
public static int getInt(BufferedReader br) throws IOException
//---------------------------------------
{
  int localInt = 0;
  try
  {
    localInt = Integer.valueOf(br.readLine().trim()).intValue();
  }
  catch (NumberFormatException e)
  {
    System.err.println();
    System.err.println("Error entering int: " + e.getMessage());
    System.err.println();
  }
  return localInt;
}
```

A `NumberFormatException` is thrown to indicate that an attempt has been made to convert a string to one of the numeric types, but that the string does not have the appropriate format. So for example, if there were invalid characters entered that cannot form part of an integer - decimal point, embedded + or space, alphabetic char etc - this exception will be thrown. The method catches it and outputs a message "`Error entering int`" together with any details supplied by the exception using the Java supplied method, `getMessage`.

Exercises

These programs should all get their input from data files.

1. Write a program that reads in a student's data from a file - name, student number, and 5 test marks - and determine the student's average class mark by discarding the lowest mark and averaging the remainder.

2. Write a program to input an integer from the keyboard, and then generate 1000 random numbers between 0 and this number and output them to an output file, 10 per line.

3. Write a program to generate a random integer N between 50 and 100, and then create an output file consisting of N random numbers between 0 and 100, one per line. Then a write a second program to read these values (till the end of the file is reached) and calculate their average, which should be displayed to 2 decimal places.

12. Arrays

In the programs written so far, each variable was associated with a single memory cell. There are many cases where it is convenient to store a related set of values without having to create a separate variable for each value - for example, the marks of all the students in a class.

12.1 Simple arrays

An array is a data structure which stores a collection of data items all of the same type. Using an array, we can associate a single name with a set of data items, and reference the individual items by a number representing their position in the array. This means we can use integer variables - for example, the loop control variable of a for loop - to access each element in an array in turn. If all the data values were stored using individual variable names this would not be possible as variable names have to be directly specified and cannot be generated by the program.

An array is bounded - it has a fixed size which is specified when the array is created, and the individual elements of an array are indexed by a number in the range 0 to this upper limit. In contrast to some other languages (eg Pascal), all arrays are indexed by numeric values starting at 0.

ARRAY DECLARATION
type [] *arrayname*;
type [] *arrayname* = **new** *type* [*size*];
type [] *arrayname* = {*list of values*};

The first form merely declares an array variable to be of a certain array type; the next two create the array variable and initialise it. The first form would be require the array to be initialised at a later stage - perhaps when its size is known.

Some examples:

```
     int [] marks = new int [30];        // assuming 30 students in the class
```

or
```
     double [] examMarks;
     int numStudents;
       :
     System.out.print("How many students wrote? ");
     numStudents = Keyboard.getInt();
     examMarks = new double [numStudents];
```

or
```
     char [] vowels = {'A','E','I','O','U'};
```
This creates an array of 5 elements - the size is deduced from the number of values given.

Note that it is also possible to write the array declaration as
```
     type arrayname [];
```
eg
```
   int marks [] = new int [30];
```
but this splits the array type which is actually `int []`.
The declaration is read as "*an integer array called marks* …."

The `marks` array declared above will have 30 integer elements numbered from 0 to 29.

The number of elements in the `examMarks` array depends on the value read for `numStudents`, and its elements will be indexed 0 to `numStudents-1`.
The `vowels` array has 5 character elements (deduced from the number of values specified) indexed 0 to 4.

To reference a specific item in an array we use the array name and a numeric subscript which identifies a particular element. For example,

```
System.out.println(marks[0]);
```

will output the first element in the `marks` array, while

```
System.out.println(marks[10]);
```

will output the 11th element.
The subscript must be in the range `0..arraysize` or an exception will be thrown (`ArrayIndexOutOfBoundsException`).

A for loop is frequently used to process all elements in an array. For example, to read in 30 marks and store them in the marks array:

```
for (int count=0;count<30;count++)
{
  System.out.println("Enter mark "+(count+1)+">");
  marks[count] = Keyboard.getInt();
}
```

Similarly, we could output all the marks in the array:

```
for (int count=0;count<30;count++)
{
  System.out.println(marks[count]);
}
```

Or add them up and calculate the average:

```
int total = 0;
double average;
for (int count=0;count<30;count++)
  total += marks[count];
average = total/30;
```

Or find the top mark and which student got it:

```
int maxMark = marks[0];
int topSt = 0;
for (int count=1;count<30;count++)
{
  if (marks[count]>maxMark)
  {
    maxMark = marks[count];
    topSt = count;
  }
}
System.out.println("Top mark of " + maxMark +
    "was obtained by student " + topSt);
```

Some points to note about arrays:

- Arrays can be of any **element type** or class, for example arrays of Reals, arrays of characters, arrays of Strings, arrays of Dates, and even arrays of arrays. These are called 2-dimensional arrays and will be considered in more detail later.

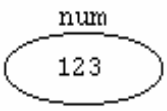
- An array can be any **size** (subject to the constraints of memory), but the size is fixed when the array is created and cannot be changed.
```
double [] rainfall;
rainfall = new double [31];
```
When the second statement is executed the size of `rainfall` is fixed at 31 elements. They do not all have to be used, but the storage for 31 double values is reserved.

- Each array has a special property associated with it, its **length**. This is the size of the array as specified in its declaration, and can be accessed. For example,
```
System.out.println(rainfall.length);
```
would output 31.
Note that `length` is a property, not a method, and does not have brackets after it.

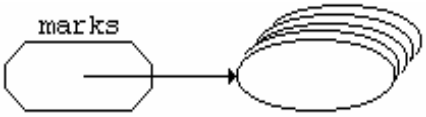
- Java is strict about **index checking**. Only array elements that actually exist can be referenced. Each time an array is accessed, the subscript supplied is checked against the array size given in the declaration. If the subscript used is below 0 or greater than or equal to the size declared, an `ArrayIndexOurOfBoundsException` is thrown. The exception can be caught and handled, or else the program stops.

- A simple variable such as an integer stores the actual numeric value in its memory location. An array variable does not actually store all the array values, or even the first value, in its memory location. Instead it stores a **reference** to the place in memory where the array is stored. When the array is declared the reference is set up, and when its size is specified the storage will be created and its location stored in the array variable. This is an important concept and its relevance and use will be further explained in later sections.

```
int num = 123;
```




```
int [] marks = new int [5];
```

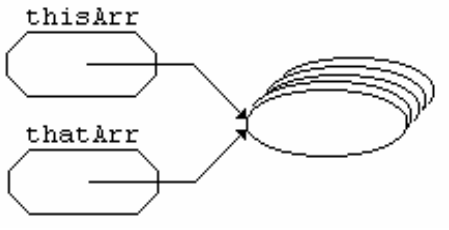


- The arithmetic and logical **operators** do not apply to arrays, only to simple variables. Statements such as
```
double [] rainJan,rainFeb,rainYr;
rainJan = new double [31];
rainFeb = new double [31];
  :
rainYr = rainJan + rainFeb;
```
are illegal and will cause errors.
Similarly, one cannot write
```
if (rainJan < rainFeb)
```

The only operator that applies to a whole array is assignment,

```
thatArr = thisArr;
```

However, this does not copy the entire array, but merely copies the reference so that both arrays will refer to the same storage. Any changes made to one array will affect the other.



In order to copy an entire array a for loop should be used:

```
for (count=0;count<max;count++)
  thatArr[count] = thisArr[count];
```

Similarly, in order to add two arrays a for loop is used:

```
for (day=0;day<31;day++)
  rainYr[day] = rainJan[day]+rainFeb[yr];
```

Note that all the arrays used must have been declared and initialised.

- Arrays can be passed as **parameters** to methods. In the method header it is sufficient to use square brackets to indicate that the parameter is an array, its size does not have to be specified. This means we can, for example, write a general `IsEqual` method that accepts 2 integer arrays and returns true only if all corresponding elements are equal. The same method can then be passed arrays of 10 elements or 100 elements.

```
boolean IsEqual(int [] arr1, int [] arr2)
{
  boolean same = true;
  int count = 0;

  if (arr1.length != arr2.length)    // unequal sized arrays
    same = false;                    // cannot be equal

  while (same && (count<arr1.length))
  {
    if (arr1[count]==arr2[count])
      count++;
    else
      same = false;
  }

  return same;
}
```

It is worth spending a minute to examine this method. A boolean variable `same` is initialised to true. Then the lengths of the 2 arrays are checked, and if they are unequal the arrays cannot possibly be equal so `same` is set false. We then come to the loop to compare corresponding elements and this loop is controlled by 2 conditions, the boolean value `same` (which means that this loop won't even be entered if the lengths are unequal, and also that as soon as an unequal pair is found the loop will cease) and the array size (so that we don't go out of array bounds). If the loop completes all its iterations and terminates because the array bound is reached, then `same` will still have its initial value of true and the two arrays are equal; otherwise `same` will have been set false, the loop will terminate and a value of false is returned indicating the two arrays are not equal.

Contrast this method with another, seemingly similar method:

```
boolean IsEqual(int [] arr1, int [] arr2)
{
  boolean same;

  if (arr1.length == arr2.length)    // unequal sized arrays
    same = true;                     // cannot be equal
  else
    same = false;

  for int count=0;count<arr1.length;count++)
  {
    if (arr1[count]==arr2[count])
      same = true;
    else
      same = false;
  }

  return same;
}
```

Will this method work correctly?

To illustrate, consider 2 3-element arrays, where A1 contains the integers 1 3 5 and A2 contains the integers 3 4 5.

```
                                          same
A1.length == A2.length          ->        true
count=0:  A1[0]=1, A2[0]=3      ->        false
count=1:  A1[1]=3, A2[1]=4      ->        false
count=2:  A1[2]=5, A2[2]=5      ->        true
```

returns: true !

The problem is that `same` is set and reset each iteration, and so effectively the answer returned is the result of comparing the last pair of values only. The correct way to handle a problem like this is to set `same` to true, and thereafter only to set it false when an unequal pair is found.

The following code is not as efficient as the first because all the values are checked even when it is already known that the arrays contain unequal values, but at least it gives the right result.

```
boolean IsEqual(int [] arr1, int [] arr2)
{
  boolean same;

  if (arr1.length == arr2.length)    // unequal sized arrays
    same = true;                     // cannot be equal
  else
    same = false;

  for int count=0;count<arr1.length;count++)
  {
    if (arr1[count]!=arr2[count])
      same = false;
  }

  return same;
}
```

Classes and arrays can interact in a number of ways.

A class may contain methods that operate on array parameters, as in the example above (`IsEqual`).

A class may contain an array and the methods that operate on it. In this case the array will (usually) be private and all access is via the defined methods. For example, a class representing a class of students may use an array to store the students marks, and methods to store the marks, display the average, and grade the marks as Good, Fair or Poor.

```
class MyClass
/* The class representing the class of students
 * ----------------------------------------
 * stores the number of students, their marks and the average
 */
{
// attributes
//-------------------------
  private int maxSt;        // the maximum class size
  private  int numSt;       // the actual number of students
  private  double [] marks; // array marks for the students
  private double average;   // the class average
//-------------------------
```

```
MyClass(int max)
//----------------
// constructor - creates the array to hold marks for the maximum
// number of students, and initialises all other attributes.
  {
  marks = new double [max];
  maxSt = max;
  numSt = 0;
  average = 0.0;
  }

  void getMarks(String filename) throws IOException
//-------------------------
// Read in the array of marks from the specified file,
// keeping count of the actual number of students
  { … }

  int getNumSt()
//--------------
// Returns the number of marks stored
  { … }

  double getAv()
//--------------
// Returns the average mark
  { … }

  void gradeMarks()
//------------------
// Output the marks, assigning a comment to each:
//  average +- 10% : "fair"
//  over average +10%  : "good"
//  below average -10% : "poor"
  {
    double upav = average*1.1,       // upper split, av +10%
           lowav = average*0.9;      // lower split, av -10%

    System.out.println();
    System.out.println("Graded class marks");
    System.out.println("------------------");

    for (int count=0;count<numSt;count++)
    {
      System.out.print(Formatter.format(marks[count],10,1));
      if (marks[count]>upav)
        System.out.println("   Good");
      else if (marks[count]<lowav)
        System.out.println("   Poor");
      else
        System.out.println("   Fair");
    }

    System.out.println();
  }

}
```

part of `class MyClass` *in* `GradeClassMark.java`

Thirdly, you can have an array of a class of objects, as in this array that stores the dates of all the public holidays.

```
Dates [] publicHolidays = new Dates [12];

publicHolidays[0].set(1999,0,1);       // New Years Day
publicHolidays[1].set(1999,2,21);      // Human Rights Day
…
publicHolidays[11].set(1999,11,26);    // Day of Goodwill
```

(months are stored as 0-11, not 1-12)

12.2 Sorting

When working with sequences of numbers, a common requirement is to be able to sort the numbers into ascending or descending order. For example, you may wish to sort an array of student marks so that the highest marks is first and the lowest mark is last. There is a wide range of sorting algorithms ranging from simple sorts which perform in time proportional to n^2 (where n is the number of items to be sorted) to more efficient and more complex sorts such as Quicksort or Heap Sort which are O(n.log(n)). We will consider 2 of the simple sorts, Bubble Sort and Selection sort, which have the advantage of being easy to understand and are adequate for our pruposes.

Bubble Sort

Bubble Sort compares pairs of adjacent elements and if they are out of order, exchanges them. In this way the smaller (or larger) elements "bubble" their way to the top of the sequence, while the larger (or smaller) elements move to the bottom.

Assume we wish to sort the n elements of array A into ascending order. The basic method is:

- Compare A[0] and A[1] ; swap if necessary so that the larger value is in A[1]
- Compare A[1] and A[2] ; swap if necessary so that the larger value is in A[2]
  :
- Compare A[n-2] and A[n-1] ; swap if necessary so that the larger value is in A[n-1]

This is called a pass.

Repeat this pass as many times as are necessary until no values are swapped during a pass. This means that the values are all in the right order, so the array A is then sorted.

For example, consider the sequence A = < 5,3,1,7,4,6 >

```
5    3    1    7    4    6

5 ? 3
  swap

3    5 ? 1
        swap

     1    5 ? 7 ? 4
             OK    swap

                  4    7 ? 6
                          swap

3    1    5    4    6    7        End of pass 1
```

Notice that after 1 pass the largest value is in the last position. This means that in pass 2, we do not have to compare the last pair since we know that the last value is bigger than the second last, so we need only compare the n-2 pairs up to A[*n*-3] and A[*n*-2].

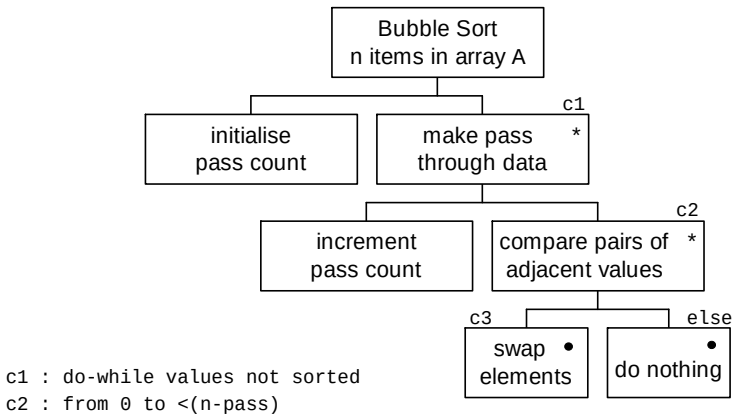
```
3    1    5    4    6    7

3 ? 1
  swap

1    3 ? 5 ? 4
        OK   swap

             4    5 ? 6
                     OK

1    3    4    5    6    7        End of pass 2
```

After pass 2 the second largest value is in the second last position, so in the third pass we need compare only the *n*-3 pairs up to A[*n*-4] and A[*n*-3].

```
1    3    4    5    6    7

1 ? 3 ? 4 ? 5
   OK    OK    OK

1    3    4    5    6    7        End of pass 3 - no swaps
```

This means that the sequence is now sorted.

The processing required to program the bubble sort is thus:



The method to perform a simple Bubble Sort to sort an array of integers into ascending order is shown below. It forms part of class `mySort` which contains a number of sorting algorithms.

```
public static void Bubble (int [] A, int n)
//-------------------------------------------
// implements a Bubble sort to sort n elements in A into
// ascending order
{
  int pass = 0;             // counts the number of passes
  boolean swapped;

  do
  {
    pass++;
    swapped = false;            // no swaps made

    for (int i=0;i<(n-pass);i++)
    {
      if (A[i]>A[i+1])
      {
        int temp = A[i];        // exchange values
        A[i] = A[i+1];
        A[i+1] = temp;
        swapped = true;
      }
    }
  } while (swapped);            // repeat while not sorted
}
```

method `Bubble` *in* `mySort.java`

One point to note is that, because arrays are passed by reference, any changes made to an array during the sort (ie. repositioning of elements) will be effective and visible outside the sort method.

A part of a program which generates an array of random integers and sorts them using the bubble sort is shown below, together with some output.

```
/*
* Program to test sorting routines.
* An array of random integers is generated, displayed,
* sorted and then displayed again.
* ------------------------------------
*/
import java.util.*;
import SortAndSearch.*;
import Utilities.*;

class TestSort
{
  public static void main(String[] args)
  {
    int [] Numbers = new int [50];
    int count = 0;

    count = Generate(Numbers);
    System.out.println();
    System.out.println(count + " Numbers as generated");
    Display(Numbers,count);
    mySort.Bubble(Numbers,count);
    System.out.println(count + " Numbers when sorted");
    Display(Numbers,count);
    System.out.println();
  }
```

part of TestSort.java

```
C:\java1.2\bin\java.exe   TestSort

16 Numbers as generated
=======================

   570  -149  -650   854   -70   416  -532   327   122   669
   810  -898   557  -653  -721    -4

16 Numbers when sorted
======================

  -898  -721  -653  -650  -532  -149   -70    -4   122   327
   416   557   570   669   810   854

Process Exit...
```

When creating a package of "useful" classes and methods, there are a few points to note:

- each file containing a package class must contain a package statement at the beginning;

```
/* MySort.java
 * Class implementing various sorting routines
 *-------------------------------------
 */

package SortAndSearch;

public class mySort
{

  public static void Bubble (int [] A, int n)
//-------------------------------------------
// implements a Bubble sort
  {
    …
  }

   public static void Select (int [] A, int n)
//-------------------------------------------
// implements a Selection sort
  {
    …
  }
}
```

- the package name must be a folder, and the various classes comprising that package are located in that folder

- the folder must be in the same folder as the classes that import it; or it must be at a level above all the folders that contain classes that use it and its parent directory must be included in your classpath.

- all classes and methods that can be used in a package must be declared as public.

Selection Sort

Selection sort considers each position in the array in turn, and selects the correct value to be in that position. An ascending selection sort will look for the smallest value in the array and store that at index 0, then look for the next smallest value and store that at index 1, and so on.

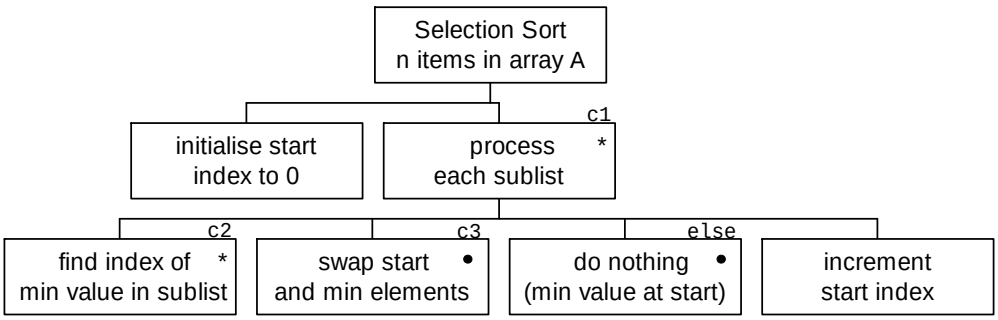
Once again, assume we wish to sort the *n* elements of array A into ascending order. The basic method is:

- Find the smallest of the n elements A[0] … A[*n*-1] and swap it with the first element A[0] so that the smallest value is in the first position.
- Then find the smallest of the *n*-1 elements A[1] … A[*n*-1] and swap it with the second element A[1] so that the next smallest value is in the second position.
- Continue like this until there is only 1 element A[*n*-1] to be considered. The array is then sorted.

For example, consider the sequence A = < 5,1,4,6,7,3 >

- Consider the 6 elements A[0] … A[5]
  Smallest is A[1]=1, swap A[1] and A[0]: A = < 1,5,4,6,7,3 >
- Consider the 5 elements A[1] … A[5]
  Smallest is A[5]=3, swap A[5] and A[1]: A = < 1,3,4,6,7,5 >
- Consider the 4 elements A[2] … A[5]
  Smallest is A[2]=4, in correct position: A = < 1,3,4,6,7,5 >
- Consider the 3 elements A[3] … A[5]
  Smallest is A[5]=5, swap A[5] and A[3]: A = < 1,3,4,5,7,6 >
- Consider the 2 elements A[4] … A[5]
  Smallest is A[5]=6, swap A[5] and A[4]: A = < 1,3,4,5,6,7 >
- Consider 1 element -> array is sorted.

The processing required to program the selection sort is:




```
c1 : while >1 element in sublist
c2 : from start index to <n
c3 : if A[start]!=A[min]
```

The method to perform a simple Selection Sort to sort an array of integers into ascending order is shown below. It also forms part of class `mySort`.

```
public static void Select (int [] A, int n)
//-------------------------------------------
// implements a Selection sort to sort n elements in A into
// ascending order
{
  int startIndex = 0;       // the start of the sublist
  int minIndex;         // index of min value in sublist

  while (startIndex<n)      // if >1 element, process sublist
  {
    minIndex = startIndex;  // assume first value is smallest

    for (int i=startIndex+1;i<n;i++)
    {                   // scan sublist for smaller value
      if (A[i]<A[minIndex])
        minIndex = i;
    }

    if (A[startIndex]!=A[minIndex])
    {                   // check min value not at start
      int temp = A[startIndex];     // and exchange values
      A[startIndex] = A[minIndex];
      A[minIndex] = temp;
    }

    startIndex++;       // start sublist at next element
  }
}
```

method `Select` *in* `mySort.java`

Lets consider the efficiency of both Sorts. Taking a worst case scenario we would have a list of *n* elements in completely reverse order.

Bubble sort:
pass 1: n-1 comparisons, n-1 exchanges
pass 2: n-2 comparisons, n-2 exchanges
:
pass n-1: 1 comparison, 1 exchange
in total, $\sum k = n(n-1)/2 \approx n^2$ or $O(n^2)$ comparisons and exchanges

In contrast, for the best case scenario when the list is already sorted, only 1 pass would be made, no exchanges are necessary and the sort would terminate.
In total, n-1 comparisons and 0 exchanges

Considering selection sort, in the worst case:
start index 0: n-1 comparisons (to find min), 1 exchange
start index 1: n-2 comparisons, 1 exchange
:
start index n-2: 1 comparison, 1 exchange
in total, $O(n^2)$ comparisons, n-1 exchanges

However, in the best case, we still require all the comparisons to find the min value each scan, but no exchanges are made.
In total, $O(n^2)$ comparisons and 0 exchanges.

Many students choose to use a "brute force" sort that is something like

```
for (int i=0;i<n-1;i++)
{
  for (int j=i+1,j<n;j++)
  {
    if (A[i]>A[j])
      <swap elements>;
  }
}
```

This will always require $O(n^2)$ comparisons and in the worst case, $O(n^2)$ exchanges.

Worse however, is this variation of a "brute force" search:

```
for (int i=0;i<n;i++)
  {
    for (int j=0;j<n;j++)
    {
      if (A[i]>A[j])
        <swap elements>;
    }
  }
```

because is doesn't work correctly! (why?)

12.3 Tables

Because array elements can be of any type, it is possible to have elements that are themselves arrays and build up arrays of multiple dimensions. 2-dimensional arrays are known as tables or matrices, and are the most common type of multi-dimensional array.

```
int [][] matrix = new int [3][5]
```

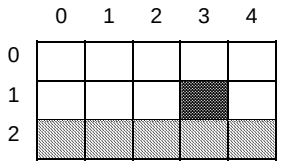
Rows are always given first, so this table would have 3 rows each with 5 columns.

A specific element in a table can be referenced using double subscripts

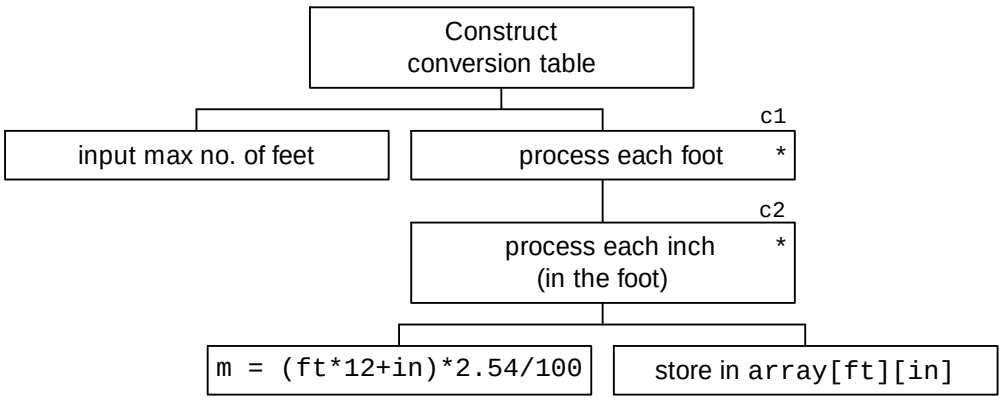
```
matrix[1][3]
```

or an entire row at a time can be referenced using just the first (row) subscript.

```
matrix[2]
```



For example, consider the use of a 2-dimensional array to store a conversion table to convert from feet and inches to metres. We can define a class, `ConvertToM`, with a constructor that generates the conversion table for a specified range of feet, a `Display` method to display the table with suitable headings, and a `toMetres` method that accepts either 2 integer parameters representing feet and inches, or 1 double parameter representing feet and fractions of a foot, or 1 integer parameter representing inches only, and for each returns the equivalent length in metres.




```
c1 : for ft from 0 to maxFt
c2 : for inch from 0 to 11
```

```
class Conversion
/* The class for a conversion table from feet, inches to metres
 * ------------------------------------------------------------
 * that stores the table and will perform conversions
 */
{
// attributes
  private  int maxFt;           // size of table
  private double [][] ftToM;    // size given when created

  Conversion (int numFt)
// constructor - sets up the table and generates the values
  {
    ftToM = new double [numFt+1][12];
              // rows for 0-numFt feet, cols for 0-11 inches
    maxFt = numFt;

    for (int ft=0;ft<=numFt;ft++)
    {
      for (int in=0;in<12;in++)
      {
        ftToM[ft][in] = (ft*12+in)*2.54/100;
      }
    }
  }

  double toMetres(int ft, int in)
  // returns the metre equivalent of the supplied parameters
  // after checking the parameters are within range
  {
    if (ft<0 || ft>maxFt || in<0 || in>11)
      return 0.0;
    else
      return ftToM[ft][in];
  }

  void Display()
  // displays the conversion table with suitable headings
  {
      :   <code not shown>
  }

}
```

part of `class Conversion` *in* `ConvertEg.java`

When dealing with 1-dimensional arrays, we considered how to test whether 2 arrays were equal.

This can be extended to 2-dimensions on the same basis - assume all elements are equal, and when the first unequal pair is found stop checking and return false. If the entire array is scanned without any unequal pair found, then the arrays are equal:

```
boolean IsEqual(int [][] arr1, int [][] arr2)
{
  boolean same = true;
  int row = -1;
  int col;
                                    // if unequal lengths
  if ((arr1.length != arr2.length)     // of rows
    || (arr1[0].length != arr2[0].length)) // or columns
    same = false;                       // they cant be equal

                                // check row by row
    while (same && (row<arr1.length))
    {
      row++;
      col = 0;                  // check each col in row
      while (same && (col<arr1[0].length))
      {
        if (arr1[row][col]==arr2[row][col])
          col++;
        else
          same = false;
      }
    }

    return same;
  }
```

Note that the number of elements declared for a column of a 2-dimensional array can be determined by ascertaining the `length` property for a row of the table

```
matrix[0].length
```

As another example, consider a simple relief map, where the data is stored in a 2-dimensional array as real numbers representing heights above (+) or below (-) sea level. One can envisage a situation where the map is an object, and there are methods to input the map, to retrieve or change specific values, to ascertain whether a given point is above or below sea level, to identify the highest point, etc, etc. Such a map can also be visually represented as an equivalent array of characters, where sea is represented by a space, land by a dot `'.'`, plateaux above 1000m by a plus `'+'`, and the highest point by a star `'*'`.

Clearly we can use a class to represent the map.
What are its *attributes*?

- a table (`altitude`) containing heights above sea level.
- The map dimensions (number of rows and cols)

And *methods*?

- to input the map from a file;
- to retrieve the height at a specified location;
- to set the height at a specified location;
- to check whether a given location is above sea level;
- to return the location of the highest point;
- to display the chart (character) array with a border;
- to generate the chart array - private, used by `showChart`;

Map
| altitude|
| nRows |
| nCols |
getMap(filename)
getHeight(row,col)
setHeight(ht,row,col)
isLand(row,col)
getHighest()
showChart()
|makeChart(chart)|

One consideration is whether or not the highest point should form an attribute. The arguments for and against are

- if it is an attribute, then we have to check and possibly update it each time method `setHeight` is called, in case the new height is the highest, or else we run the risk of our data being inconsistent.
- if it is not an attribute, the entire array has to be scanned to determine the maximum value each time `getHighest` is called.

The best approach will depend on the expected use of the system. For this exercise we will not store the highest point as an attribute, but have method `getHighest` calculate it when called.

The full program can be viewed in `Mapping.java` (contains the `Map` class and the driver class, `Mapping`) but some points of interest from the perspective of 2-dimensional processing of arrays are:

- method `getHighest` that scans all rows and columns in the array to find the max value:

```
  int getHighest()
//----------------
// determines and returns the location of the highest point
// as a single 4-digit integer rrcc (row*100+col)
  {
    int maxR=0;    // the row and
    int maxC=0;    // col coordinates of highest point
    int location;  // highest point as rrcc

    // scan array to find max value
    for (int r=0;r<nRows;r++)
    {
      for (int c=0;c<nCols;c++)
      {
        if (altitude[r][c]>altitude[maxR][maxC])
        {
          maxR = r;
          maxC = c;
        }
      }
    }

    location = maxR*100+maxC;
    return location;
  }
```

method `getHighest` from `Mapping.java`

The process is the same as for finding the maximum of a simple list, but the entire table is scanned row by row, and for each row all the columns are scanned, and the location of the "biggest-value-so-far" is updated when a larger value is found.

- method `makeChart` that constructs the visual display of the chart, by scanning every array element, testing whether the height at that position is below sea level, above sea level, or over 1000m above sea level, and storing the appropriate symbol in the corresponding element of a character array:

```
private void makeChart(char [][] chart)
//--------------------------------------
// constructs a visual representation of the altitude map where
//     space  = sea
//       .    = land<=1000m
//     +    = land>1000m
//       *    = highest point
{
    double height;   // the altitude at a specific point
    int location;        // location of highest point
    int row,col;     // coordinates of highest point

    for (int r=0;r<nRows;r++)     // for each row
    {
      for (int c=0;c<nCols;c++)       // and col position
      {
        height = getHeight(r,c);
        if      (height<0)            // get the height and store
          chart[r][c] = ' ';          // the appropriate char
        else if (height<=1000)
          chart[r][c] = '.';
        else                          // height>1000
          chart[r][c] = '+';
      }
    }

    location = getHighest();      // determine the highest point
    row = location / 100;
    col = location % 100;
    chart[row][col] = '*';        // and store * in chart location
}
```

Notice that method `getHighest` is called to return the co-ordinates of the highest point - this re-use of an existing method is desirable because it saves rewriting code to do the same thing twice, the existing method is thoroughly tested, and it ensures consistency.

In both these methods the characteristic use of nested for-loops to visit every element in a 2-dimensional table is clearly demonstrated.

The driver class instantiates a map object of a specific size, inputs data into the object from a given file, and displays the map.

```
    Map anArea = new Map(30,20);  // instantiate a 30x20 map
    anArea.getMap("MapA.dat");        // input the heights
    anArea.showChart();               // display the chart
```

Thereafter the highest point is located (`getHighest`) and it co-ordinates are displayed, and the user is asked to enter a new height and the height at that location is changed

(`setHeight`) and the new highest point is found (`getHighest` again). Finally the user is asked to enter co-ordinates of a point and `isLand` is invoked to determine whether it is above or below sea level.

Sample output from the driver program is:

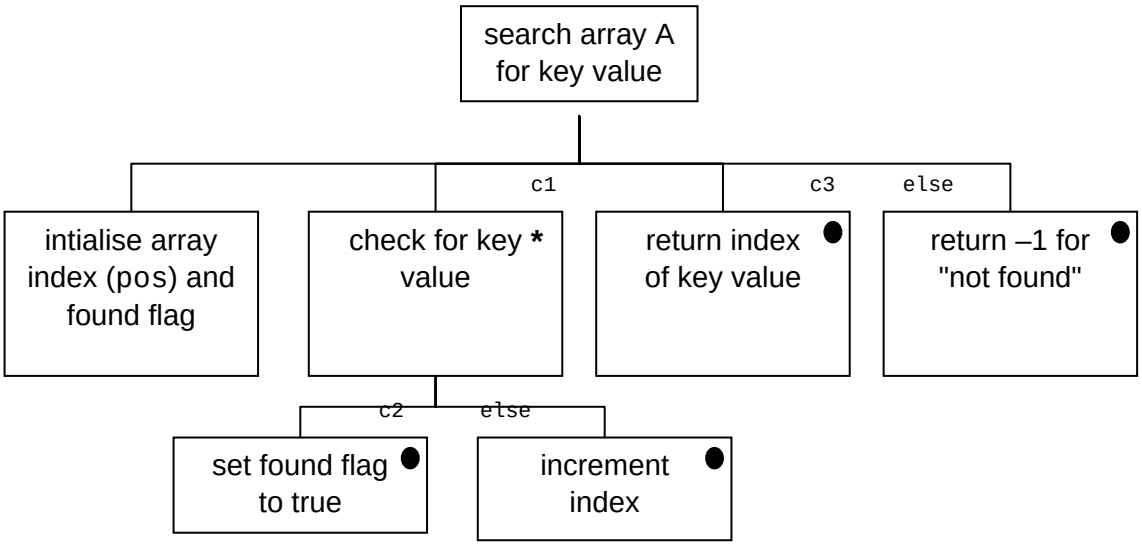
```
------------
|     .....|
|   .......|
|  ......++|
| ......++.|
|....+*+...|
|...++..  .|
|....... ..|
|++........|
------------

The highest point of 1510.0m is at position (4,5).

Enter a new height for this location > 1000

------------
|     .....|
|   .......|
|  ......++|
| ......+*.|
|....+.+...|
|...++..  .|
|....... ..|
|++........|
------------

The highest point of 1500.0m is now at position (3,8).

Enter coordinates of a point:
  row > 5
  col > 8
This point is -1.0m and
is below sea level
```

12.4 Searching

We often need to be able to search an array in order to locate a particular value, called the **key**. If the key value is found in the list, a search should return the index of the array element with the key value, but if it is not in the list this also needs to be indicated somehow. If it were possible to return 2 values for a method we could return a boolean value to indicate whether or not the key value was found, and then its (`int`) index position, but this is not possible. The simplest solution is to return only an integer value and use an out-of-range value such as –1 to indicate “not found”, and the index position if the key was found. Since array indices start at 0 there is no possibility of confusion, and the method requesting the search must check for –1 indicating “not found”, or know it was found at the given index position.

The type of approach used to search may vary depending on whether the array to be searched is unordered or in sequential order (ascending or descending).

With an unordered array the data is in random order, so we need to check every element in the array to see whether or not it is the one we are looking for, and we also need to be able to tell if the value we’re looking for does not appear in the array. The processing required is:




```
c1 : while key not found and while not at end of array
c2 : if A[pos] == key
c3 : if key found
```

The method to perform a search of an unordered array is shown below. It forms part of class `mySearch` which contains a number of searching algorithms.

```
public static int Unordered (int key, int [] A, int n)
//-----------------------------------------------------
// Implements a full search for the key value in the n-element,
// unordered array, A. All elements in the array are checked
// until the key is found or the end of the array is reached.
// If key is found its position is returned, if not an
// out-of-range value (-1) is returned.

{
  int pos = 0;          // the position being checked in A
  boolean found=false;      // whether or not key has been found

  while (!found && pos<n)   // check each position until found
  {
    if (A[pos] == key)
      found = true;
    else
      pos++;
  }

  if (found)            // return either
    return pos;         //   the array index for key
  else                  // or
    return -1;          //   -1 for "not found"

}
```

method `Unordered` *in* `mySearch.java`

If we know the array has been sorted (for example, into ascending order), we only need to check through the array until we find the key value, or until a value smaller than the key is found. This latter means that the key value is not contained in the array, since it would have been found by now because the elements are in order and it should have been before the smaller value we're now checking. This means that the search can terminate sooner, and not all elements have to be checked to be sure that the key is not in the list.

```
public static int Linear (int key, int [] A, int n)
//-----------------------------------------------------
// Implements a linear search for the key val in the n-element,
// ascending array, A. The array is checked from the beginning
// until the key is found or a value larger than the key is
// reached. If key is found its position is returned, if not an
// out-of-range value (-1) is returned.
{
  int pos = 0;          // the position being checked in A
  boolean found=false;      // whether or not key has been found

  while (!found && pos<n && A[pos]<=key)
  {                     // check values in range until found
    if (A[pos] == key)
      found = true;
    else
      pos++;
  }

  if (found)            // return either
    return pos;         //   the array index for key
  else                  // or
    return -1;          //   -1 for "not found"
  }
```

method `Linear` *in* `mySearch.java`

With a linear search, if there are n elements in the array then on average n/2 elements need to be examined to either locate the key value or to ascertain that it is not in the array – ie. the work done is O(n/2). With the unordered search, if the key is in the array the work done is O(n/2), but is O(n) if the key is not in the array. In both cases, the execution time increases linearly with the number of elements in the array, which is a problem for large n.

A more efficient searching technique is the **binary search**, which is only applicable to sorted arrays. It takes advantage of the fact the array is ordered to eliminate half the elements under consideration each pass. Its approach is similar to the way we find a word in a dictionary, for example. We open the dictionary and see if the word we are looking for is before or after page we are at. If before we only consider the first part of the dictionary and try again, if after we only consider the last part of the dictionary and try again. In other words, we reduce the search space each iteration.

For example, assume we are searching the following sequence of values for the key 61:

```
11  21  24  35  36  41  49  53  57  60  61  68  69  72  81
```

We check the middle element (53) to see whether 61 is in the first or last half of the list.
Its greater than 53 so we split the list in two and consider only the sublist

```
57  60  61  68  69  72  81
```

Again we check the middle element (68), and 61 is less than 68 so we split the list again and consider only the lower half:

```
57  60  61
```

Again we check the middle element (60), and 61 is greater than 60 so we split the list again and consider only the last half:

```
61
```

which is the key value we are searching for.

This search took just 4 probes to find the value, compared with the 10 tries needed with a linear search. In fact, the average work required is $O(\log_2 n)$.

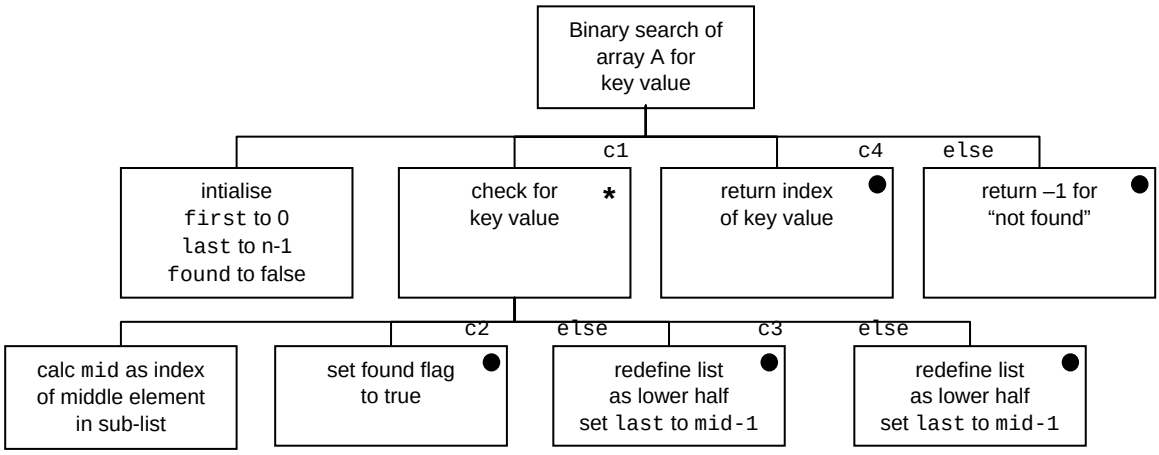
If the key value is not in the list the search results in an empty list. Consider searching for the key 62. The search proceeds as before up to the 4th probe when we have the single element list

```
61
```

This is also the middle (only) element, and 62 is greater than 61 so we split the list again, considering only the upper half, and end up with an empty list.

The actual process involved is:

- initialise `first` to `0` and `last` to `n` (or `A.length`)
- calculate the middle position as `middle = (first+last)/2`
- if the key is equal to the value of the middle element then the key has been found;
  if the key is greater than the middle element, we need to search the upper half:
  set `first to middle+1`
  if the key is less than the middle element, we need to search the lower half:
  set `last to middle-1`
- recalculate the middle position as `middle = (first+last)/2`
  and repeat these last two steps until …..
  - either the key is equal to the value of the middle element and the search is over,
  - or the list to be considered is empty (`first>last`) and the key is not in the list.




```
c1 : while key not found and at least 1 element in sub-
list
c2 : if A[mid] == key
c3 : if A[mid] > key
c4 : if key found
```

The method to perform a binary search is shown below. It also forms part of class `mySearch`.

```
public static int Binary (int key, int [] A, int n)
//-----------------------------------------------------
// Implements a binary search for the key val in the n-element,
// ascending array, A. The array is repeatedly split in two,
// with only the half containing the key value being considered
// each iteration. If key is found its position is returned, if
// not an out-of-range value (-1) is returned.
{
  int first,last,mid=0;     // indexes of the current sublist
  boolean found = false;    // whether key has been found

  first = 0;
  last = n-1;

  while (!found && first<=last)
  {                         // with sublist of >1 element
    mid = (first+last)/2;
    if (A[mid] == key)          // key is at mid position
      found = true;
    else if (A[mid] > key)      // key is in lower half
      last = mid-1;
    else //A[mid] < key     // key is in upper half
      first = mid+1;
  }

  if (found)                // return either
    return mid;             //   the array index for key
  else                      // or
    return -1;              //   -1 for "not found"
  }
```

method `Binary` *in* `mySearch.java`

Exercises

12.1 Write a class, NumArray, with data member an array of integers, and write methods to input the array; to calculate the average of it elements; to find the maximum and minimum elements; to check whether all values are positive; to multiply it by a scalar value; and to add to it another instance of the NumArray class.

12.2 Write a program that reads in a student number (6digits) and a % mark for each of N students (input N first; or use a sentinel value; or read from a file), and outputs the student numbers of those students who obtained above the average mark. Do not use a sort in your solution.

12.3 Write a program that, as above, reads in a student number (6digits) and a % mark for each of N students (input N first; or use a sentinel value; or read from a file), and determines what the highest pass mark should be that will allow at least 75% of the students to pass – ie, at least 75% of the students should obtain that mark or greater. Do not use a sort.

12.4 A certain instructor awards letter grades to student papers having numeric scores in the following manner:

- the papers with the highest and lowest marks are found, thus determining the range
- papers with marks in the top 25% of the range are awarded an A, papers with marks in the lowest 30% of the range get a C, and the rest get a B.

Write a program that reads in student number (6digits) and % marks for each of N students (input N first; or use a sentinel value; or read from a file), and outputs the student together with the letter grade obtained. Do not use a sort in your solution – the student numbers output must be in the original order.

12.5 Write a class, Marks, which has as data member an array of student numbers and of marks (%) and write methods to input the arrays; output only those that are above average (3.2); determine the pass mark for a 75% pass rate (3.3), and assign letter grade to the marks (3.4)

12.6 Write a method that will sort the student number array into ascending numeric order, and order the mark array in the same way (so that the first student number corresponds to the first mark etc)

12.7 Write a method that will determine the median mark and output the median mark and the student who achieved it. The median mark is the mark obtained by the middle student – ie, if there are 31 students, the mark obtained by the student who came 16^{th}.

12.8 Write a program that will read N integer values into an array and then delete duplicate copies of any numbers that appear more than once and move the unique values towards the beginning, preserving their order. (ie, no sort at this stage). Finally sort the list of numbers into descending numeric order. Your program should use only one array to hold the numbers, and the array should be displayed after each stage of the program.

12.9 Write a class, BinaryNum that has as data member an array that represents a 16-bit binary value using 2's complement notation. Write methods that will input an integer

value and store it as a binary number, or that converts the binary number to a decimal integer.

12.10 Write a matrix class that will input a $n_x n$ matrix, and with methods to check whether it is diagonal ($a_{ij}=0$ for $i \neq j$); upper triangular ($a_{ij}=0$ for $i>j$); lower triangular ($a_{ij}=0$ for $i<j$);

12.11 Write a program to do matrix multiplication

12.12 Write a program to generate and output Pascals triangle, where the size of the triangle N is input (N<10). For N=6, Pascal's triangle is

```
     1
    1 1
   1 2 1
  1 3 3 1
 1 4 6 4 1
1 5 10 10 5 1
```

12.13 Write a method that extends 3.6 by reading a student number, and if the student number exists in the array, ask for a new mark and change the mark stored for that student.

12.14 Using the same data as in 3.13, and without re-sorting it into mark order, output the student number of a student who obtained 90%

13. Strings

We have already encountered string literals or constants which are lists of characters enclosed in double quotes, such as `"Hello there!"` In Java, `String` is a class, so strings are objects and not variables. There are in fact 2 classes for manipulating strings: class `String` for which the contents, once assigned, may not be changed, and class `StringBuffer` which enables changes to the contents of strings. Java distinguishes constant strings and modifiable strings for optimisation purposes – in particular, Java can perform certain optimisations involving `String` objects because it knows these objects will not change.

13.1 Strings

There is a special notation for creating strings with known contents – unlike other objects the `new` operator is not needed. A string is created by assigning it another string, or a character array, or a `StringBuffer`. For example, if the following variables are all declared as `String`,

```
aString = "Hello";
bString = "";
cString = Keyboard.readString();
```

are all valid assignments.

A string variable can be assigned different string values at different times - it is not a constant - but once assigned, the characters comprising the string may not be changed. The entire string must be reassigned as an entity.

As with all objects, a string instance must be declared before it is used, and may be initialised at the same time.

```
String bString = "";
String dString;
```

There is a difference between an empty (but initialised) string, and a null (ie. not initialised) string. In the example above, `dString` is uninitialised and will have a special value called null until it is given some other string value in an assignment statement. Such a string is not valid for string manipulation methods, and if an attempt is made to access it, a `NullPointerException` will be thrown by the system. On the other hand, `bString` is initialised and can be used in string manipulation – it just has no contents.

Like arrays, strings know their own size, and a method `length()` will give the length of the string. Note that with arrays, `length` is an instance variable and is accessed as `myArray.length`. With strings, `length()` is a method call, and brackets are required:

```
int size = myString.length();
```

A string is made up of a sequence of characters, and although it cannot be indexed directly (eg `myString[7]`), the character at any index position can be obtained using the method `charAt(int)`.

```
String myName = "Percival";
char initial = myName.charAt(0);
```

If an attempt is made to access a character outside string bounds (i.e. <0 or ≥length), a `StringIndexOutOfBoundsException` is obtained.

Methods `toLowerCase()` and `toUpperCase()` return a new string composed of all lower case or upper case characters respectively. Special characters are unaffected, and the original string is unchanged, unless it is reassigned to the new string.

For example,

```
String original,upper,lower;
original = "Computer Science 1B";
upper = original.toUpperCase();
lower = original.toLowerCase();
System.out.println("original string: " + original
                + "\n  in upper case: " + upper
                + "\n  in lower case: " + lower);
```

will output

```
original string: Computer Science 1B
  in upper case: COMPUTER SCIENCE 1B
  in lower case: computer science 1b
```

Another useful method is `trim()`, which returns a new string without any of the whitespace characters (blanks, newlines or tabs) that may appear at the beginning or end of the original string. Internal whitespace is untouched.

For example

```
String original,trimmed;
original = "   Computer Science 1B   ";
trimmed = original.trim();
System.out.println("original string: *" + original + "*"
                + "\nafter trimming : *" + trimmed + "*");
```

will output

```
original string: *   Computer Science 1B   *
after trimming : *Computer Science 1B*
```

Strings may be compared in two ways. The boolean methods `equals(String)` and `equalsIgnoreCase(String)` will return true or false depending on whether the string supplied as argument is equal to the string instance invoking the method, or not.

For example

```
aString = "CS1A";
bString = "CS1B";
cString = "cs1a";

if (aString.equals(bString))
  System.out.println(aString+" equals "+bString);
else
  System.out.println(aString+" does not equal "+bString);

if (aString.equals(cString))
  System.out.println(aString+" equals "+cString);
else
  System.out.println(aString+" does not equal "+cString);

if (aString.equalsIgnoreCase(cString))
  System.out.println(aString+" equals (no case) "+cString);
else
  System.out.println(aString+" does not equal "+cString);
```

part of `CompareStrings.java`

will output

```
CS1A does not equal CS1B
CS1A does not equal cs1a
CS1A equals (no case) cs1a
```

Note that the equals operator (= or <, > for that matter) should not be used with objects, only with variables. The reason will be explained more fully in a later section, but is to do with the fact that objects are stored as references to data values, while variables store the values themselves. Using `=` to compare two objects will compare the references of the two objects – *do they refer to the same storage?*, not *are the values equal?*

On the other hand the `compareTo(String)` method compares the 2 strings on their respective UNICODE codes, and returns an integer result, where 0 means the two strings are equal, a negative number means the string that invoked `compareTo` is less than the string passed as argument, and a positive number means the string that invoked `compareTo` is greater than the string passed as argument (the actual number is the code difference between the first pair of unequal characters). The method works by comparing pairs of characters from the left, and the first unequal pair determines the result.

For example,

```
aString = "hello";
bString = "Hello";
cString = "goodbye";
dString = "good";

System.out.println("Comparing "+aString+" and "+bString+": "
  + aString.compareTo(bString));
System.out.println("Comparing "+bString+" and "+cString+": "
  + bString.compareTo(cString));
System.out.println("Comparing "+cString+" and "+dString+": "
  + cString.compareTo(dString));
System.out.println("Comparing "+dString+" and "+dString+": "
  + dString.compareTo(dString));
```

part of `CompareStrings.java`

will output

```
Comparing hello and Hello : 32
Comparing Hello and goodbye : -31
Comparing goodbye and good : 3
Comparing good and good : 0
```

For the first, the code for `'h'` is 104, while that for `'H'` is 72, hence `"hello"` is greater than `"Hello"`. In the next, the code for `'H'` is 72, while that for `'g'` is 103, hence `"Hello"` is smaller than `"goodbye"`. In the third, the first 4 chars match, but `"goodbye"` is longer so is considered greater than `"good"`, while the last pair register as equal.

Other useful methods are

`indexOf(string)` and `indexOf(string,pos)`

both of which return the index of the start of the substring specified as argument, or –1 if the substring is not found. The second version specifies the position at which to start the search.

For example,

```
aName = "I love computing and computers";
System.out.print("put is at index " + aName.indexOf("put"));
```

would display

```
put is at index 10
```

while

```
System.out.print("put is at index " + aName.indexOf("put",15));
```

would display

```
put is at index 21
```

If you know the start location of a substring, it can be copied into another string using the method `substring(startindex)` or `substring(startindex,endindex)`, where `startindex` is the index position of the start of the substring, and `endindex` is the index position after the last position to be copied. If the endindex is not specified, the remainder of the string is copied. For example, we could use a combination of `indexOf` and `substring` to split string containing a name into the firstname and surname:

```
String aName,firstname,lastname;
int pos;

System.out.println();
System.out.print("Enter a name as firstname space surname : ");
aName = Keyboard.readString();

pos = aName.indexOf(" ");
if (pos<0)
  System.out.println("no space entered");
else
{
  firstname = aName.substring(0,pos);
  lastname = aName.substring(pos+1);
  System.out.println("First name is " + firstname
    + " and surname is " + lastname);
}
```

SplitString.java

This displays

```
Enter a name as firstname space surname : Nelson Mandela
First name is Nelson and surname is Mandela
```

Two strings can be joined together or concatenated using the `concat(string2)` method, which returns a new string consisting of the argument string (`string2`) joined to the end of the invoking string.

For example,

```
aString = "abcd";
bString = "xyz";
cString = aString.concat(bString);
```

will store `"abcdxyz"` in `cString`, while

```
cString = bString.concat(aString);
```

will store `"xyzabcd"` in `cString`.

Note that the operators `+` and `+=` have been overloaded for strings to perform concatenation, and hence one can use statements such as

```
cString = aString + bString;       // "abcdxyz"
cString += aString;                // "abcdxyzabcd"
```

To summarise, the string methods discussed above are tabulated below. There are a number of other methods, as well as variations on these methods (different arguments etc), but the ones shown here are the more common and useful methods.

STRING METHODS	
`length()`	returns the (`int`) length of the string
`charAt(int)`	returns the char at the given index
`toUppercase()`	returns the string as upper case
`toLowerCase()`	returns the string as lower case
`trim()`	returns the string with leading and trailing whitespace removed
`equals(String)`	returns true if the string argument equals the invoking string, and false otherwise
`equalsIgnoreCase(String)`	returns true if the string argument equals the invoking string, ignoring case, and false otherwise
`compareTo(String)`	returns a negative, zero or positive `int` to indicate the invoking string is less than, equal to, or greater than the argument string
`indexOf(String{,int})`	returns the index position of the argument string in the invoking string
`substring(int{,int})`	returns the substring starting at the given index in the invoking string
`concat(String)`	returns a new string consisting of the invoking string with the argument string joined at the end.

Example
Read in a list of names and sort them into alphabetical order.

If we have an array of strings, the standard sorting algorithms can be adapted to compare strings and sort the array into alphabetical order.

```
while (startIndex<n)        // if >1 element, process sublist
{
  minIndex = startIndex;    // assume first value is smallest
                          // scan sublist for smaller value
  for (int i=startIndex+1;i<n;i++)
  {
    comp = names[i].compareTo(names[minIndex]);
    if (comp<0)
      minIndex = i;
  }
                  // check min value not at start of sublist
  if (startIndex!= minIndex)
  {
    String temp = names[startIndex];         // exchange values
    names[startIndex] = names[minIndex];
    names[minIndex] = temp;
  }

  startIndex++;         // start sublist at next element
}
```

part of `SortStrings.java`

13.2 String buffers

As has been mentioned before, the contents of a string cannot be modified. If we need to change the contents or length of a string, we need to break it up, rejoin it and assign it to a new string, or to use the alternative `StringBuffer` class. This provides methods to let you change the length of a string of characters, modify individual characters, insert characters etc. However similar it appears, a `StringBuffer` is not the same as a `String` - it is created differently, and the `String` methods may not be used with `StringBuffers`.

Every StringBuffer has a capacity and a length. The capacity defines the number of characters that can be stored in the StringBuffer, while the length records how many characters are currently stored. If the capacity is exceeded, it is automatically expanded to accommodate the additional characters.

To instantiate a `StringBuffer` object, the `new` operator must be used.
Eg

```
aBuffer = new StringBuffer();       no contents, initial capacity is 16 (default)
bBuffer = new StringBuffer(10);     no contents, initial capacity as given
cBuffer = new StringBuffer("Hello there!");
                                    initial capacity is length of contents + 16
```

Methods `length()` and `capacity()` return the number of characters currently stored in a StringBuffer, and the number of characters that can be stored in a StringBuffer without increasing its size. For example (`StrBuffDemo.java`),

```
aBuffer.length() returns 0,   aBuffer.capacity() returns 16
bBuffer.length() returns 0,   bBuffer.capacity() returns 10
cBuffer.length() returns 12,  cBuffer.capacity() returns 28
```

As with strings, the character at any index position can be obtained using the method `charAt(int)`. However a method, `setCharAt(int,char)` is provided for StringBuffers to replace the character in the specified index position with the character given.
Eg.

```
StringBuffer myName = new StringBuffer("Sally");
myName.setCharAt(1,'i');
System.out.println("My name is " + myName);
```

will display

```
My name is Silly
```

If an attempt is made to access a character outside the StringBuffer bounds (ie. <0 or ≥length), a `StringIndexOutOfBoundsException` is obtained. For example, if the statement above was

```
myName.setCharAt(11,'i');
```

the following occurs

```
java.lang.StringIndexOutOfBoundsException:
String index out of range: 11
```

The append methods allow new characters to be added at the end of a StringBuffer. Any type of data can be appended, and the append method converts it to a string and adds it to the end of the invoking string.
E.g.

```
StringBuffer myBuffer = new StringBuffer("Hello");
myBuffer.append(" there. ");          // a string
myBuffer.append(12345);               // an int
myBuffer.append(' ');                 // a char
myBuffer.append(999.99);              // a double
```

Displaying the buffer, the length and capacity after each append (StrBuffDemo2.java) gives

```
"Hello"  length: 5  capacity: 21
"Hello there. "  length: 13  capacity: 21
"Hello there. 12345"  length: 18  capacity: 21
"Hello there. 12345 "  length: 19  capacity: 21
"Hello there. 12345 999.99"  length: 25  capacity: 44
```

When an increase in capacity is needed, the new allocation is 2xcapacity + 2.

StringBuffer provides insert methods to allow various data type values to be inserted in any position in a StringBuffer.

```
insert (index,value);
```

inserts the data value (converted to a string) starting at the index specified (which must be ≥0 and ≤ the length of the StringBuffer.
E.g.

```
StringBuffer myBuff = new StringBuffer("Jo Brown");
myBuff.insert(2,"nathon");               // a string
myBuff.insert(0,12345);                  // an int
myBuff.insert(5,'-');                    // a char
myBuff.insert(myBuffer.length(),'-');    // another char at end
myBuff.insert(myBuffer.length(),99.99);   // a double
```

Displaying the buffer, the length and capacity after each append (StrBuffDemo3.java) gives

```
"Jo Brown"  length: 8  capacity: 24
"Jonathon Brown"  length: 14  capacity: 24
"12345Jonathon Brown"  length: 19  capacity: 24
"12345-Jonathon Brown"  length: 20  capacity: 24
"12345-Jonathon Brown-"  length: 21  capacity: 24
"12345-Jonathon Brown-99.99"  length: 26  capacity: 50
```

13.3 Tokenizers

When you read a sentence, your mind breaks it up into its component words, or tokens, each of which convey meaning to you. On the other hand, a sentence stored in a Java string is just a sequence of characters, with no inherent meaning ascribed to portions of the string. Java strings do not differentiate between alphabetic characters, digits, spaces, control characters – they are all merely individual characters in a string. However, Java does supply a `StringTokenizer` class (in `java.util`) that breaks a string into its component portions (*tokens*), demarcated by whitespace. (Whitespace refers to the delimeters space,tab,newline and carriage return.)

So, for example, given a string such as

```
myString = "Happy birthday to you";
```

we can extract the tokens as the individual strings

```
"Happy", "birthday", "to", "you"
```

To do this we first declare a tokenizer on the string

```
StringTokenizer myTokens = new StringTokenizer(myString);
```

Then we can look through the tokens, checking for the end, and store each token in separate elements of an array of strings (or output them, or process them, etc) as demonstrated in `TokenDemo.java`:

```
nWords = 0;
while (myTokens.hasMoreTokens())
{
  words[nWords] = myTokens.nextoken();
  nWords++;
}
```

Some output from this demo program is

```
Enter a sentence of at most 20 words:
I have a dog, her name is Meg.

I
have
a
dog,
her
name
is
Meg.
```

Notice the punctuation included as part of each token. The tokenizer constructor also allows you to specify a string containing your delimiters (the default one is `" \n\t\r"`), as in

```
myTokens = new StringTokenizer(myString," .,\n\t\r");
```

which defines the delimiters as comma and full stop as well as the defaults.

Using this constructor would give the output as

```
I
have
a
dog
her
name
is
Meg
```

13.4 classes Keyboard and GenIO

Up till now, all input from the keyboard has been using a class, `Utilities.Keyboard`, which was written to simplify input. We now have the tools to examine the methods of this class and see how they are written and also to extend it to cater for more than one data value to be entered on a line.

For example, consider `getInt()`:

```
private static BufferedReader br =
      new BufferedReader(new InputStreamReader(System.in));

public static int getInt()
//-------------------------
  {
    int localInt = 0;
    try
    {
      localInt =
           Integer.valueOf(br.readLine().trim()).intValue();
    }
    catch (IOException e)
    {
      System.out.println();
      System.out.println("IO exception: " + e.getMessage());
      System.out.println();
    }
    catch (RuntimeException e)
    {
      System.out.println();
      System.out.println("Error entering int: " +
                                          e.getMessage());
      System.out.println();
    }
    return localInt;
  }
```

`getInt()` *from* `Keyboard.java`

The first statement instantiates an object `br` of type `BufferedReader`. Recall from chapter 2 (Streams and Exceptions), that this declares a new stream, connected to the standard input device `System.in`, with all the facilities of the `BufferedReader` class, in particular readLine( ) which returns a string of data from the default input device, `System.in`, up to the first newline or carriage return code.

Analysing the actual read statement:

```
localInt = Integer.valueOf(br.readLine().trim()).intValue();
```

`readline()` returns a string from the keyboard,

then the string method `trim` is invoked for this string object

`trim()` removes any whitespace from either end of the string,

the resultant string is passed as argument to the Integer class method

`valueOf` which converts the string to an Integer object

and finally the Integer method `intValue` is invoked for this Integer object

`intValue()` returns the corresponding `int` value.

This is probably an appropriate place to point out that each numeric datatype has a corresponding class for use when an object as opposed to a variable of the type is required. One of the strict rules in Java is that variables and objects cannot be mixed. Usually this is no problem, but there are times when a standard package requires an object and we wish to send it a variable, say of type `int`. In order to do so, it is first placed in an envelope class, making it into an object. The envelope class provides access to the value and also has various conversion methods available.

These envelope classes are `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Character`, `Float` and `Double`, and are in Java's `lang` package. They provide class-level versions of the respective primitive data types.

Many methods are available - the important ones are (equivalents available for the other classes):

Constructors:

```
Integer (int value);
Integer (String s);
Double (double value);
Double (String s);
```

Class methods

```
static Integer valueOf(String s);       // returns s as given class
static Double valueOf(String s);
static String toString(int x);          // returns x as a String
static String toString(double x);
```

Some newer class methods:

```
static int parseInt(String s)           // returns s as an int
static double parseDouble(String s)     // returns s as a double
```

Instance methods

```
int intValue();                  // returns required value of instantiating class
double doubleValue();
```

In order to enter more than one value on a line, an entire line must be read into a string and the individual tokens extracted and dealt with.

```
S = br.readLine();
if (S==null)     // no more data in file
{
  System.err.println("End of file detected");
  throw new EOFException();
}
T = new StringTokenizer(S);
```

The basic input class `Keyboard` and the file input class `FileIO` have been extended and enhanced to provide general input functionality including the ability to handle more than one value per line. For example, the processing in `GenIO`'s method `getDouble()` is

```
double myDouble = 0.0;
String item;

if (T==null) getNewLine(br);
while (true)
{
  try
  {
    item = T.nextToken();
    myDouble = Double.valueOf(item.trim()).doubleValue();
    return myDouble;
  }
  catch (NoSuchElementException e)
  {
    getNewLine(br);
  }
  catch (NumberFormatException e)
  { … }
}
```

If there is no stringTokenizer (ie. no data has yet been read), get the first line. Then extract a token from the line and convert it to a `double` value and return it. However, if there are no available tokens, (`NoSuchElementException`) get a new line, and if the value is not a legal double, (`NumberFormatException`) take appropriate action.

The class `GenIO` also contains a number of other useful input and output methods. The full specification is:

```
public static BufferedReader open (InputStream in)
```
creates a buffered reader object for the standard input stream

```
public static BufferedReader open (String filename)
                                    throws FileNotFoundException
```
creates a buffered reader object for the specified file input stream

```
public static PrintWriter create (String filename)
                                    throws IOException
```
creates a print writer object for the specified file output stream

```
public static long getLong(BufferedReader br) throws IOException
```
inputs a long value

```
public static int getInt(BufferedReader br)  throws IOException
```
inputs an int value

```
public static float getFloat(BufferedReader br)
                               throws IOException
```
inputs a float value

```
public static double getDouble(BufferedReader br)
                                 throws IOException
```
inputs a double value

```
public static char getChar(BufferedReader br) throws IOException
```
intended to read in a single char at a time (ie. individual tokens), otherwise read in a string and extract the chars

```
public static String readString(BufferedReader br)
                                   throws IOException
```
intended to read an entire line, including spaces

```
public static String readStrToEol(BufferedReader br)
                                    throws IOException
```
intended to read the rest of the line, including spaces

```
public static String readStrWord(BufferedReader br)
                                   throws IOException
```
intended to read the next string token

```
public static void getNewLine(BufferedReader br)
                                 throws IOException
```
inputs the next data line for processing

```
public static String format(long l,int toWidth)
```
outputs a long in toWidth places

`public static String format(int i,int  toWidth)`
outputs an int in toWidth places

`public static String format(double d,int toWidth,int decWidth)`
outputs a double in toWidth places, with decWidth decimal places.
If decWidth is 0, outputs as a long.

`public static String format(float f,int toWidth,int decWidth)`
outputs a float in toWidth places, with decWidth decimal places
If decWidth is 0, outputs as an int.

`public static String alignL(String S,int toWidth)`
outputs a string left aligned in toWidth places

`public static String alignR(String S,int toWidth)`
outputs a string right aligned in toWidth places

13.5 toString methods

In order for Java's `print` and `println` methods to output variables of different data types and classes, the format and required representation of these variables must be known. Java "knows" how to display the standard data types. For the non-standard types, one option is to write an explicit display method to output values together with any formatting, heading info etc. A more general option is to write a `toString` method, which returns the data values expressed as a string.

For example, consider the statement

```
System.out.println("The answer is " + x);
```

How this is dealt with depends on what x is.

- If `x` is a standard type (e.g. `int` or `double`) then its value will be displayed

- If `x` is a variable of one of Java's standard classes, the class specification includes a `toString` method which expresses this data value as a string, so its value will automatically be displayed
  eg

```
Date x = new Date();

// output current time
System.out.println("\nCurrent time is " + x);
```

  would display

```
Current time is Fri Jul 13 12:16:15 GMT+02:00 2001
```

  In this case the Date class's `toString` method represents the value stored in variable x (actually the number of milliseconds since Jan 1 1970) as day of week, date and time as shown.

- If x is a variable of a user-defined class, any attempt to display will result in output such as

```
The answer is ClassName@607474cd
```

which is not much use.

The print methods output a string. If a class variable is included to be output then Java attempts to convert it to a string, and looks for an instance method called `toString`. This method is by convention included in all Java library classes to enable the data to be easily output. If you are writing your own class you should write a `toString` method for the class, so that data values can be converted to a string representation and easily displayed or sent to file or whatever.

Recall the Quadratic equation class discussed earlier. This had an explicit display method consisting of a print statement to display the quadratic equation object.

```
void Display()
// Displays the equation
{
    System.out.print(a+ "x^2 + " +b+ "x + " +c+ " = 0");
}
```

This method was then invoked to display the equation

```
// create object called myEq
QuadEq myEq = new QuadEq(aa,bb,cc);

System.out.println("The quadratic equation ");
// invoke the class's Display method to display myEq
myEq.Display();
```

Using the approach of writing a general `toString` method:

```
public String toString()
// Represent the equation as a string
{
    String myString = "" +a+ "x^2 + " +b+ "x + " +c+ " = 0";
    return myString;
}
```

Then the object is merely included in the output statement:

```
// create object called myEq
QuadEq myEq = new QuadEq(aa,bb,cc);

// display myEq (automatically invokes toString)
System.out.print("The quadratic equation "+myEq);
```

It is useful to note that the `format` method in classes `Formatter, FileIO` and `GenIO` returns a String, so can be used within a `toString` method in building up a representation of a data value in the correct format.

Exercises

13.1 Write a program that reads in a string and checks whether it is a palindrome – reads the same forwards and backwards. You should consider only alphabetic characters – ie ignore spaces and punctuation. For example, *"Madam, I'm Adam"* is a palindrome, as is *"A man, a plan, a canal, Panama!"*

13.2 Write a program to read in a piece of text (over several lines), terminated with `***`, and to determine the number of words it contains and the average word length. You may assume the text consists only of alphabetic characters, commas and full stops, and that the words are separated by at least 1 space.

13.3 Write a program that reads in a sentence and displays only the **unique** words (irrespective of case) that it contains. Exclude all common punctuation such as `.,!?-():';"`. For example, if the sentence

```
Consider the Dog, the Cat and the Mouse. The cat is black,
the mouse is brown, and the dog is brown and black!
```

were entered, the output should be

```
Consider
the
Dog
Cat
and
Mouse
is
black
brown
```

Extension – display your output in alphabetical order.

13.4 Write a program which reads in a piece of text (over several lines) and makes it feminine by replacing all occurrences of `he` and `his` with `she` and `her` respectively.

13.5 Write a program which reads in a piece of text (over several lines), ending with followed by a line with 2 words only on it, and exchanges all occurrences of the 2 words in the text. You will need a sentinel to indicate the end of the text and hence the 2 words.
For example,

input:
```
Jack and Jill went up the hill
to fetch a pail of water,
Jack fell down and broke the crown
and Jill came tumbling after.
***  Jack  Jill
```

output:
```
Jill and Jack went up the hill
to fetch a pail of water,
Jill fell down and broke the crown
and Jack came tumbling after.
```

13.6 Write a program which reads in two strings (from 2 data lines) and finds the largest common subsequence, and its starting position in both strings.

eg. input
```
abcdefefdcbabcd
xyzababaxy
```
output
```
largest common subsequence is "bab"
found at 10 in line 1, and 4 in line 2.
```

13.7 Write a program to read in arithmetic expressions, one to a line, as strings, and evaluate them. The expressions consist of only 2 operands with the operators +,-,*,/.

a) assume the operands and operator are separated by spaces
   eg. `123 + 456`

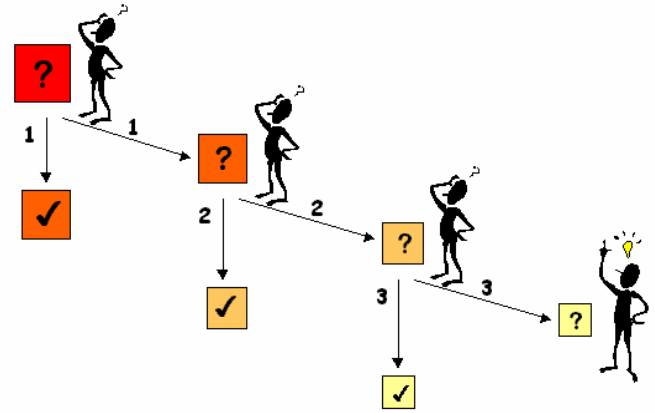
b) assume there are no spaces in the expression
   eg. `79/24`

Report any errors.

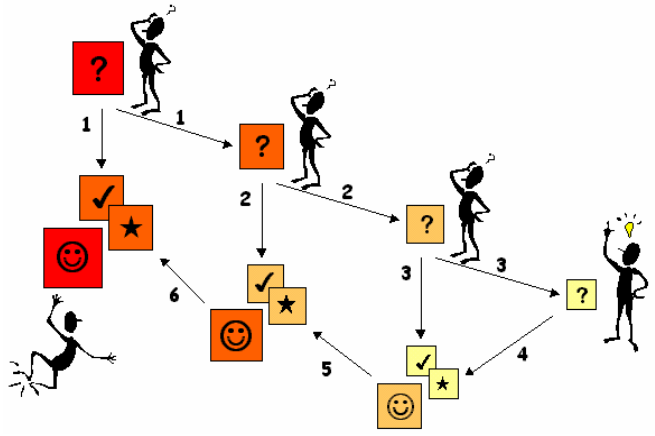
14. Recursion

Recursion is the ability of a method to call itself. This can be a powerful programming technique as in many instances using recursion enables us to specify a natural, simple solution to a problem that would otherwise be difficult to solve. Recursion can often be used as an alternative to iteration, but a **recursive solution is generally <u>less efficient</u> than an iterative solution** because of the overhead of the extra method calls.

Problem solving using recursion has a common approach. A recursive method is called to solve the problem. This method actually only knows how to solve the simplest case, called the base case, and if it is called with the base case, it returns a result. If however it is called with a more complex problem it can't solve immediately, but expresses it in terms of a part that it can do and the part that it cannot. To make recursion feasible, the part it cannot do must resemble the original problem but a slightly simpler or smaller version of this problem. Because this new problem looks like the original problem, the method launches a fresh copy of itself to go to work on the smaller problem – this is the recursive call. Of course this process repeats – if it is not the base case it is broken into two parts and another recursive call is made … etc.
Because each recursive step is made with the slightly simpler version of the original problem, at some point the stage will be reached where the solution is known (the base case), and this answer must be returned to the preceding call which combines it with the bit it knows, and returns to the preceding call … etc. until the starting point is reached and the final solution is known. So each recursive step must contain a `return`, to send back its solution to the previous level.



The recursion step executes while the original call to the method is still open – i.e. it has not finished executing. The recursion step can result in many more recursive calls as the method divides each new subproblem into 2 conceptual pieces and launches another recursive call.



Problems that lend themselves to recursive solutions have the following characteristics:

- One or more simple cases of the problem (base or stopping cases) have a simple solution that is known.
- The other cases can be expressed in terms of a combination of a known part and a simpler case of the original problem that is "closer" to the stopping case.
- Eventually the problem can be reduced to the base case(s), which can be solved – i.e., the simplifications of the original problem converge to a base case.

The usual form of a recursive method is

```
if (this is the base case)
{
    return solution;
}
else
{
    determine known part;
    recursive call to get solution to simpler case;
    return known part combined with recursive result;
}
```

Consider the problem of finding the factorial of a number, N, ($N \geq 0$).
A factorial (!) can be defined as:

$1! = 1$

$N! = N \times (N-1)!$

This recurrence relation defines a base case (N=1) with a simple solution, and expresses all other cases as a combination of a known part ($N\times$) and a simpler case of the original problem ((N-1)!).

Writing a recursive method to calculate a factorial:

```
long factorial(int N)
{
    if (N==1)    // the base case
    {
        return 1;
    }
    else                 // recursive step
    {
        long ans = factorial(N-1);    } usually written as
        return N*ans;                 } return N*factorial(N-1)
    }
}
```

If we call this method in a program that inputs a value for N and calls factorial(N) to calculate its factorial, execution will proceed as follows : *(additional output statements were included at the beginning and end of the recursion method to trace the execution)*

```
Calculating factorials
----------------------

Enter number for which to calculate factorial : 5

          Entering  with N=5
            Entering  with N=4
              Entering  with N=3
                Entering  with N=2
                  Entering  with N=1
                  Returning from N=1  with ans=1
                Returning from N=2  with ans=2
              Returning from N=3  with ans=6
            Returning from N=4  with ans=24
          Returning from N=5  with ans=120

5! = 120
```

Notice that the factorial method is initially called with N=5, and before it completes it calls `factorial` again with N=4, which calls factorial again with N=3 … and so on, until the `factorial` is called with the base case (N=1). This instance of the method completes its execution (does not call factorial again) and returns a value (1), which is passed back to the previous instance (N=2) of the method which can now complete and return a value (2) to its previous instance (N=3) ... and so on until the original method call (N=5) completes and returns a result (120) to the main method which displays the result.

As another example, consider the problem of calculating Fibonacci numbers. These are also defined by a recurrence relation:

$fib_1 = 1$

$fib_2 = 1$

$fib_N = fib_{N-1} + fib_{N-2}$ for $N>2$

This recurrence relation defines 2 base cases (N=1,2) with a simple solution, and expresses all other cases as a combination of 2 simpler cases of the original problem.

A recursive method to calculate the N^{th} fibonacci number is:

```
long fibonacci(int N)
{
    if (N==1 || N==2)     // the base cases
    {
        return 1;
    }
    else                  // recursive step
    {
        return fibonacci(N-1)+fibonacci(N-2);
    }
}
```

If we call this method in a program that inputs a value for N and calls fibonacci(N) to calculate the N^{th} number, execution will proceed as follows: *(additional output statements were included at the beginning and end of the recursion method to trace the execution)*

```
Calculating Fibonacci Numbers
-----------------------------

Enter fibonacci number required : 5

          Entering  with N=5
            Entering  with N=4
              Entering  with N=3
                Entering  with N=2
                Returning from N=2  with ans=1
                  Entering  with N=1
                  Returning from N=1  with ans=1
              Returning from N=3  with ans=2
                Entering  with N=2
                Returning from N=2  with ans=1
            Returning from N=4  with ans=3
              Entering  with N=3
                Entering  with N=2
                Returning from N=2  with ans=1
                  Entering  with N=1
                  Returning from N=1  with ans=1
              Returning from N=3  with ans=2
          Returning from N=5  with ans=5

5th Fibonacci number is 5
```

The `fibonacci` method is called with N=5, and before it completes it calls `fibonacci` again with N=4, … and so on until a base case (N=2) and this instance of the method completes it execution and returns a value (1), which is passed back to the previous instance of the method (N=3), which still needs $fibo_1$ to complete so it calls fibonacci again with N=1, a base case and returns a result to the previous instance (N=3) which can now complete and return a result to the previous call (N=4) which still needs $fibo_2$ so calls the method again … and so on until finally all the method calls are complete and a result is returned to the main method which displays the result.

Notice that in calculating this 5^{th} Fibonacci number a total of 9 recursive calls were made. This recursive solution is especially inefficient because 2 recursive method calls are required for each Fibonacci number in each recursive step. In fact, calculating the 10^{th} Fibonacci number (55) requires over 100 calls, the 15^{th} (610) requires over 1200 calls, and the 20^{th} (6765) requires over 13500 calls!

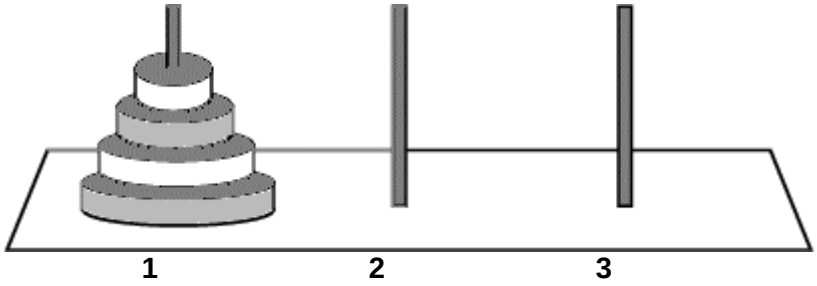
In fact, many recursive problems can be expressed iteratively. Iteration uses a repetition structure (for/while) and recursion uses a selection structure (if). However, they both involve repetition – iteration explicitly, recursion through repeated method calls. Both use a termination test – iteration when the loop continuation condition fails, recursion when a base case is recognised. Both can occur infinitely – iteration if the loop-continuation test never becomes false, recursion if the problem is not reduced each step in a manner that converges on the base case.

The reason recursion is so inefficient is because it repeatedly invokes the mechanism and hence the overhead of the method calls. This can be expensive both in terms of processor time and memory space. Each recursive call causes a copy of the method variables to be created. Iteration normally occurs within a method so there is no overhead of repeated method calls and extra memory assignment.

So why choose recursion?
A recursive approach is generally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem, and results in a program that is easier to understand and debug. Another reason is that the iterative solution may not be apparent.

As an example of a case where recursion is required because an non-recursive algorithm is extremely complex, consider the *Towers of Hanoi* problem.

> *Legend has it that in a temple in the Far East (Hanoi?), is a gold rod with 64 golden discs stacked on it, arranged in order of size so that each disc rests on a larger disc. Monks are attempting to move all the discs from this rod to another one (using a third in the process), by only moving 1 disc at a time and ensuring that each disc always rests on a larger disc. When they complete their task, the world would end!*



Consider the problem for N discs.

When N=1, the solution is trivial: simply move the 1 disc to the other rod.

When N=2, the solution is manageable:
Move the smaller disc out of the way and onto the third rod,
Move the larger one to its destination
And replace the smaller on top of it.

When N=3, things start getting more complex: (calling the rods 1,2,3 and assuming we wish to move the stack of discs from rod 1 to rod 2)
Move the smallest disc onto rod 2,
Move the middle disc into rod 3,
And replace the smaller one on top of it,
Move the largest disc onto rod 2,
Move the smallest disc from rod 3 out of the way onto rod 1,
Move the middle disc on top of the largest one on rod 2,
Place the smallest disc on top of the stack on rod 2.

To devise an algorithm in this manner for the larger cases is extremely difficult. However, if we consider a recursive approach, the solution is almost trivial.

We have shown we can move 1 disc to a specified destination (the base case). To move N discs to a specified rod, we simply move N-1 discs out of the way to the spare rod, then 1 disc (the largest) to the specified rod, and then move the N-1 discs from the spare rod onto the specified rod.

Following this approach (by hand) to move 3 discs from rod 1 to rod 2:

move 3-1=2 discs from rod 1 to rod 3;
move 2-1=1 disc from rod 1 to rod 2;
move 1 disc from rod 1 to rod 3;
move 2-1=1 disc from rod 2 to rod 3;
move 1 disc from rod 1 to rod 2;
move 3-1=2 discs from rod 3 to rod 2;
move 2-1=1 disc from rod 3 to rod 1;
move 1 disc from rod 3 to rod 2;
move 2-1=1 disc from rod 1 to rod 2;

which is the same as the algorithm devised above.

The program is as simple and involves outputting a set of instructions. No value is returned by the method – the recursive calls are merely to output the steps in the correct order. We number the rods 1,2 and 3 as above, which has the nice feature that if you know the numbers of the origin and destination rods, the spare rod can be found by 6-(origin+destination).

```
Hanoi(int N,int from,int to)
// ------------------------------------
// recursive method to move disks
{
   int spare = 6-(from+to);
   if (N==1)    // the base case
   {
        System.out.println("  move disc from "+from+" to "+to);
   }
   else                  // recursive step
   {
        Hanoi(N-1,from,spare);
        Hanoi(1,from,to);
        // or more simply saving 1 call
        //System.out.println("  move disc from "+from+" to "+to);

        Hanoi(N-1,spare,to);
   }
}
```

recursive method from `TowersHanoi.java`

For N discs, 10^N –1 moves are needed. This means that for 64 discs, if the monks move 1 disc a minute, it will take over 3.5×10^{13} (35 million million) years for the world to end – even if we give them the solution!.

Exercises

For each of these exercises, write programs that test your recursive methods

1. Write a recursive method
   ```
   double Power(double x, double y)
   ```
   That calculates x^y

2. Write a recursive method
   ```
   void Backward(int[] A)
   ```
   To display the contents of an integer array A in reverse order.

3. Write a recursive method
   ```
   boolean isAPalindrome(String myPhrase)
   ```
   that returns true if myPhrase reads the same forwards and backwards, and returns false otherwise

4. Write a recursive Binary Search method
   ```
   int Binary (int key, int[] A, int first, int last)
   ```
   that returns the array index of the key in the array A, or –1 if it is not found. The parameters first and last define the beginning and end index of the sublist under consideration.

5. The highest common factor (HCF) of two positive integers is defined as

   HCF(M,N) = HCF(N,M) if M<N
   = M if N=0
   = HCF(N,M mod N) otherwise

   where M mod N means the remainder after dividing M by N (i.e. M % N in Java)

   Write a recursive method
   ```
   int HCF(int M, int N)
   ```
   to calculate the highest common factor of the two positive integers M and N.

   Variation:
   The iterative algorithm is: *repeatedly subtract the smaller from the larger number until they become equal* – this value is the HCF. Write an iterative HCF method as well.

15. Useful Data Structures

We've so far considered two structured data types – arrays, which are used for storing sequences of data items all of the same type, and strings, used for storing sequences of characters. The problems we need to solve don't always fit into these simplistic constraints, and it is useful to consider a number of variations on these basic structures which may be suitable under different circumstances.

15.1 Inner classes

Arrays are used to store data items all of the same type, for example, all elements of

```
int[][] myArr = new int[4][5];
```

are integers.

This is limiting as at times we wish to store some information about a particular entity (eg a person, or a book, or a can of paint), and the information is not all of the same type. Typical attributes might be

person:	name (String), ID number (long), date of birth (date object, or 3 integers), annual salary (double) ….
book:	title (String), author (String), ISBN number (long), year published (int) ….
paint:	product code (int or String), colour code (int or String), description (String), volume in litres (int), selling price (double), number of cans in stock (int) ….

If we are designing a class to deal with a single instance of such an object we would define attributes of different types and write a constructor, `Get…` and `Set…` methods and any other methods required to provide the object with functionality.

However, consider the case where we do not require the individual objects to have functionality, but rather merely wish to use them to store data, and in particular wish to deal with many such objects, for example, all the persons employed by a company, or all the books in a library, or all the cans of paint in a shop. In that case the class we're designing would be an Employer class, or a Library class, or a Paintshop class and an attribute of such a class would be an array of persons or books or paint.

In order to define an array of such objects we need to define a class from which to instantiate the objects to form the elements of the array. Such a class only needs to have context and meaning within its parent class – for example, a book within the Library, or a can of paint within the Paintshop – and so is not defined as a general purpose, stand-alone class with full object functionality, but rather as an **inner class**.

An inner class is a class defined within another class (called its *outer class*), and which has meaning and scope only within that enclosing class. It's purpose is to support the work of the containing outer class. In this context an inner class can be thought of as a "user-defined data type" in that we are defining our own structure with component elements (fields) of any type which together represent some entity. The outer class can access the attributes of its inner class directly.

For example, consider a class representing a university course which needs to store information about the students registered for that course such as student name and number, class mark, exam mark and final mark. An inner class can be defined to represent a student with suitable attributes, and the course class can have an array of such student objects as attribute.

```
// A course class consisting of a number of student objects
// Illustrates using an inner class
//---------------------------------------------------------
import java.io.*;
import Utilities.*;

public class Course
/*******************/
// The Course class
{
      private class Student
      //====================
      // inner class to represent a student
      {
        private long stNum;             // student number
        private String stName;          // student name
        private double examMark=0;      // stored
        private double classMark=0;     // stored
        private double finalMark=0;     // calc from class&exam

      } // end of Student class
      //=======================

// attributes of class Course
//---------------------------------------------------
  private String courseName;         // module name
  private int nSt;                   // num students registered
  private Student [] students;       // array of Student objects
  private int nPass;                 // num students who passed
//---------------------------------------------------

 Course(String name, int numSt, String fname) throws IOException
//---------------------------------------------------
// constructor - creates a course object with name,
// number of students and array of registered Students,
{
  courseName = name;
  nSt = numSt;
  students = new Student[nSt];

  BufferedReader fin = GenIO.open(filename);

  for (int i=0;i<nSt;i++)       // sets up the array of students
  {
    // instantiate each student object
      students[i] = new Student();
    // read attrib values from file and calc final amark
      students[I].stNum = GenIO.getLong(fin);
      students[I].examMark = GenIO.getDouble(fin);
      students[I].classMark = GenIO.getDouble(fin);
      students[I].stName = GenIO.readStrToEol(fin);
      students[I].finalMark =
        (students[I].examMark*2 + students[I].classMark)/3.0;
  }
 }
```

```
public void SortStudents()
//--------------------------
// sort Student array into ascending order on student number
  {
    int pass = 0;
    boolean swapped;

    do
    {
      pass++;
      swapped = false;
      for (int st=0;st<(nSt-pass);st++)
      {
        if (students[st].stNum>students[st+1].stNum)
        {
          // exchange references. – no need to copy data values
          Student temp = students[st];
          students[st] = students[st+1];
          students[st+1] = temp;
          swapped = true;
        }
      }
    } while (swapped);          // if no swops made => sorted
  }

  public void Display()
//---------------------
// Displays the student data
  {
    System.out.println(courseName + "\n---------");

    for (int st=0;st<nSt;st++)
    {
      System.out.println(GenIO.format(students[st].stNum,9)+" "
        + GenIO.alignL(students[st].stName,20)
        + GenIO.format(students[st].examMark,6,1)
        + GenIO.format(students[st].classMark,6,1)
        + GenIO.format(students[st].finalMark,6,1));
    }
  }
```

Course.java

Note that the attributes of class `Student` can be accessed in class `Course` even though they are declared private since `Student` is defined inside `Course` and is hence within its scope. The individual attributes are referenced using the particular object name (eg `students[i]` for that particular object in the array) together with the attribute required as in

```
students[i].finalMark =
    (students[i].examMark*2 + students[i].classMark)/3.0;
```

In other words, since this is an internal class it is not necessary to provide `Get..` and `Set..` methods to access the attribute values from the methods of outer class.

When creating an array of objects there are 3 steps to follow to instantiate the array correctly:

1. declare an array of the appropriate object type
```
private Student [] students;
```
2. instantiate this array to hold the required number of elements (objects)
```
students = new Student[nSt];
```

or 1 and 2 can be combined as in
```
private Student [] students = new Student[nSt];
```

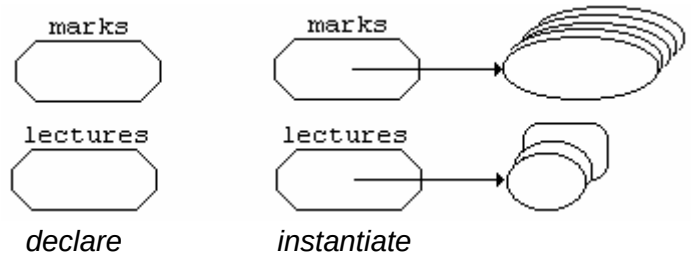
3. instantiate each object in the array
```
for (int i=0;i<nSt;i++)
{
  students[i] = new Student();
  ...
}
```

Recall the process that occurs when an array (or any object) is declared.

- declare: `int [] marks;`
- instantiate: `marks = new int[5];`

or

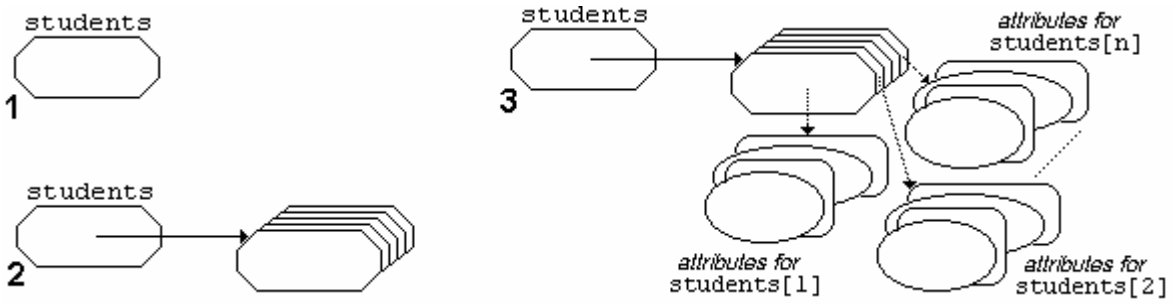
- declare: `Time lecture;`
- instantiate: `lecture = new Time();` *(assuming no constructor)*



The declaration defines the object name and allocates a variable to hold the address of the object; the instantiation reserves the memory needed to hold the array elements or object attributes and stores the address of the start of this memory block in the object variable.

In the 3 step declaration of an array of objects,

1. the first declaration defines the name of the array of objects and allocates a variable to hold its address;
2. the instantiation of this array serves as the declaration of the n objects in the array and reserves the n units of storage needed to hold the memory address of each object;
3. the instantiation of each of the n objects in the array reserves space for that object's attributes and puts their start address in the array element for that object.



The inner class discussed above provides a means of grouping items of dissimilar data types into a single object, but no methods are provided to give functionality to the class – it merely represents a data structure.

Another approach is to include some limited functionality to augment the basic data representation of the inner class. This does not normally consist of a full range of methods of different types providing a variety of behaviours – such functionality is usually reserved for independent classes (discussed below) – but typically the class is provided with one or more constructors to allow the object to be instantiated with specific values and not just defaults, and a `toString` method to allow for default display of the object's attributes values.

The inner class `Student` of the `Course` class discussed above has been modified to illustrate this approach of incorporating some limited functionality into the inner class.

```
  private class Student
  //====================
  // inner class to represent a student
  {
    private  long stNum;             // student number
    private String stName;           // student name
    private double examMark=0;       // stored
    private double classMark=0;      // stored
    private double finalMark=0;      // calc from class&exam
  //-----------------------------------------

  Student(long num,String name)
//-----------------------------
// constructor - creates a student object with number and name
  {
    stNum = num;
    stName = name;
  }

  public String toString()
//-------------------------------------------------------------
// returns a String containing all the details for this student.
// GenIO's formatting methods all return strings, and are used
// here to convert the numeric data to strings.
  {
    String outString = "";

    outString = GenIO.format(stNum,9) + " "
      + GenIO.alignL(stName,20)
      + GenIO.format(examMark,6,1)
      + GenIO.format(classMark,6,1)
      + GenIO.format(finalMark,6,1);

    return outString;
  }
} // end of Student class
//=======================
```

inner class `Student` *in* `Course1.java`

15.2 Arrays of independent objects

In some cases the object being modelled is an independent entity in its own right, and is more correctly represented as a public class with attributes and methods. Certainly a case can be made for designing a fully independent `Student` class to model a student, instead of merely the data values and perhaps limited functionality of the inner classes used above.

Such a `Student` class may have a number of attributes, for example, student number, name, marks for tests and assignments, class mark, exam mark and final mark and grade, and would require methods to initialise it with a name and student number, to enter marks, to calculate the class mark and the final mark and grade, and to make each of these values available to other users. Obviously, the complexity and level of detail depends on the use we're going to make of the `Student` class. And analogous to the inner classes above, in a class `Course` we may chose to have as attribute an array of `Student` objects, each with its own attributes and defined functionality.

Apart from the increased functionality, the other significant difference between this fully independent, public `Student` class and an inner class with limited functionality is that the public class is defined in its own file and so does not fall within the scope of the `Course` class. This means that `Student`'s attribute values cannot be accessed directly (as can the attributes of an inner class) but that explicit methods (Gets and Sets) must be provided for the `Student` class to make its attribute values available to `Course` and to other classes that may wish to use `Student`.

To illustrate this, let us consider a specific example:
All modules offered by the Computer Science Department have a maximum of 5 assignments and 3 tests. These all contribute to the class mark, with the assignments counting equally towards 50% of the class mark, and the tests counting equally towards the other 50% of the class mark. The class mark counts 30% of the final exam, and in order to pass a student must get 40% or more for each of the class mark and the exam mark, and have a final mark of over 50%. Letter grades are awarded as

A :	$\geq$ 80%	D :	50-59%
B :	70-79%	S :	40-49%, or failed sub-minimum
C :	60-69%	F :	<40%

A student data system is required that will

- store student number, name and marks for all students in each module
- allow a mark to be entered for all students for a particular assignment/test or the exam
- calculate their class and final marks, and letter grades
- determine average marks for the class
- determine the pass rate
- output class lists (with all the marks) in alphabetic order, student number order, and descending order of final mark
- look up and display a specific mark for a given student (using student number)
- modify any of the test/assignment marks for a given student
- store and retrieve the data from file

Considering the objects required:

We need an class to represent a student. As a minimum it needs attributes to store student number and name, an array of assignment marks, an array of test marks, and the exam mark. Do we need attributes to store classmark and final mark and grade? It depends largely on the use which is made of the class. If we're only going to use it to calculate and display the final result, then we don't need to store them as part of the class; on the other hand if it is going to be used to record the calculated marks and access them in order to output them in different ways and display them and possibly modify them, then they must be stored as attributes and form part of the students record.

The methods required are methods to calculate the class mark; to calculate the final mark and grade; to store a mark; and to retrieve a mark. We can use individual methods to store each of the marks, eg `storeAss1(mark), storeAss2(mark), storeTest1(mark)`... etc; or we could write methods to store each type of mark, eg `storeAss(n,mark), storeTest(n,mark), storeExam(mark)`; or we could write a generic method to specify which mark to store by a reference number, say 1-5 for assignments, 11-13 for tests, and 21 for the exam (or something similar), as in `storeMark(which,mark)`.The first approach is unnecessary duplication, the second is probably makes the most logical sense, but is essentially a replication of the same method, while the third approach requires remembering a code to reference which mark to store but only writing one method. I'll use this third approach, mainly because I'd prefer to write 1 method rather than 3, even though the combined method requires testing to determine exactly which mark is needed. Similar arguments apply to retrieving a mark with the added requirement for retrieving the class and final marks (codes 22 and 23?). Another method is needed to return the letter grade. In addition, methods are needed to return the student number and name – eg for display purposes.

What about the constructor? – we'll assume it merely creates the object and stores the student number and name, and all marks are stored using the `storeMark` method. This is logical because it is unlikely that all the data is known when the object is created, but marks will need to be entered when the test/assignment has been marked.

A `toString` method is also useful. This method will return the object as a string and can then be used in writing the object to a file, or displaying it to the `System.out` device, or whatever.

Student
| stNum |
| stName |
| assigns [5] |
| tests[3] |
| examMark |
| classMark |
| final Mark |
| grade | - char
calcClMark()
calcResult()
storeMark(which,mark)
getMark(which)
getGrade()
getStnum()
getName()
toString()

The module is also represented as a class. It needs attributes to store up to 100 student objects (an array of objects), the number of students registered for the module, the number of assignments and tests given, various average marks, and the number of passes. Apart from a constructor to define the module parameters, we need a method to set up the students registered for the module which we'll get from a specified file. Methods are also needed to display the student records in a particular order, to enter marks for all students for a particular test, assignment or exam, to update a particular mark for a student and to return a given student's record as a String.

In addition, a method is needed to calculate the averages and the pass rate. One approach is to call this method automatically whenever a mark is changed, because the averages may now be out of date. This could be extremely wasteful if a number of changes are being made, because the entire array will be scanned and the averages recalculated after each individual change. Alternatively, the responsibility could be placed on the user to recalculate the averages after a batch of changes – more efficient but places the onus on the user.

Because the module object and its data will need to be used over a period of time, a method could also be written to write the module object to a data file, and an alternate constructor provided that will restore a previously created module from file. The filename will be the parameter for the constructor instead of the module setup data, and all the attributes will be retrieved from the file. (Left as exercise for the reader)

Module
| modName | | nSt | | students[] | - Students | nTests | | nAss | | avExam | | avClass | | avFinal | | nPass |
setupModule(filename) enterMarks(which) change1Mark(st,which,mark) calcAvs() displayAvs(to) getNumPass(); displayAlpha(to) displayNumeric(to) displayMarkOrder(to) getStudent(st) – returns String

A number of points need to be considered:

- should we hold the student objects in the array in any specific order – ie one of the output orders such as student number, alphabetic, or mark order? or just in random order?

If they are in random order we will need to do a linear, unordered search of the entire array each time a student record needs to be accessed – inefficient.
It is very difficult to hold them in mark order because it may change dynamically, and is even unknown at early stages – rather sort when required.
Alphabetic order is OK, but if we are using student number to locate a student eg to update a mark, then alphabetic order will once again require an unordered search. If a key is being used to look up data, it should be ordered on that key. In this case the specified key is student number (guaranteed to be unique, which name might not be), so the correct ordering is student number order.

- what do we do if we want to update a students mark, or want info on a particular student, and the student doesn't exist?

Some sort of error or signal must be given if a record cannot be found, and the class / user program must check for it after each access and react to it. Any method that needs to locate a particular student should use a single private method `findSt` that will return the array index for that particular student or –1 if the student is not found. Then the method to change marks or display the student number or whatever can watch for the –1 and react to it.

- how do we ensure that the data is consistent – for example, that the class marks and averages are always up to date, even if particular students marks are changed?

By ensuring that any method that changes a mark always invokes the relevant methods to update averages etc – anything that can be affected by a changed mark – before exiting. In general the marks will be entered as a batch, so the `enterMarks` modules must recalculate that student's average and the module averages. So instead of a single method we'll have 2 private methods, one to calculate the module averages that can be used when a single student's mark and hence the average has been changed (`calcModAv`), and one to calculate class and final marks for all students which also calls the `calcModAvs` method (`calcAvs`).

- are there any special considerations for sorting an array of objects rather than an array of (eg) numbers?

No – apart from the obvious one of ensuring that an appropriate form of comparison of objects is used. In this case that does not apply, because we are sorting on student name (String) for which appropriate comparison methods are supplied, or student number (long) or marks (double) both of which are simple logical comparisons. However, one consideration when sorting a large array of objects with many data items, is whether we should physically exchange the data items during a sort, or instead just sort a reference to each item to avoid the overhead of copying all the data items many times – particularly when we're just sorting in order to display the data and do not wish to make permanent changes. More about this later.

The classes are all saved in the `ArrayObjDemo` folder, and should be carefully studied to ensure you understand what the different methods do, how they do it, and how they interrelate. Only extracts are shown here, together with some discussion.

Consider first the `Student` class.

Its attributes are as discussed, and the constructor merely stores the student number and name in the relevant attributes. All entry of marks is done by specific methods.

Method `storeMark` is straightforward – it checks which mark has to be stored and replace that mark in the students attributes by the parameter value. If an illegal mark code is used and error message is sent to the System error device, but as this method should only be used by the `Module` class which only invokes it with valid values this is not really requires. It is included because it is a natural consequence of checking for which mark to store, and in case this method is ever invoked by another user.

```
public class Student
/*******************/
// The Student class
{
// attributes
//-----------------------------------------------
  private  long stNum;           // student number
  private String stName;         // student name
  private double [] assigns = new double[5];
                              // max 5 assignments
  private double [] tests = new double[3];
                              // max 3 tests
  private double examMark=0;     // stored
  private double classMark=0;    // calculated from assigns+tests
  private double finalMark=0;    // calculated from class+exam
  private char grade=' ';        // letter grade - calculated
//-----------------------------------------------

  Student(long num,String name)
//-----------------------------
// constructor - creates a student object with number and name
  {
    stNum = num;
    stName = name;
  }
```

Method `calcClMark` is also routine – averages the assignment and test marks and combines them in equal proportions. The number of tests and assignments is specified as a prameter so that as long as the overall departmental guidelines on how to calculate the class mark are met, individual module differences in the numbers of assignments and tests can be accommodated.

Likewise `calcResult` will combine the class and exam marks according to prescribed ratios (30% : 70%), as long as both values are non-zero. This is to guard against calculating final marks before the components are available, and also caters for a student who may be absent for the exam. A sub-minimum is in effect – if either of the marks are <40 a maximum final mark of 48% can be attained, and the method checks for this. Then the letter grade is determined using a cascading `if` to award the appropriate grade for the mark group.

Four "get" methods are present – to return the student number, the student name, the grade, and any of the marks. The `getMarks` method is the only interesting one in that the mark code must be specified in order to indicate which mark to return, and this is checked.

The `toString` method returns a string consisting of all the attribute values for a student neatly formatted. `GenIO`'s methods all return strings and are used to build up the string. The advantage of a `toString` method instead of a display method is that the string can be used in any appropriate context – eg. sent to file, or manipulated programmatically – instead of merely being sent to the standard output device.

The `Module` class is more complex. The constructor merely sets up the module's parameters – name, number of students, tests and assignments - and creates the array to hold all the student marks. This is an array of `Student` objects, and each student is represented as an element of the array, ie. a `Student` object.

```
public class Module
/*******************/
// The Module class
{
// attributes
//-------------------------------------------------
   private String modName;           // module name
   private int nSt;                  // number of students registered
   private Student [] students;      // array of Student objects
   private int nAss,nTests;          // number of assignments and tests
   private double avExam,avClass,avFinal;
                                  // average marks for this module
   private int nPass;                // number of students who passed
//-------------------------------------------------

  Module(String name, int numSt, int numA, int numT)
//-------------------------------------------------
// Constructor
// Creates a module object with name, array of registered Students,
// and numbers of tests and assignments in this module
   {
      modName = name;
      nSt = numSt;
      students = new Student[nSt];
      nTests = numT;
      nAss = numA;
   }
```

Each `Student` object in the array has to be instantiated, and this is done in the method `setupStudents`, which sets up all the names and student numbers for each student in the module from a specified file, and then sorts the attribute array into ascending order of student number. Because each array element is (a reference to) an object, the sort needs merely to exchange the references and not actually copy the values. A bubble sort is used.

There are two alternate modules to enter a set of marks for all students for a specified assignment, test or the exam. One accepts data values from the keyboard, and the other inputs them from a file. (ie. method `enterMarks` is overloaded). Both methods return a boolean value to indicate whether or not the mark entry completed successfully, or whether errors were detected.

```
  public boolean enterMarks(int which)
//-------------------------------------
// allows a particular set of marks denoted by which to be
// entered from the keyboard for all students in the class.
// The method returns false if an error occurs.
_____________________________________________________________________

  public boolean enterMarks(String filename, int which)
                                          throws IOException
//-----------------------------------------------------
// alternate enterMarks method to allows a particular set of
// marks denoted by which to be entered from the specified file.
// The file must contain the student number and the mark in the
// correct student num order so that the numbers can be matched.
// The method returns false if an error occurs.
```

In both methods the mark code is checked to ensure a valid assignment or test or the exam has been specified, and if not the method terminates returning false. The keyboard entry version displays each students number and name in turn and inputs that students mark, while the file entry version inputs marks one by one from a file, and for each checks that the student number for the mark matches the next student in the array. If the file of marks and the array are not in exactly the same order the method returns false.

The method to calculate the student and module averages is a private method as it is automatically invoked each time a mark is entered or changed, and so the averages are always correct and there is never a need to specifically invoke the method to calculate the averages, they can merely be retrieved or displayed using the appropriate methods.

```
  private void calcAvs()
//----------------------
// Calculates the class mark and final mark and letter grade for
// each student, and call calcModAvs to calculate the modules
// averages and number of passes.
  {
  // do calcs for each student in this module
    for (int st=0;st<nSt;st++)
    {
      students[st].calcClMark(nAss,nTests);
      students[st].calcResult();
    }
    calcModAvs();
  }
```

The processing is done in 2 sections: `calcAvs` loops through all students and invokes each student's `calcClMark` method to calculate the class and final marks for that student, and then invokes `calcModAvs` to calculate the module averages for the class, exam and final marks. The reason these were separated is that when an individual student's mark is changed (`change1Mark`) we only need to call that student's `calcClMark` method and then recalculate the module averages, so `calcModAvs` is invoked in both cases.

```
  private void calcModAvs()
//-------------------------
// Calculates the average exam, class and final marks, and
// the number who passed.
  {
    double totExam = 0,
           totClass = 0,
           totFinal = 0;
    double finMark;
    int count = 0;

    // loop through each student in this module
    for (int st=0;st<nSt;st++)
    {              // total these values for the averages
      totExam  += students[st].getMark(21);
      totClass += students[st].getMark(22);
      finMark = students[st].getMark(23);
      totFinal += finMark;

      if (finMark>=50)          // count the number of passes
        count++;
    }

    avExam = totExam/nSt;       // calculate class averages
    avClass = totClass/nSt;
    avFinal = totFinal/nSt;
    nPass = count;
  }
```

A number of the module methods require that a student number be specified so that a particular mark for that student can be retrieved or changed, or the student's results can be displayed. All these methods use a private method, `findStudent` to locate the array element in which the student's data is stored using the student number as the key, and to return the array index or –1 if the student is not found in the array. It uses a linear search since the data is arranged in student number order (the key field).

Methods `get1Mark` and `change1Mark` are similar in that they both accept a student number (which is used by `findStudent` to locate the student in the array), and a mark code to indicate which mark is to be returned/changed, and `change1Mark` also accepts a new value for the mark. However `get1Mark` returns a double value, either the requested mark or -1 if an error occurs, while `change1Mark` returns a boolean value to indicate either a successful operation or an error.

```
  public double get1Mark(long stnum, int which)
//-----------------------------------------------
// locates the given student, and returns the specified mark if
// is the operation is successful, and -1 if an error occurs
// which can be because the student is not in the array, or an
// invalid mark code was specified.
  {
    int st = 0;   // array index for this student

  // check mark code - only return marks if valid code given
    if (!((which>=1&&which<=nAss)
          || (which>=11&&which<=10+nTests)
          || (which>=21&&which<=23)))
    {
    System.err.println("Error - invalid mark code " + which);
      return -1;
    }

    st = findStudent(stnum);
    if (st<0)
    {                      // student not found
      System.err.println("Error - no such student " + stnum);
      return -1;
    }
    else
    return students[st].getMark(which);
  }
```

Notice the calls to `findStudent` to determine the array index, the use of class `Student`'s `getMark` and `storeMark` methods to access the student's attribute values, and the error checking on the which code which differs for the 2 methods – codes of 22 and 23 (class and final marks) are valid if the marks are being returned for information, but are not valid if changes are to be made as these marks are calculated.

```
  public boolean change1Mark(long stnum, int which, double mark)
//-------------------------------------------------------------
// locates the given student, and sets the specified mark to the
// value given. returns true is the operation is successful, and
// false if an error occurs which can be because the student is
// not in the array, or an invalid mark code was specified.
  {
    int st = 0;   // array index for this student

  // check mark code - only enter marks if valid code given
    if (!((which>=1&&which<=nAss)
          || (which>=11&&which<=10+nTests)
          || which==21))
    {
    System.err.println("Error - invalid mark code " + which);
      return false;
    }

    st = findStudent(stnum);
    if (st<0)
    {                      // student not found
      System.err.println("Error - no such student " + stnum);
      return false;
    }

  // store mark in the correct field
    students[st].storeMark(which,mark);
  // assignment or test mark changed, recalculate class mark
    if (which<20)
      students[st].calcClMark(nAss,nTests);
  // in any case recalculate final marks and module avs
    students[st].calcResult();
    calcModAvs();

    return true;           // mark change successful
  }
```

Because this method changes a student's mark, that students final marks are recalculated as are the module averages.

There a number of output/display methods. Both methods `getStudent` and `displayAvs` return a String which can then be sent to any output stream, whereas methods `displayNumeric`, `displayAlpha` and `displayMarkOrder` require a PrintWriter stream to be specified as a parameter to which the data is sent.

Method `getStudent` specifies the student number for the student required, and `findStudent` is invoked to return the array index. If the student is not in the array and –1 is returned then an error string is generated, otherwise class `Student`'s `toString` method is used to return the required string.

```
  public String getStudent(long stnum)
//-------------------------------------
// returns a string containing the students marks, or a message
// if the student is not in the array
  {
    int st = 0;   // array index for this student

    st = findStudent(stnum);
    if (st<0)          // student not found
    {
      return("Student " + stnum + " not taking this module");
    }
    else
    {
      return students[st].toString();
    }
  }
```

Method `displayAvs` is straightforward and uses `GenIO` to construct a string consisting of the module averages (ie attribute values) for the exam, class and final marks, and also the module pass rate.

And since the attribute array of students is maintained in student number order, the `displayNumeric` method merely loops through all the students in the array and uses their `toString` method to display each student's details.

The methods to display the table of student details in alphabetic name order, or descending mark order appear more complex because the attribute array is not maintained in those orders so a sort is required before the data can be displayed. However the attribute array itself must not be sorted, so there are two possible approaches: copy the entire array of students plus all their details to another array, sort that into the required order and display it; or set a new array with each student's entry consisting of his array index in the original array and the required sort field. This simpler array is then sorted into the required order, and then each element in turn is accessed and the index field used to access the array element in the original array to retrieve its data, which is displayed. This approach to sorting arrays of objects is examined in more detail in the next section.

15.3 Sorting arrays of objects

This section examines various alternates in sorting arrays of objects, where the objects themselves (the array elements) may be quite large, complex structures. In such cases a large overhead may be encountered in copying and exchanging the data values and a more efficient approach is required. It is also a useful technique for when we do not wish to disturb the order of the original array but merely to (for example) display it in sorted order.

What is required is to create an inner class to use as a temporary class within the current class or even an individual method (such as the sort method), and supply it with a constructor that will instantiate it with a subset of the original array's data (usually the field(s) on which to sort) as well as the index of this element in the original array. Then this sub-class is sorted, and the (now ordered) array index values are used to access the original array to display (or whatever) the full object.

Eg.

attribute array in number order:

	st. num	*name*	*other fields…*
0	12345	Jane	……
1	23456	Anne	……
2	34567	Susan	……
:	:	:	:
N	98765	Mary	……

sort array (as set up):

index	*name*
0	Jane
1	Anne
2	Susan
:	:
n	Mary

sort array (sorted):

index	*name*
1	Anne
0	Jane
n	Mary
:	:
2	Susan

When displaying the original array the sort array index field would be used to display "*the original array element whose index is in position x in the sort array, where x=0..n"*

2	23456	Anne	……
1	12345	Jane	……
n	98765	Mary	……
:	:	:	
3	34567	Susan	……

Declaring an inner class follows the same process as declaring any class – define the attributes and write a constructor that will store the required values in the attributes. Then instantiate objects of this class to create the sort array.

The class `sortdata` used in displaying the student array in alphabetical order (class `Module`) is shown here:

```
  public void displayAlpha(PrintWriter to)
//---------------------------------------
// Displays the student data in alphabetic order to the device
// specified. A "sort array" is set up with an index number
// and key field (student name) and this is sorted, and then
// the index number is used to retrieve the correct element
// from the student array.
  {
    class sortdata
  //--------------
  // inner class used to store the required 2 data fields –
  // index and student name
    {
      int index;
      String name;

      sortdata (int i, String n)
    //-------------------------
    // constructor for class sortdata
    // stores the index and makes its own copy of the string
      {
        index = i;
        name = n.substring(0);
      }
    }
  //---------------------------------  end of sortdata class

  // create sort-array with index and name for all students
    sortdata sortArr [] = new sortdata[nSt];
    for (int st=0;st<nSt;st++)
    {
      sortArr[st] = new sortdata(st,students[st].getName());
    }
```

Then once the sort array has been sorted into the correct order (a selection sort is used in this example to minimise the number of exchanges), the index is used to access the array attribute to display its elements in the appropriate order:

```
  // loop through the sorted sort-array, and use the index to
  // reference the student array to display the values

    for (int i=0;i<nSt;i++)
    {
      int st = sortArr[i].index;
      to.println(students[st].toString());
    }
```

15.4 Merging data sets

Consider the scenario where we have two sets of data stored in two files or two arrays or a file and an array, and we wish to combine then into a single data set, or process them together to form a combined result. Examples of when this may be required are

- combining 2 lists (my friends, your friends) into a single list (our friends) in alphabetical order
- for a shop, merging the list of items in stock (and quantities) with the list of new items delivered to form an updated stock list.
- processing purchases on account against the list of account holders before sending out monthly statements.

These examples share the feature that some items may appear in one or other list only while some may appear in the both lists. An essential requirement of the data sets to be merged is that they are both arranged in order on the same key (both in alphabetical order, or ascending numeric order of account number …etc).

As an example consider the running club and the cycling club in the town of Nottoohilly that decide to merge to form the Nottoohilly Athletic Club. There is a list of runners and a list of cyclists that need to be combined to form a list of athletes. Although some members only run or only cycle, there are others who do both and so will appear on both the original lists.

Runners:
Anne, Bill, Colin, David, Eric, Paul, Percy, Simon
Cyclists:
Alec, Barry, Betty, Bill, Carol, Charles, Colin, David, Eric, Fred

In order to process the names one by one in sequence to create the new list, you first need to access the first member of each list. Then an iterative phase is entered wherein the names are compared and the “first” one (ie, alphabetically or on whatever key is being used to order the items) is processed (copied to the new list) and the next item is accessed from that data set to replace the one that has been processed. If both names are the same then it is processed and the next item is accessed from both lists to replace the ones that have been processed. At any stage there must be one item from each data set under consideration. This phase continues until all the items in one or other list have been dealt with. The remaining items (all from one data set) are in the correct order so can just be processed sequentially.

Using the names in the example above, we’d first consider

runner: Anne
cyclist: Alec

Anne is before Alec so would be transferred to the new list and the next runner considered:

runner: Bill athletes: Anne
cyclist: Alec

Alec is before Bill so would be transferred to the new list and the next cyclist considered:

runner: Bill athletes: Anne, Alec
cyclist: Barry

Barry is before Bill so would be transferred to the new list and the next cyclist considered:

runner: Bill athletes: Anne, Alec, Barry
cyclist: Betty

Betty is before Bill so would be transferred to the new list and the next cyclist considered:

runner: Bill athletes: Anne, Alec, Barry, Betty
cyclist: Bill

Bill is on both lists so would be transferred to the new list and the next runner and the next cyclist would now be considered:

runner:	Colin	athletes:	Anne, Alec, Barry, Betty, Bill
cyclist:	Carol		

and so on until the stage

runner:	Eric	athletes:	Anne, Alec, Barry, Betty, Bill, Carol, Charles,
cyclist:	Eric		Colin, David

was reached. Eric is on both lists so would be transferred to the new list and the next runner and the next cyclist would now be considered:

runner:	Paul	athletes:	Anne, Alec, Barry, Betty, Bill, Carol, Charles,
cyclist:	Fred		Colin, David, Eric

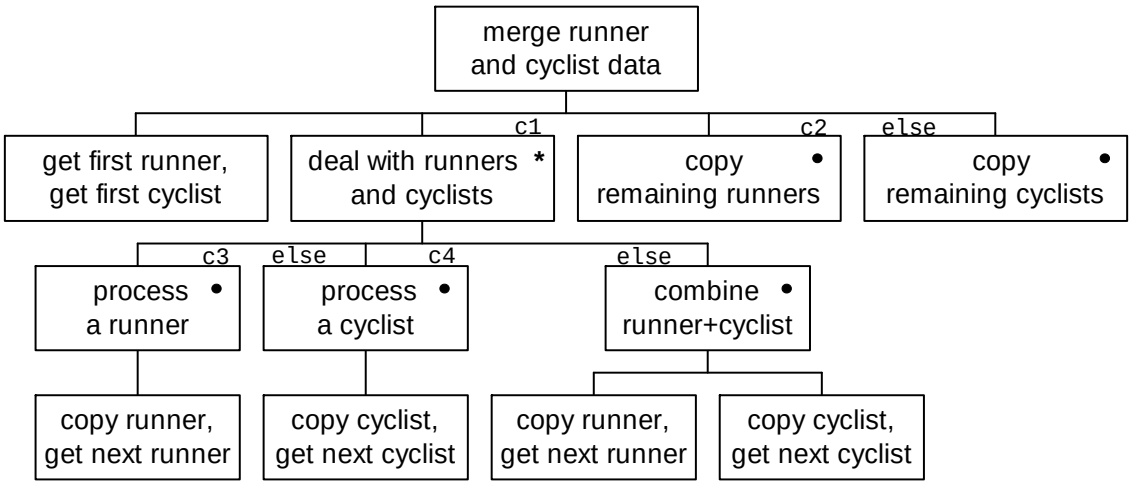
Fred is before Paul so would be transferred to the new list but the end of the cyclist list has now been reached so there is no cyclist to compare with runners.

runner:	Paul	athletes:	Anne, Alec, Barry, Betty, Bill, Carol, Charles,
cyclist:			Colin, David, Eric, Fred

At this stage all remaining runners could be transferred one after the other to the athlete list.

athletes:	Anne, Alec, Barry, Betty, Bill, Carol, Charles,
	Colin, David, Eric, Fred, Paul, Percy, Simon

The diagram below illustrates the processing required:




```
c1 : while still runner
     and cyclist data
```

```
c2 : if more runners
c3 : if runner before cyclist
c4 : if cyclist before runner
```

Aspects that need to be considered when writing a program to deal with a specific case are:

- how to access the next item in a data set. For an array it means updating the index, for a file you need to read the next record.
- how to determine when the end of a data set has been reached
- the processing required for each item
- how matching items are to be combined
- whether the items in each list are unique, or whether there could be more than one entry for a particular item (eg, 2 deliveries of the same product to a shop)

The file `MergeFiles.java` contains an example program to merge a file of runners and a file of cyclists to produce a file of athletes.

16. Simple Graphics

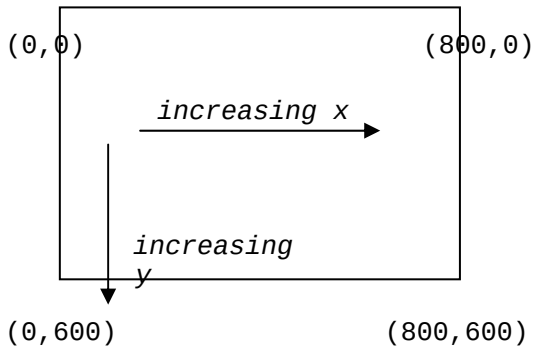
(based on Savitch: An Introduction to Computer Science and Programming)

16.1 Intro to AWT and Swing

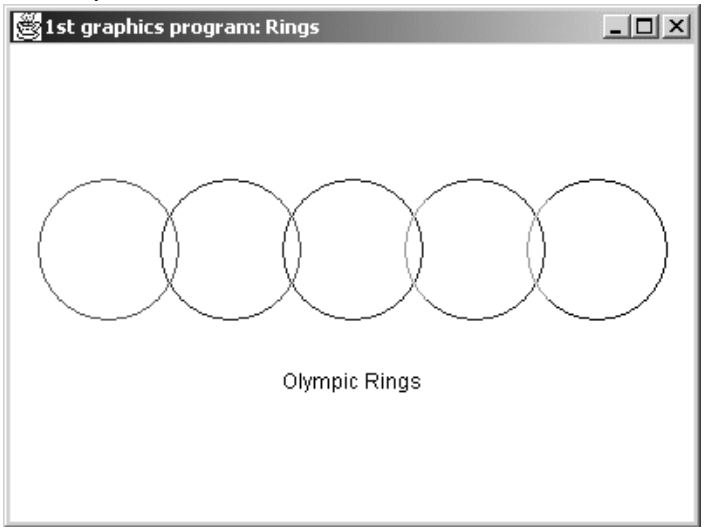
This is a brief introduction to using Swing to create simple windows. Swing is a standard library that allows programmers to develop Graphical User Interfaces (GUIs). Swing is based on an older library of classes (before Java 1.2) called the Abstract Windows Toolkit or AWT.

A window is a portion of the user's screen that serves as a smaller screen within the screen. It has a border defining its outside edges and a title, usually within the top border. There are many things that can be put inside a window when designing a GUI interface. We will cover how to include text and simple graphic shapes, how to use colour, and how to close the window. Then we'll consider how to include and use clickable buttons.

When using AWT, the size of an object on the screen is measured in pixels (picture elements). The actual size will depend on the screen resolution. In other words, a graphics screen is made up of a grid of tiny rectangles (pixels), each of which can have a colour. The more pixels on your screen, the greater the resolution. For example, a screen resolution of 800 x 600 means there are 800 pixels across the screen and 600 down the screen. The screen coordinate system has its origin in the top left corner of the screen, and the x-coordinate is measured from the left across the screen and the y-coordinate is measured from the top down the screen.



The easiest way to get going with graphics is to take a simple program and adapt it. Consider the following program that produces this window:



```
import java.awt.*;
import javax.swing.*;

class Rings extends JFrame
//------------------------
// draws Olympic rings
{
  public static void main (String[] args)
  //-----------------------
  // create a graphics frame of a given size, amd show
  // the drawing outlined in the paint method
  {
    Rings myWindow = new Rings();
    myWindow.setVisible(true);
  }

  public Rings()
  //------------
  // constructor - sets the window size & title
  {
    setTitle("1st graphics program: Rings");
    setSize (400,300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }

  public void paint (Graphics g)
  //----------------------------
  // draw 5 interlocking rings of different colours
  {
    g.setColor (Color.red);
    g.drawOval(20,100,80,80);
    g.setColor (Color.blue);
    g.drawOval(90,100,80,80);
    g.setColor (Color.green);
    g.drawOval(160,100,80,80);
    g.setColor (Color.yellow);
    g.drawOval(230,100,80,80);
    g.setColor (Color.black);
    g.drawOval(300,100,80,80);

    // label the drawing
    g.drawString("Olympic Rings",160,220);
  }
}
```

Rings.java

The first 2 statements says that the program uses the AWT library for drawing graphics and the newer Swing library for drawing windows. These should appear in any program using the AWT/Swing libraries.

The first line of the class definition says that this window is a `JFrame` (i.e. a derived class with base class `JFrame`). A `JFrame` is Swing's representation of a basic window, with a border, a place for a title and a close button.

This class has 3 methods:

- a constructor (`Rings`) which is called when a new `Rings` object (a `JFrame`) is constructed. In the constructor we specified some properties for this window i.e.the size and title of the window, and what should happen when we click on the close icon (for this window just exit the application);
- a `main` method, which creates a `Rings` object and ensures it is seen (the default is not to display the window);
- a `paint` method, which is called automatically by AWT and allows us to specify any graphical objects that must be displayed within the window.

The constructor normally includes statements that specify values for properties of the window (`JFrame`). In this example there are statements to specify the title to appear in the window's border and the size of the window (400 pixels across x 300 pixels down). The last statement sets the default action to take when the user clicks the close box. In this case the action is `EXIT_ON_CLOSE`. Some other options are `DO_NOTHING_ON_CLOSE` and `HIDE_ON_CLOSE`.

The `paint` method is called automatically and has a single parameter, a `Graphics` object, that allows the methods of the `Graphics` class to be used to display shapes, text and colours inside the window. Some useful `Graphics` methods are:

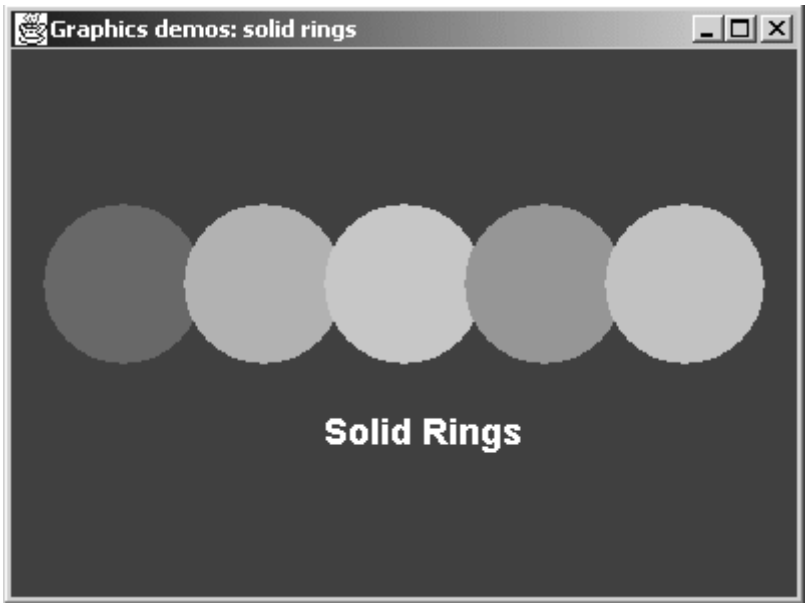
- `drawstring (String str, int x, int y)`
  Displays the string `str` starting at coordinate `(x,y)`
- `drawLine (int x1, int y1, int x2, int y2)`
  Draws a line from point `(x1,y1)` to `(x2,y2)`
- `drawRect (int x, int y, int w, int h)`
  Draws a rectangle with upper left corner at `(x,y)` and dimensions `w` x `h`
- `drawOval (int x, int y, int w, int h)`
  Draws an oval that fits within the rectangle that has its upper left corner at `(x,y)` and dimensions `w` x `ht`. To draw a circle, specify the same values for `w` and `h`.
- `fillRect / fillOval (int x, int y, int w, int h)`
  Draws the specified shape as above but fills it with the current colour
- `setColor (Color c)`
  Sets the current colour (for drawing) to `c`. This colour is used until a new default colour is set. `Color` constants are red, yellow, blue, orange, pink, cyan, magenta, black, white, gray, lightGray, darkGray. Or a user can "mix" a new colour scheme by constructing a new `Color` object and specifying the components. *(see `Rings3`)*
- `setFont (Font f)`
  Sets the current font (for text) to f. User must construct a font f by specifying the typeface, style and pointsize. *(see `Rings2`)*

Another example is

```
import java.awt.*;
import javax.swing.*;

class Rings2 extends JFrame
//-----------------------
// draws Olympic rings
{
  public static void main (String[] args)
  //-----------------------
  // create a graphics frame of a given size, amd show
  // the drawing outlined in the paint method
  {
    Rings2 myWindow = new Rings2();
    myWindow.setVisible(true);

  }

  public Rings2()
  //------------
  // constructor - sets the window title
  {
    setTitle("Graphics demos: solid rings");
    setSize (400,300);
    setBackground(Color.black);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }

  public void paint (Graphics g)
  //----------------------------
  // draw 5 interlocking rings of different colours
  {
    g.setColor (Color.magenta);
    g.fillOval(20,100,80,80);
    g.setColor (Color.cyan);
    g.fillOval(90,100,80,80);
    g.setColor (Color.pink);
    g.fillOval(160,100,80,80);
    g.setColor (Color.green);
    g.fillOval(230,100,80,80);
    g.setColor (Color.orange);
    g.fillOval(300,100,80,80);

    // label the drawing
    Font myFont = new Font ("Arial",Font.BOLD,18);
    g.setFont (myFont);
    g.setColor (Color.white);
    g.drawString("Solid Rings",160,220);
  }



}
```

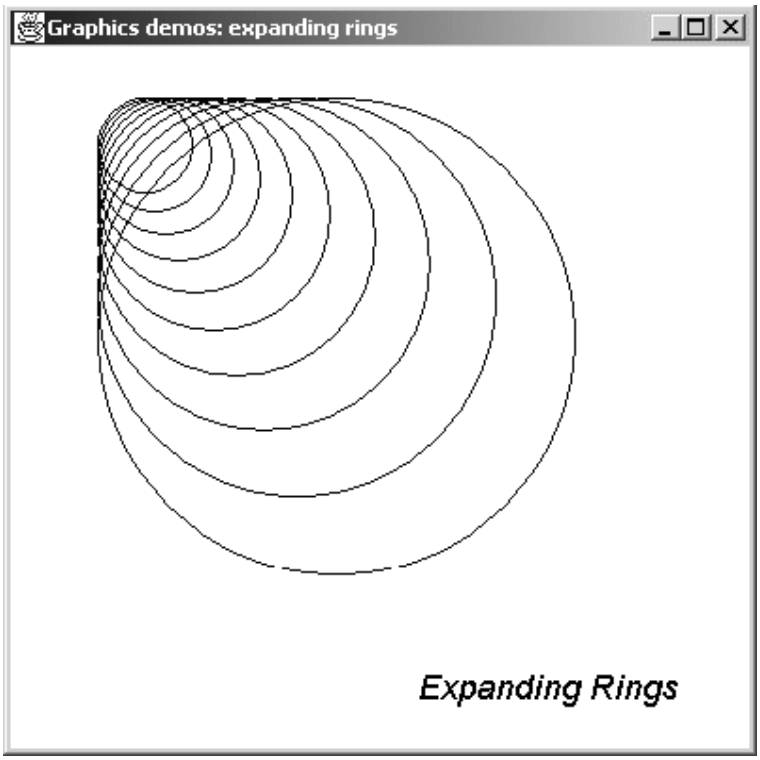
Rings2.java

This produces the output:



and shows how to set the window background colour, and how to set the font.

As a final example of a simple window, consider `Rings3.java` which creates a `Color` object with random red, green and blue values, and displays circles of different sizes calculated within a *for loop*. The output produced is



```
import java.awt.*;
import javax.swing.*;

class Rings3 extends JFrame
//-----------------------
// draws Olympic rings
{
  public static void main (String[] args)
  //-----------------------
  // create a graphics frame of a given size, amd show
  // the drawing outlined in the paint method
  {
    Rings3 myWindow = new Rings3();
    myWindow.setVisible(true);

  }

  public Rings3()
  //------------
  // constructor - sets the window title
  {
    setTitle("Graphics demos: expanding rings");
    setSize (400,400);
    setBackground(Color.lightGray);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }

  public void paint (Graphics g)
  //----------------------------
  // draw 5 expanding rings of different colours
  {
    int x=50,y=50,width=50,height=50;
    int red,green,blue;

    Color tint;

    for (int i=1;i<=10;i++)
    {
      red = (int) (Math.random()*256);
      green = (int) (Math.random()*256);
      blue = (int) (Math.random()*256);
      tint = new Color(red,green,blue);
      g.setColor (tint);
      g.drawOval(x,y,height,width);
      width = (int) (width*1.2);
      height = (int) (height*1.2);
    }

    // label the drawing
    Font myFont = new Font ("Arial",Font.ITALIC,18);
    g.setFont (myFont);
    g.setColor (Color.black);
    g.drawString("Expanding Rings",220,370);
  }


}
```

Rings3.java

16.2 Event-driven programming

Swing/AWT programs and virtually all other graphical user interfaces (GUI's) use events and event handlers. An **event** is an object that represents some action (e.g. clicking a mouse, pressing a key, choosing a menu option). When an object generates an event the event is **fired**. Every object that can fire events (e.g. a button that may be clicked on) can have one or more **listener** objects attached to it. These listener objects specify what will happen when events of various kinds are sent to the listeners. When an event is sent to a listener object the listener object invokes one of its methods. This method is called the **event handler**. As a programmer, you can define one or more listener objects for a graphical object and write event handlers to define what you want to happen when the object fires events.

Event driven programs are not executed by executing all the statements sequentially as defined in the main method. Instead the program sits in an infinite loop waiting for an event to be fired and then responds to it. Instead of writing a sequence of statements that call methods in a predetermined order, you create objects that can fire events and you create listener objects to react to those events.

When an event is fired it is automatically sent to the appropriate listener object for the object that fired the event. The listener object then calls the appropriate event-handling method to handle the event. You, as a programmer, do not explicitly invoke the event handling (and other) methods you write, it is done automatically by the Swing system.

In these programs we are defining classes that are derived classes of some basic predefined classes in the AWT/Swing library. All these classes inherit methods from their parent class. Some of these default methods work fine and some need to be customised to do whatever it is you need. For example, the `paint` method is called automatically to draw objects in your window. But the default `paint` method does not have any objects to draw. You need to write your own `paint` method to specify what it is you want to see on the screen.

In `Rings4.java` we create a listener object of type `WindowDestroyer` for the Window (`JFrame`). When the user clicks on the window's close box, a close window event is fired. This event is sent to all listener objects registered with this object, in this case just the `WindowDestroyer` object. Since it is a "windowClosing" event, the `WindowDestroyer` object invokes its `windowClosing` method, which is the event handler for the window closing event.

```
import java.awt.*;
import java.awt.event.*;

public class WindowDestroyer extends WindowAdapter
//-------------------------------------------------
// An object of this class will close a window if the
// close-window button is clicked. The object must be
// registered as a listener to any object of class Frame.
{
  public void windowClosing(WindowEvent e)
  {
    System.exit(0);
  }
}
```

WindowDestroyer.java

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class Rings extends JFrame
//------------------------
// draws Olympic rings
{
  public static void main (String[] args)
  //-----------------------
  // create a graphics frame of a given size, amd show
  // the drawing outlined in the paint method
  {
    Rings myWindow = new Rings();
    myWindow.setVisible(true);
  }

  public Rings()
  //------------
  // constructor - sets the window size & title
  {
    setTitle("1st graphics program: Rings");
    setSize (400,300);
    addWindowListener(new WindowDestroyer());
  }

  public void paint (Graphics g)
  //----------------------------
  // draw 5 interlocking rings of different colours
  {
    g.setColor (Color.red);
    g.drawOval(20,100,80,80);
    g.setColor (Color.blue);
    g.drawOval(90,100,80,80);
    g.setColor (Color.green);
    g.drawOval(160,100,80,80);
    g.setColor (Color.yellow);
    g.drawOval(230,100,80,80);
    g.setColor (Color.black);
    g.drawOval(300,100,80,80);

    // label the drawing
    g.drawString("Olympic Rings",160,220);
  }
}
```

Rings4.java

In order to use the event model we have to import the AWT event library in line 3. The last statement in the constructor creates and registers a listener that watches out for a mouse click on the close-window button. It creates a `WindowDestroyer` object from the class `WindowDestroyer` that must be included as a private inner class or defined externally as a public class (as in this example program). When the close window event is fired, the `windowClosing` method is invoked and the application terminates. We could have put additional processing or cleanup in the `windowClosing` method.

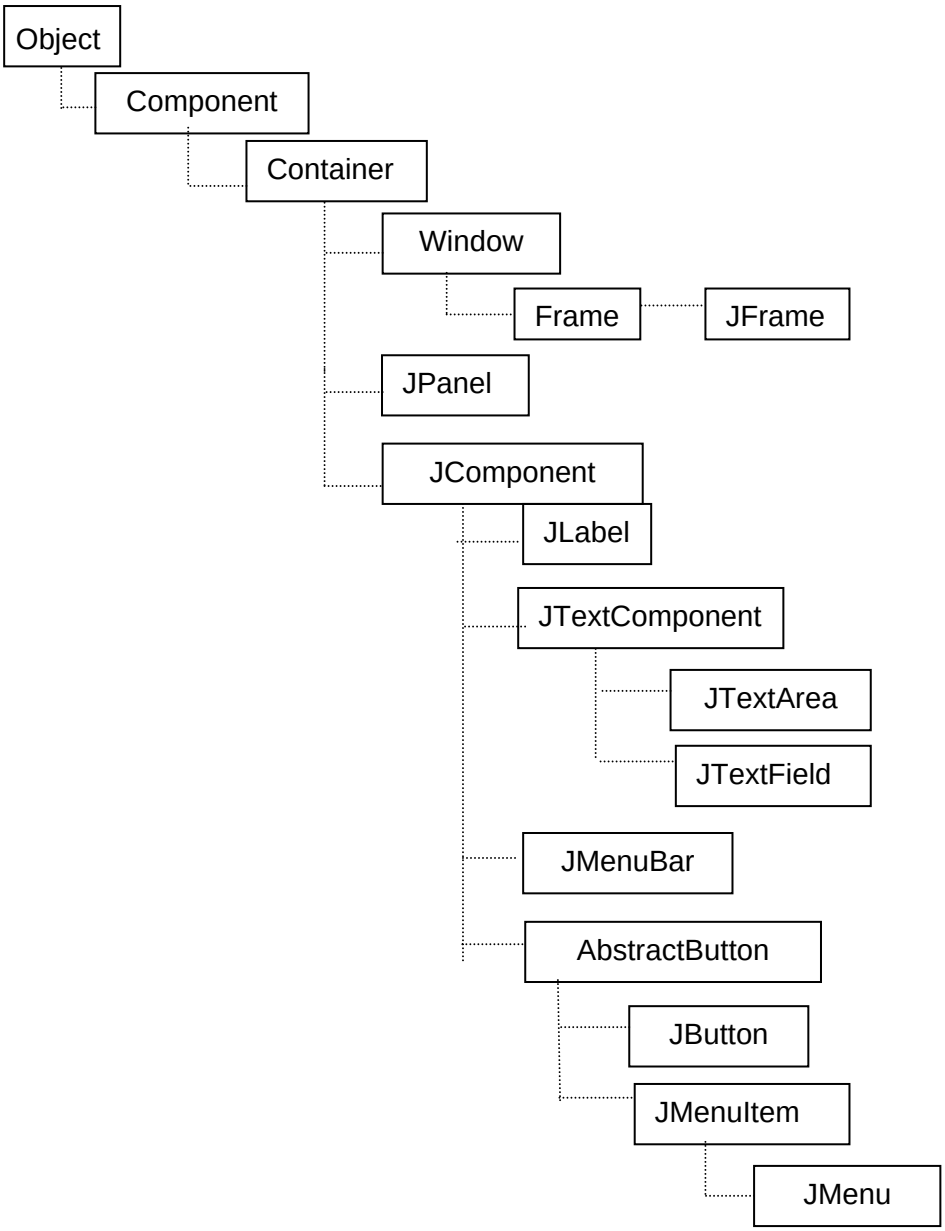
16.3 Components, containers and layout managers

Many of Swing/AWT's classes can be used as a container into which you place various other objects (such as other sub-windows, buttons, menus, ...). AWT has a predefined class called `Container`, and any descendant of `Container` (such as `JFrame`) can have objects added to it. That means that windows like those of the previous examples can have other objects added to them.

There are three kinds of objects you deal with when using `Container` classes:

- the **container class** itself (usually some sort of window). Every container class has a method called `add` which you use to add components to the class.
- the **components** you add to the container (eg buttons, menus, text fields)
- a **layout manager**, which is an object that positions components inside the container.

The hierarchy of objects is shown in this diagram. A line linking two objects means the lower class is derived from the higher class.



16.4 Buttons

The `JButton` class is descended from the `JComponent` class. An object of the `JButton` class is displayed as a clickable button with a label which is specified as a string when you construct a `JButton` object.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JButtonDemo extends JFrame implements ActionListener
{
   private String theMessage = " Watch this space!";

   public static void main (String[] args)
   //-----------------------
   // create a graphics frame of a given size and show its contents
   {
      JButtonDemo myWindow = new JButtonDemo();
      myWindow.setVisible(true);
   }

   public JButtonDemo()
   //----------------
   // constructor - sets the window size & title
   {
      setTitle("Button Demonstration");
      setSize (400,300);
      addWindowListener(new WindowDestroyer());

      // get the content pane from the window
      Container contentPane = getContentPane();

      contentPane.setBackground(Color.lightGray);
     // layout manager to arrange components
      contentPane.setLayout(new FlowLayout());

      Button redButton = new Button("Red");
      redButton.addActionListener(this);
      contentPane.add(redButton);

      Button greenButton = new Button("Green");
      greenButton.addActionListener(this);
      contentPane.add(greenButton);

      Button blueButton = new Button("Blue");
      blueButton.addActionListener(this);
      contentPane.add(blueButton);
   }

   public void paint (Graphics g)
   //----------------------------
   {
      super.paint(g);
     Font myFont = new Font ("Arial",Font.BOLD,18);
      g.setFont (myFont);
      g.setColor (Color.black);
      g.drawString(theMessage,120,220);
   }
        ..... more
```

part of JButtonDemo.java

The program `JButtonDemo.java` contains, a class `JButtonDemo` which is derived from `JFrame`, a constructor that sets the window size, title and background colour (and defines some buttons), and a `paint` method that displays some text using `drawString`. The string to be displayed is an instance variable `theMessage`. When a button is clicked, the `actionPerformed` method is invoked and the value of this variable is changed and `repaint` is automatically called to redraw the window. The method `repaint` takes care of some overheads and calls the method `paint` which we has been defined in this example. If you wish to **force** a refresh of a window you can call `repaint` (even though you haven't defined it), which in turn calls `paint` Note that you should never call `paint` directly rather call `repaint`.

A `JFrame` has a container called a `ContentPane`. All graphical objects, such as buttons to be displayed within the window, must be added to the `ContentPane`. To get the `contentPane` for this window:

```
Container contentPane = getContentPane();
```

To add buttons to the `contentPane` you must first create a button with a label that will be displayed on the button:

```
JButton redButton = new JButton("Red");
```

You then add an `ActionListener` to the button to register a listener to receive events from the button (e.g. to do something when the button is clicked)

```
redButton.addActionListener(this);
```

Finally you add the button to the container class by calling the add method.

```
contentPane.add(redButton);
```

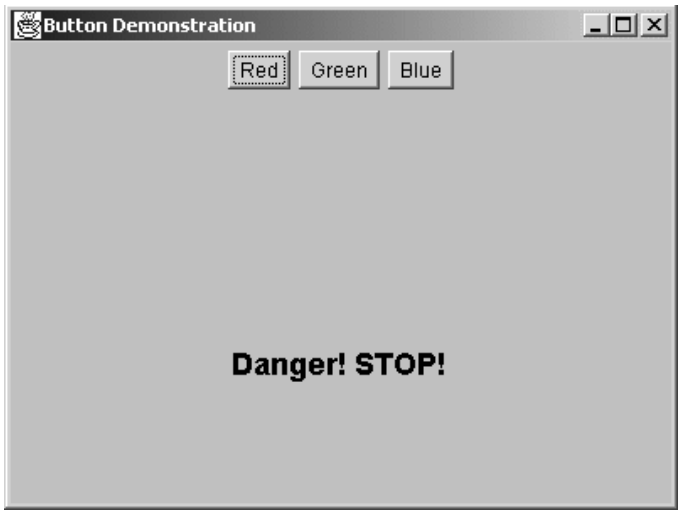
The method `add` simply throws buttons into the window. A layout manager controls the position in which the buttons or other graphical components are added. When you are adding components to a container you must create a layout manager object which arranges the components you add to a class.
The three basic layout managers are:

- `FlowLayout` – displays components left to right along the top row, then the second row… You can define whether they are centred (default) or left or right justified.
- `BorderLayout` – displays in 5 areas – north, south, east, west or center. The 2nd argument of `add` specifies which area the component goes into.
- `GridLayout` – displays them in a grid with each component stretched to fill the grid. The 2 arguments of the constructor define the number of rows and columns in the grid

In our example we create a new layout object in this case FlowLayout, and set this object to be the layout manager

```
contentPane.setLayout(new FlowLayout());
```

Thus the buttons are added along the top row from left to right (centred) in order.



A button fires events known as **action events**, and these are handled by listeners called **action listeners**. `ActionListener` is not a class, it is a property that can be given to any class by:

- Adding the phrase `implements ActionListener` to the class definition
- Defining a method called `ActionPerformed`.

The statement

```
class JButtonDemo extends Frame implements ActionListener
```

makes the window class `JButtonDemo` into the `ActionListener` that handles button events.

Recall that to register a listener for each button in the window we used the statement

```
RedButton.addActionListener(this);
```

This statement is inside the constructor for the window so it adds the action listener to "`this`" i.e. the actionlistener is the current object (itself).
This means the window will listen for actions on any of its buttons, and the action it takes will affect the window itself. You could have implemented the actionlistener as a separate class, but since we are changing the background colour and text in the this window, it makes more sense to make the `JButtonDemo` class the listener as well.

When an action listener receives an action event, the event is automatically passed to its `ActionPerformed` method. This method determines what type of action event was fired and branches accordingly to execute the required statements.

```
public void actionPerformed(ActionEvent e)
//---------------------------------------
// determine which button was clicked and take appropriate action
{
  Container contentPane = getContentPane();
  if (e.getActionCommand().equals("Red"))
  {
    contentPane.setBackground(Color.red);
    theMessage ="   Danger! STOP!";
  }
  else if (e.getActionCommand().equals("Green"))
  {
    contentPane.setBackground(Color.green);
    theMessage = "  Green means GO";
  }
  else if (e.getActionCommand().equals("Blue"))
  {
    contentPane.setBackground(Color.blue);
    theMessage = "my favourite colour";
  }
  else
    theMessage = "Error in button interface";
}
```

The method `getActionCommand` returns the label on the button that was clicked to fire the event. This is used to determine which button was clicked and change the background and the content of the text variable (`theMessage`). The `paint` method is automatically called for the window to be redrawn to display the new background and message. Note that the call to the super class `super.paint(g)` makes sure that other graphical objects are also updated in the window.

16.5 Panels

A panel is a container and a component, so you add panels to your window to subdivide it into different areas that you can add components to and deal with independently. For example, we can put buttons into a panel and locate that panel in a particular area of our window, and display any data in another area of the window.

Consider the constructor for the class `JPanelDemo` in `JPanelDemo.java` given on the next page. Here we define a frame as before, and then create a panel within the frame and add it to the window according to the border layout manager which is defined for the window.

```
// define a panel within the frame
   JPanel buttonPanel = new JPanel();
   buttonPanel.setBackground(Color.white);
      :
      :
// define layout for the full frame
   contentPane.setLayout(new BorderLayout());
// add panel at the bottom
  contentPane.add(buttonPanel,"South");
```

We can define a different layout manager for the panel (the default is generally FlowLayout):

```
buttonPanel.setLayout(new FlowLayout());
```

create and add action listeners to the buttons:

```
redButton.addActionListener(this);
```

and add the buttons explicitly to the panel:

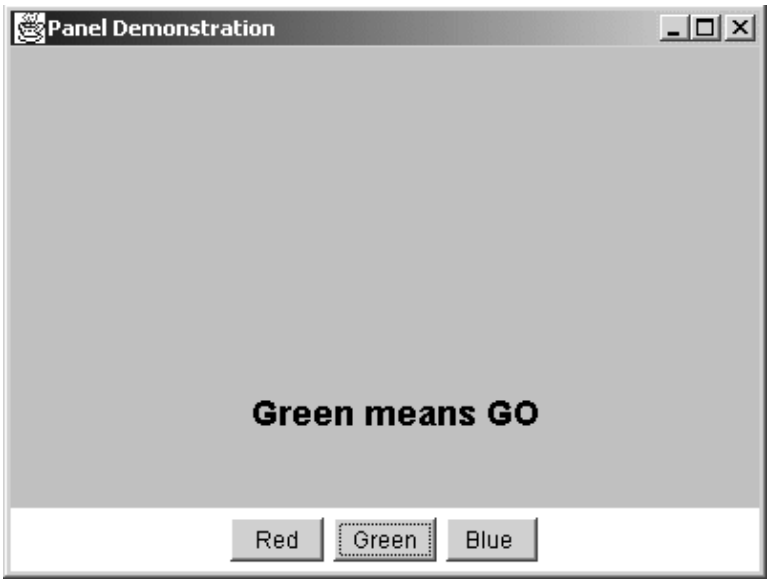
```
buttonPanel.add(redButton);
```

Because the action listeners alter the background of the content pane i.e. the window (referring to `this` inside the window constructor implies the window itself), only the window's background is changed. The panel background is unchanged.

To summarize, a `JFrame` represents a window that has a `ContentPane`. We can add graphical objects or components such as buttons, labels etc.. or even panels to the window via the `JFrame`'s `ContentPane`. The `ContentPane` forms the main container for the window (`JFrame`). If we add a `panel` to the `ContentPane`, it represents a sub-container within the main container. Now we can add graphical components to the main container (the `contentPane`) or the sub-container (the panel).

```
   public JPanelDemo()
   //----------------
   // constructor - sets the window size & title
   {
      …

      // get the content pane from the window
      Container contentPane = getContentPane();

      contentPane.setBackground(Color.lightGray);
      // layout manager to arrange components

      contentPane.setLayout(new BorderLayout());

     // define a panel within the frame
      JPanel buttonPanel = new JPanel();
      buttonPanel.setBackground(Color.white);

      // layout manager to arrange components in panel
      buttonPanel.setLayout(new FlowLayout());

      // add buttons to panel
      redButton.addActionListener(this);
      buttonPanel.add(redButton);

      greenButton.addActionListener(this);
      buttonPanel.add(greenButton);

      blueButton.addActionListener(this);
      buttonPanel.add(blueButton);

       // add panel at the bottom
      contentPane.add(buttonPanel,"South");
   }
}
```

part of *JPanelDemo.java*



16.6 Text Areas, Text Fields and Labels

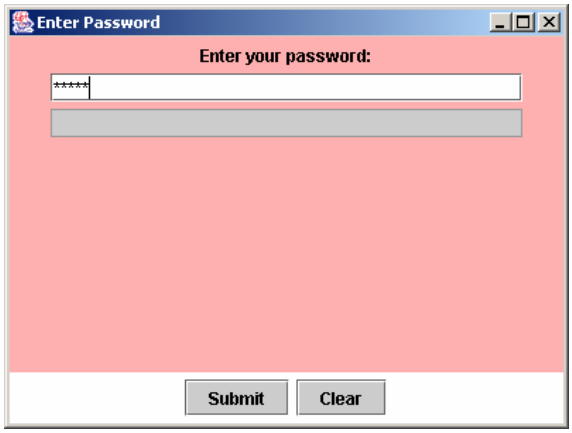
To allow the user to enter data into a window we need to define a text area or a text field into which the user can enter text data. The difference between the two is that `JTextArea` allows for multiple lines of data, whereas `JTextField` caters for a single line only.

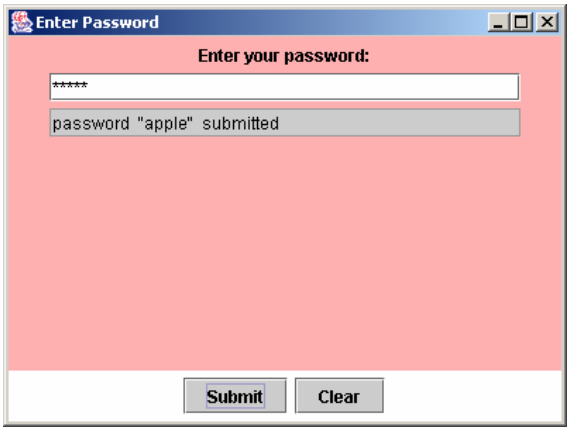
This program defines 2 panels, one containing a label and two text fields and the other containing some buttons.

```
public JTextDemo()
//----------------
// constructor - sets the window size & title
{
   setTitle("Enter Password");
   setSize (400,300);
   addWindowListener(new WindowDestroyer());
   // get the content pane from the window
   Container contentPane = getContentPane();

   contentPane.setBackground(Color.lightGray);
  // layout manager to arrange components
   contentPane.setLayout(new BorderLayout());
   // define a panel to hold the text field and label
   JPanel textPanel = new JPanel();
   textPanel.setBackground(Color.pink);
   textPanel.setLayout(new FlowLayout());

   JLabel pwLabel = new JLabel("Enter your password:");
   textPanel.add(pwLabel);
   password = new JPasswordField(30);
   textPanel.add(password);
   status = new JTextField(30);
   status.setEditable(false);
   textPanel.add(status);

   // add text panel to frame
   contentPane.add(textPanel,"Center");

   // define another panel to hold the buttons
   JPanel buttonPanel = new JPanel();
   buttonPanel.setBackground(Color.white);
   buttonPanel.setLayout(new FlowLayout());

   // add buttons to panel
   JButton b = new JButton("Submit");
   b.addActionListener(this);
   buttonPanel.add(b);

   b = new JButton("Clear");
   b.addActionListener(this);
   buttonPanel.add(b);

   // add button panel to frame
   contentPane.add(buttonPanel,"South");
}
```

part of *JTextDemo.java*

The constructor first defines the basic window requirements, including the layout manager to be used when adding components to the window. It then creates a panel (`textPanel`) and defines its background colour and layout manager. Next various components are defined and added to this panel: first a label (`pwLabel`) is constructed with some text and then a specialised text field `JPasswordField` of a given size (30) is constructed. The JPasswordField does not display the actual characters entered by the user. The `password` variable is an instance variable and is declared outside the constructor. This is because it needs to be referenced in the `actionPerformed` method and this could not be done if it were local to the constructor. Another general text field `JTextField`, called `status`, is constructed and set to not editable (textfields are editable by default). Note that `status` is an instance variable as well, and is also modified in the `actionPerformed` method. The `textPanel` is added to the window and another panel (`buttonPanel`) is defined to hold some buttons in a manner similar to before.





As before, the method `actionPerformed` defines the actions to take place when a button is clicked

```
public void actionPerformed(ActionEvent e)
//---------------------------------------
// determine which button was clicked and take appropriate action
{
   if (e.getActionCommand().equals("Submit"))
   {
      String pword = new String(password.getPassword());
      status.setText("password  \""+pword+"\"  submitted");
   }
   else if (e.getActionCommand().equals("Clear"))
   {
      password.setText("");
      status.setText("password  cleared");
   }
   else
      status.setText("Error in button interface");

}
```

Both text areas and text fields have `getText` and `setText` methods that return the text entered as a string, or display a string in the text area/field. If the submit button is clicked the contents of the text field are obtained and a confirmation message is sent to the `status` text field. If the clear button is clicked an empty string is stored in the `password` text field.

If you want to enter numbers in a text field/area they must be converted from a string to a numeric value. Similarly a number must be converted to a string to display it in a text field/area.

Appendix A : Errors and Testing

A.1 Coding for testing

Studies of the programming process have shown that design and initial coding of a program account for 20% of the total program development cycle; testing and debugging over 80%.

These results are applicable to large software projects. Generally the time spent in testing and debugging is not nearly so large in smaller application program development efforts. However the development of good habits in small projects will be well worth the effort when a large project is tackled. To increase our ability to debug programs quickly there are four techniques:

- work modularly.
- code and test incrementally
- trace parameter values on procedure entry and exit
- where available, use debugging aids and tools

We must assume that the programs we write have bugs. This is a normal human characteristic. Our programming techniques must provide the support to reduce bugs originally, known as ***anti-bugging***, and to more easily detect and correct them when they occur, known as ***debugging***.

We can think of software errors as having three conceptual sources.

- internal procedure errors
- inter-procedure communication errors
- environmental errors

Short of catastrophic failures to the physical environment, the goal of all programs during execution must be that they work reasonably and well regardless of whether they are being abused by the user, by the operating system, or by the run-time collection of functions and routines themselves. We consider:

- input validation
- arithmetic problems: over/underflow; roundoff errors

Input Validation – Range/Validity Checking

It is the normal expectation of programmers when they first begin designing a program that data values will be within the range they expect. Unfortunately this might not be the case for a number of reasons:

- users may not understand or know the allowable data range.
- there may be a simple typing error.
- the hardware may make some unexpected transformations to the data.

Consequently, all data that a program expects must be treated as suspect. The techniques for validating input are straightforward but they contain a number of hidden assumptions and have the typical side effect of greatly increasing the size of the code. As a result , the code can be less clear and its meaning and intent and its structure can become more obscure.

The problem is often not in writing code to verify that values are within range. Rather, the problem comes in deciding what to do when an erroneous out-of-range value is found. There are a number of choices:

- stop
- ignore
- fix up: don't tell the user
- fix up: tell the user
- fix up: ask if ok
- force program into a loop until ok
- flag an exception; replace with default values and continue

Programs designed with consideration for user errors are likely to be much more complex in structure, and much more lengthy, than programs that assume all input is correct in number and format.

Run-Time Arithmetic Errors

Overflow is the result of an arithmetic operation that yields a result that is too large to be represented in the normal machine representation.

In most languages, overflow occurs in integer arithmetic when an operation produces a result large than *maxint.* For example:

```
TooBig = 2 * maxint / N;
```

produces run-time arithmetic overflow even if N is larger than two.

One might think that we can guard against overflow by conversion to doubles. Suppose that `EvenBigger` has been declared as a *double* variable. Then we could replace the following assignment by:

```
EvenBigger = 2.0 * maxint / N;
```

Then this delays overflow but it will still occur at some stage.

A more subtle and potentially more troublesome set of errors relates to finite representation and the associated problem of ***roundoff errors***. This occurs because not all values can be represented exactly in a finite number of bits.
For example, $0.1_{10} \cong 0.00011001100110011001\ldots_2$ and does not have an exact binary representation.

Consider the following Java statements

```
double total = 0.0;
while (total != 1.0)
{
  total = total + 0.1;
  System.out.println(total);
}
```

When this is executed the results obtained are

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2
1.3
1.4000000000000001
1.5000000000000002
1.6000000000000003
......
```

(infinite loop)

The problem occurs because of the inexact representation of 0.1. Each time the approximation to 0.1 is added to `total`, `total` becomes less and less accurate. After 10 iterations when we think it should be equal to 1.0 it is not exactly equal to 1.0 and so the loop doesn't terminate.

Because of roundoff errors you should avoid comparing floating point numbers for equality or inequality in your programs. There are 3 possible ways to handle this:

- use integers instead – since a computer can represent an integer exactly there are no roundoff errors. However the use of integers is not always practical
- use <= or >= instead of ==, or use < or > instead of `!=`. It often doesn't matter if a number is exactly equal to another, we just need to check if it is at least as large as (or as small as) the other number
- check whether the two numbers are very close instead of exactly equal by checking that the difference between them is very small. For example, change the condition in the while loop above to

  ```
  while (Math.abs(total-1.0)>0.00001)
  ```

  With this condition the loop will continue while the absolute difference between the two values is above some threshold. The threshold value can be chosen as small as is appropriate given the program requirements.

Another problem with roundoff error occurs when working with numbers of different sizes. For example, suppose we wish to add $1x10^{10}$ and 4.0. These numbers must be converted to the same exponent before being added:

$$\begin{array}{rll} & 1.0000000 & \times 10^{10} \\ + & 0.0000000004 & \times 10^{10} \end{array}$$

Assuming that the computer can store only 8 significant digits, the last 3 digits of the sum

$$1.0000000(004) \times 10^{10}$$

will be lost resulting in an answer of 10000000000, the original value.
And even if we were to add 4 to 10000000000 a total of a million times (when we would expect the answer to be 10004000000, the result will still appear as 10000000000 because each time 4 is added to 10000000000 the loss of significance causes the value to be lost.

A.2. Debugging

Programs that are modular, clear and well-documented are certainly easier to debug than those that are not. Fail-safe techniques that guard against certain errors and report them when they are encountered are also a great aid in debugging.

Many students seem totally baffled by bugs in their programs and have no idea on how to proceed – they simply have learned how to track down a bug systematically. Without a systematic approach, finding a small bug in a large program can indeed be a difficult task. The difficulty that many people have in debugging a program is perhaps in part due to a desire to cling to the belief that their program is really doing what it's supposed to do.

The trick to debugging is simply to use a program's output to tell us what's going on. This may sound mundane, but the real trick is to use the program's output in an effective manner. After all, you don't simply put output statements at random points of the program and have them report random information.

We want to use the output to zero in on the points in the program that are causing the problem. A program's logic implies that certain conditions should be true at various points in the program (these conditions are called invariants). A bug means that a condition, which we think ought to be true isn't. By inserting output statements at strategic locations of a program we can systematically zero in on the bug. The placement and content of each output statement should be such that it informs us of whether things start going wrong before or after a given point in the program.

Thus, after we run the program with an initial set of diagnostic output statements, we should be able to find two points between which a bug occurs. We continue this process of placing diagnostic output statements until the search is limited to just a few statements. The ability to place diagnostic output statements and to have them report appropriate information comes in part from thinking logically and in part from experience.

Here are a few guidelines:

What an output statement should report

An output statement should be used to report both the values of key variables and the location in the program at which the variables have those values.

```
System.out.println ("At point A in method Compute:");
System.out.println ("x = "+x+" y = "+y+" z = "+z);
```

Output statements in methods

Two key locations to place output statements are at the beginning and end of a method:

```
void P (…)
{
  System.out.println ("Values on entering method P  : "+ ...);
     :
  System.out.println ("Values on leaving method P  : "+ ...);
 }
```

Output statements in if-statements

Output statements should be placed before the test statement and should be used to report the branch taken as a result of the test. The values of the variables involved in the test should be reported.

```
System.out.print ("IF test based on X = "+X+" and Y = "+Y);
if (X>Y)
{
  System.out.println ("  TRUE branch taken");
    :
}
else
{
  System.out.println ("  FALSE branch taken");
    :
}
```

Output statements in loops

Output statements should be placed at the beginning and end of loops. The values of the control variables should be reported. The statements should occur both inside and outside the loop.

```
System.out.println ("Entering WHILE loop, x  =  "+x);
while (x>0)
{
  System.out.println ("Inside WHILE loop, x  =  "+x);
  :
}
System.out.println ("Exiting WHILE loop, x  =  "+x);
```

A.3. Testing

> *"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."*
>
> EW Dijkstra

What is testing?
A process of inferring certain behavioural properties of a product based, in part, on the result of executing the product in a known environment with selected inputs.

- testing is an inferential process – tester takes the program, runs it with known data, examines the output and from this tries to infer what, if anything, is wrong.
- the environment is not always "known" or controlled. There may be an intermittent hardware fault, or a run-time routine may be incorrect. The results you are seeing may be the result of a correct product running in a faulty environment.
- the "selected inputs" must be carefully chosen to be representative. Problems in a real-time environment – how do you test a system effectively that must respond to digitised data from sensors, whether temperature gauges in a factory or flight information in a rocket system.

What is meant by a correct program?

- Utility – the extent to which a user's needs are met when a correct program is used under circumstances for which it was designed/written.
- Reliability – a measure of the frequency and criticality of product failure.
  - mean time between failures – how often it fails
  - mean time to repair – how long it takes to repair it
  - mean time to repair the results of failure – is occasional failure with catastrophic results better than more frequent failures with minimal effect?
- Robustness – how well it deals with invalid input/incorrect usage etc.
- Performance – whether it meets constraints of response time or space requirements
- Correctness – if input satisfying the input specifications is provided, and the program is given all the resources it needs, then it is correct if the output satisfies the output specifications.

Black box testing (testing to specification)

Program is treated as a closed box with no knowledge of internal structure. Testing consists of feeding input and noting what output is produced. Must ensure that every type of input is used and that the output matches the expected output – potentially billions of test cases. Must attempt to set up a small, manageable set of test cases so as to maximise the chances of detecting a fault while minimising the number of tests run.

- Define equivalence classes – ie sets of test cases such that any one member of the class is as good as any other Eg. If program should be able to handle values between 1 and 1000 there are 3 equivalence classes:
  1. less than 1
  2. between 1 and 1000
  3. over 1000

- analyse boundary values – errors are more likely to occur with a test case that is either on or just to one side of the boundary of an equivalence class; ie with the above example 7 test cases are indicated:
  1. 0 : equivalence class 1 and adjacent to boundary
  2. 1 : boundary value
  3. 2 : adjacent to boundary
  4. 423 : equivalence class 2
  5. 999 : adjacent to boundary
  6. 1000 : boundary value
  7. 1001 : equivalence class 3 and adjacent to boundary

Another consideration is the output specifications – for example in a tax program after various deductions, rebates exemptions, surcharges and levies, the minimum deduction may be R0.00 and the maximum R4555.00. Then test cases should be constructed that result in deductions of exactly R0.00 and R4555.00 as well as <R0.00 and >R4555.00.

In general, for each range ($R_1..R_2$) listed in the input and output specifications 5 test cases should be set up: $< R_1$; $=R_1$; $>R_1$ and $<R_2$; $=R_2$; $>R_2$
Where it is specified that an item has to be a member of a given set (eg the input must be a digit char) there are 2 equivalence classes – a member of the set and a non-member, and where a precise value is given there are also 2 classes – the exact value and anything else.

Glass box testing (structural testing)

Test cases are selected on examination of the code, rather than the specifications.

- statement coverage – ensure that each statement is executed
- branch coverage – ensure that all branches are tested at least once
- path coverage – tests all paths through the program

Consider

```
1.  do
2.  { flag = false;
3.    if (x>y)
4.      flag = true;
5.    x++;
6.    Calculate(x,flag,answer);
7.    if (answer>0)
8.      DisplayResult(answer);
9.  } while (answer<=0);
```

To ensure statement testing we need to choose a value of x larger than y so that statements

1 2 3 4 5 6 7 8 9

will be executed.

For branch testing we need 2 cases that will test sequences

1 2 3 4 5 6 7 8 9

and 1 2 3 5 6 7 9 2

And for path testing 4 cases are needed (2 decision points with 2 possibilities at each branch).

1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 9 2
1 2 3 5 6 7 8 9
1 2 3 5 6 7 9 2

Combined black box / glass box testing

Black box testing suffers from uncertainty about whether the test cases selected will uncover a particular error, while glass box testing allows the danger of paying too much attention to the code's internal processing – may end up testing what the program *does* rather than what it *should do*.

To combine both, first use the external specifications to generate initial test cases including representatives from each equivalence class and boundary values (including invalid data). Next by viewing the internal structure of the program add data to test all decision points and as many paths through the program as possible. Consider the implementation of algorithms – if an algorithm is used that does not behave well at a particular point include cases that test the extremities of the algorithm.